

1. Osnovna struktura C++ programa

Uobičajeno je da svaka priča o programskom jeziku C++ počinje opisom kratkog programa koji ispisuje pozdravnu poruku na ekranu. Stoga će naš prvi program izgledati ovako:

```
// Ovaj program ispisuje pozdravnu poruku
#include <iostream>
using namespace std;

int main()
{
    cout << "Zdravo narode!";
    return 0;
}
```

Čak i u ovako kratkom programu, moguće je uočiti nekoliko karakterističnih elemenata:

// Ovaj program ispisuje pozdravnu poruku	<i>Komentar</i>
#include <iostream>	<i>Zaglavlj standardne biblioteke</i>
using namespace std;	<i>Deklaracija standardnog imenika</i>
int main()	<i>Zaglavlj funkcije</i>
{	
cout << "Zdravo narode!";	<i>Naredba 1</i>
return 0;	<i>Naredba 2</i>
}	

Znak “//” označava *komentar*. Prilikom prevodenja programa, kompjajler *potpuno ignorira* sve što je napisano iza znaka za komentar do kraja tekućeg reda. Svrha komentara je da pomogne ljudima koji analiziraju program da lakše shvate šta program ili njegovi pojedini dijelovi rade. Komentari su obično kratki. Postoji još jedan način za pisanje komentara, naslijeden iz jezika C. Sve što se nalazi između oznaka “/*” i “*/” također se tretira kao komentar. Ovakav način pisanja komentara može biti pogodan za pisanje komentara koji se protežu u više redova teksta, kao na primjer u sljedećem programu:

```
/* Ovo je veoma jednostavan C++ program.
On ne radi ništa posebno, osim što ispisuje pozdravnu poruku.
Međutim, od nečeg se mora početi. */

#include <iostream>
using namespace std;

int main()
{
    cout << "Zdravo narode!";      // Ispisuje tekst "Zdravo narode!"
    return 0;                      // Vraća 0 operativnom sistemu
}
```

Glavninu programa sačinjava cjelina nazvana *funkcija*. Funkcije predstavljaju funkcionalne cjeline programa koje objedinjavaju skupine naredbi koje su posvećene obavljanju nekog konkretnog zadatka. Prikazani program ima samo jednu funkciju. Iole složeniji programi gotovo uvijek imaju više funkcija, jer je veće programe mnogo lakše pisati ukoliko se prethodno razbiju na manje funkcionalne cjeline koje se kasnije izvode kao različite funkcije. U svakom C++ programu mora postojati tačno jedna funkcija koja se

zove “`main`”. Ova funkcija se naziva *glavna funkcija*, i to je ona funkcija koja će se prva početi izvršavati kada program započne sa radom. U našem primjeru, to je ujedno i jedina funkcija u programu. Drugim riječima, program uvijek započinje izvršavanje od *prve naredbe glavne funkcije*.

Funkcije u programskom jeziku C++ uvijek započinju *zaglavljem* (engl. *function heading*) koje pored imena funkcije sadrži *spisak parametara funkcije* (engl. *parameter list*), i *tip povratne vrijednosti* (engl. *return type*). Parametri su podaci koje pozivaoc funkcije treba da proslijedi pozvanoj funkciji, sa ciljem preciziranja zadatka koji funkcija treba da obavi. Popis parametara se navodi u zagradama neposredno iza imena funkcije. Pošto se `main` funkcija poziva *neposredno iz operativnog sistema*, eventualne parametre ovoj funkciji može proslijediti jedino operativni sistem. Stoga je najčešći slučaj da funkcija `main` *nema nikakvih parametara*, što se označava praznim zagradama (odsustvo parametara se ponekad označava i navođenjem riječi “`void`“ unutar zagrada, što je zastarjela praksa nasljedena iz jezika C). O parametrima ćemo detaljnije govoriti u kasnijim poglavljima.

Ispred imena funkcije navodi se *tip povratne vrijednosti funkcije*. Povratna vrijednost je vrijednost koju funkcija vraća pozivaocu funkcije. Vraćanje vrijednosti obavlja se pomoću naredbe “`return`”, koja tipično predstavlja posljednju naredbu unutar funkcije, i iza koje se navodi povratna vrijednost. U slučaju funkcije “`main`”, povratna vrijednost se vraća operativnom sistemu. Po dogovoru, vrijednost “0” vraćena operativnom sistemu označava *uspješan završetak programa*, dok se vrijednosti različite od nule vraćaju operativnom sistemu isključivo kao signalizacija da iz nekog razloga program nije uspješno obavio planirani zadatok (značenje ovih vrijednosti zavisi od konkretnog operativnog sistema). Stoga je posljednja naredba funkcije “`main`” u gotovo svim slučajevima naredba

```
return 0;
```

Sve naredbe jezika C++ obavezno završavaju znakom “;”, koji označava kraj naredbe (jedna od najčešćih sintaksnih grešaka u C++ programima je upravo zaboravljanje završnog tačka-zareza na kraju naredbe). Riječ “`int`“ ispred imena funkcije označava da je vrijednost koju ova funkcija vraća *cijeli broj*, kao što će biti jasnije iz sljedećeg poglavlja.

Kako je povratna vrijednost iz “`main`“ funkcije gotovo uvijek nula, mnogi kompjajleri dopuštaju da se naredba “`return`“ potpuno izbaci iz “`main`“ funkcije. Tako nešto ne smatra se dobrom programerskom praksom. Također, ranije verzije kompjajlera za C++ dozvoljavale su da se naredba “`return`“ izbaci i da se tip povratne vrijednosti “`int`“ zamjeni sa “`void`“, što označava da se nikakva povratna vrijednost ne vraća. Standard ISO C++98 zabranio je ovaku praksu, i strogo naredio da funkcija “`main`“ kao povratni tip može imati samo “`int`”, i *ništa drugo*.

Nakon zaglavlja funkcije, slijedi *tijelo funkcije* (engl. *function body*). Tijelo funkcije počinje znakom “{“ a završava znakom “}”, odnosno tijelo funkcije uvijek je omeđeno *vitičastim zagradama*. Kasnije ćemo vidjeti da se vitičaste zgrade koristimo kada god treba objediniti neku skupinu naredbi u jednu kompaktnu cjelinu. U ovom slučaju ta cjelina će biti upravo tijelo funkcije. Pod naredbama podrazumijevamo upute koje govore računaru *kakvu akciju treba da preduzme*, odnosno naredbe *opisuju algoritam*. Kao što vidimo, program se ne sastoji samo od naredbi. Na primjer, komentar ili zaglavlje funkcije nisu naredbe, jer ne dovode do *nikakve konkretne akcije*. Program u našem primjeru je toliko jednostavan da ima samo dvije naredbe, od kojih je druga naredba praktično trivijalna. Analizirajmo stoga jedinu netrivijalnu naredbu u programu, koja glasi:

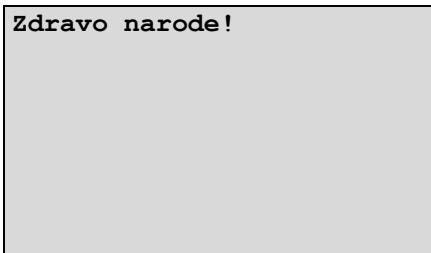
```
cout << "Zdravo narode!" ;
```

“`cout`“ (skraćenica od engl. *console out*) predstavlja objekat tzv. *izlaznog toka podataka* (engl. *output stream*), koji je povezan sa standardnim uređajem za ispis. Standardni uređaj za ispis je tipično ekran, mada ne mora biti (C++ uopće ne podrazumijeva da računar na kojem se program izvršava mora

posjedovati ekran). Znak “`<<`” predstavlja *operator umetanja* (engl. *insertion*) u izlazni tok, koji pojednostavljeno možemo čitati kao “šalji na”. Tekst “*Zdravo narode!*” koji se nalazi između navodnika, predstavlja niz znakova koji šaljemo na (odnosno umećemo u) izlazni tok. Takav niz znakova nazivamo *stringovni literal* (ili *neimenovana stringovna konstanta*). O stringovnim literalima i stringovima općenito će kasnije biti više govora, a za sada će nam stringovni literali služiti samo za ispis fiksnog teksta na izlazni uređaj. Dakle, gornja naredba može se interpretirati kao:

Pošalji niz znakova “Zdravo narode!” na izlazni uređaj (tipično ekran).

Uz pretpostavku da smo uspješno kompajlirali i pokrenuli program (tačan postupak kako se to radi ovisi od upotrijebljenog kompajlera), na ekranu bi se trebao pojaviti ispis poput sljedećeg:



U slučaju da radite pod nekim grafičkim operativnim sistemom (poput Windows-a), ovaj tekst će se tipično pojaviti u posebnom prozoru nazvanom *konzolni prozor*. Moguće je da u slučaju kompajlera sa kojim radite nećete biti u stanju da vidite nikakav tekst, jer se konzolni prozor obično automatski zatvara nakon završetka programa, čime i ispisani tekst odlazi u nepovrat. O načinima za prevazilaženje ovog problema govorićemo u narednom poglavlju, a u međuvremenu, djelimično rješenje može biti ubacivanje sljedeće naredbe neposredno ispred naredbe “`return`”:

```
cin.get();
```

Ubacivanjem ove naredbe postići ćemo da se program neće završiti prije nego što korisnik pritisne tipku ENTER, čime smo odložili zatvaranje konzolnog prozora. Ovo nije zapravo pravi smisao ove naredbe, već smo se ovdje oslonili na jednu njenu propratnu pojavu. O njenom tačnom smislu govorićemo u kasnijim poglavljima.

Objekat “`cout`” ne mora nužno prihvataći samo stringovne literali, nego i mnoge druge tipove podataka o kojima će biti više riječi kasnije. Na primjer, pored stringovnih literalova, biće prihvaćeni i *brojevi* (*brojčani literalni*), tako da možemo pisati:

```
cout << "Proba";
cout << 1234;
cout << 12.34;
```

U gornjem primjeru “`12.34`” je decimalni broj. Primijetimo da se za odvajanje decimala od cijelog dijela ne koristi zarez nego tačka. O pravilima za pisanje brojeva govorićemo detaljnije u poglavljima koja slijede.

Operator umetanja u tok može se primijeniti više puta zaredom na “`cout`” objekat, tj. možemo pisati

```
cout << "Broj je " << 10;
```

Ovo će proizvesti sljedeći ispis na izlazni uređaj:

```
Broj je 10
```

Veoma je bitno da se na samom početku shvati razlika između objekata kao što su 10 i "10". Objekat 10 je *broj* i nad njim se mogu vršiti računske operacije poput sabiranja itd. S druge strane, objekat "10" je *niz znakova (stringovni literal)* koji se slučajno sastoji od dvije cifre (1 i 0), i nad njim se ne mogu vršiti nikakve računske operacije (npr. telefonski broj je sigurno sastavljen od cifara, ali teško kome će pasti na pamet da sabira dva telefonska broja zajedno). Ova razlika će biti mnogo jasnija nešto kasnije. Ipak, sa aspekta ispisa ova dva objekta se ponašaju identično, tako da bi isti efekat proizvela i sljedeća naredba:

```
cout << "Broj je " << "10";
```

ili prosto

```
cout << "Broj je 10";
```

Stringovni literali mogu sadržavati potpuno proizvoljan niz znakova, tako da je npr. "! ?@ #" sasvim legalan stringovni literal, iako je njegov smisao upitan. Grubo rečeno, može se shvatiti da navodnici označavaju "bukvalno to", "baš to" itd. Kada god nisu upotrijebljeni navodnici, kompjajler za C++ će pokušati da interpretira smisao onog što je napisano prije nego što se izvrši ispis (o čemu će kasnije biti više riječi), dok kada se upotrijebe navodnici, na izlazni uređaj se bukvalno prenosi ono što je napisano između njih. Tako npr. ako napišemo

```
cout << "2+3";
```

na ekranu ćemo dobiti ispis

```
2+3
```

dok ako napišemo

```
cout << 2+3;
```

prikaz na ekranu će biti

```
5
```

Naravno, naredba poput

```
cout << @#!&&/3++;
```

doveće do prijave greške od strane kompjajlera, jer je kompjajler pokušao (bezuspješno) da interpretira smisao napisanih znakova, dok je s druge strane,

```
cout << "@#!&&/3++";
```

posve legalna naredba. Da bismo ovo još jednom istakli, uzmimo da želimo da ispišemo telefonski broj sa crticom. Moramo pisati

```
cout << "453-728";
```

a ne nipošto

```
cout << 453-728;
```

jer će u posljednjem slučaju “-” biti shvaćen kao znak za oduzimanje, tako da ćemo umjesto telefonskog broja imati ispisani rezultat oduzimanja (tj. -275).

Ovim još nismo objasnili sve elemente našeg prvog programa. Prvi red programa ne računajući komentar je glasio:

```
#include <iostream>
```

Naredba “#include“ predstavlja *uputu* (tzv. *direktivu*) kompjajleru da u program uključi tzv. *zaglavlj biblioteke* (engl. *library header*) sa imenom “*iostream*”. Naime, sve funkcije i objekti u jeziku C++ grupirani su u biblioteke, i na početku svakog programa obavezno se uključuju sve biblioteke koje sadrže funkcije i objekte koje koristimo u programu. Biblioteka “*iostream*“ (skraćeno od engl. *input-output stream*), između ostalog, definira objekat “*cout*“ koji koristimo za pristup izlaznom toku podataka. Tako, npr. ako pišemo matematički program, na početku programa će također stajati i

```
#include <cmath>
```

a ako pišemo program koji radi sa grafikom, vjerovatno ćemo imati i nešto poput:

```
#include <graphics.h>
```

Ideja svega ovoga je da se program ne opterećuje bespotrebno objektima i funkcijama koji se neće koristiti unutar programa. Npr. bespotrebno je program opterećivati grafičkim objektima ako program neće koristiti grafiku. Standard ISO C++98 jezika C++ predviđa da se kao sastavni dio jezika C++ obavezno moraju nalaziti sljedeće biblioteke:

algorithm	bitset	cassert	cctype	cerrno	cfloat
ciso64	climits	locale	cmath	complex	csetjmp
csignal	cstdarg	cstddef	cstdio	cstdlib	cstring
ctime	cwchar	cwctype	deque	exception	fstream
functional	iomanip	ios	iosfwd	iostream	istream
iterator	limits	list	locale	map	memory

```
new           numeric      ostream      queue      set          sstream
stack         stdexcept    streambuf   string     typeinfo    utility
valarray      vector
```

U starijim verzijama kompjajlera za C++ sva zaglavla biblioteka imala su nastavak “.h“ na imenu, tako da se umjesto zaglavla “`iostream`“ koristilo zaglavljje “`iostream.h`“. Novi kompjajleri još uvijek podržavaju stara imena zaglavla, ali njihovo korištenje se ne preporučuje, jer će podrška za stara imena biti izbačena u doglednoj budućnosti. Kompajler ima pravo da upozori programera na to da se upotreba starih zaglavla ne preporučuje emitiranjem poruke upozorenja. Sve biblioteke čija imena počinju slovom “c” osim “`complex`“ naslijedena su iz jezika C, u kojima su imali imena zaglavla bez početnog slova “c” i sa nastavkom “.h“ (npr. ”`math.h`“ umjesto ”`cmath`“). I u ovom slučaju, stara imena zaglavla se još uvijek mogu koristiti, ali tu praksi treba izbjegavati.

Pored navedenih 50 standardnih biblioteka, mnogi kompjajleri za C++ dolaze sa čitavim skupom *nestandardnih biblioteka*, koje ne predstavljaju propisani standard jezika C++. Tako, na primjer, biblioteka za rad sa grafikom sigurno ne može biti unutar standarda jezika C++, s obzirom da C++ uopće ne podrazumijeva da računar na kojem se program izvršava mora imati čak i ekran, a kamoli da mora biti u stanju da vrši grafički prikaz. Također, biblioteka sa zaglavljem “`windows.h`“ koja služi za pisanje Windows aplikacija ne može biti dio standarda C++ jezika, jer C++ ne predviđa da se programi moraju nužno izvršavati na Windows operativnom sistemu. Zaglavla nestandardnih biblioteka gotovo uвijek imaju i dalje nastavak “.h“ na imenu, da bi se razlikovala od standardnih biblioteka. Treba napomenuti i to da se nestandardne biblioteke za razne namjene često mogu nabaviti i kod neovisnih proizvođača softvera, ili besplatno skinuti sa Interneta. Očigledno su nestandardne biblioteke često itekako potrebne, ali ne mogu da uđu u dio standarda jezika C++, s obzirom da standard jezika ne smije da sadrži nikakve pretpostavke o hardverskim osobinama računara na kojem se program izvršava, niti o operativnom sistemu koji se izvršava na računaru.

Strogo uvezši, direktiva “`#include`“ ne čini sastavni dio jezika C++, nego čini tzv. *naredbu preprocesora*. Preprocesor je program koji vrši početnu obradu C++ programa još prije nego što C++ program uopće bude prihvaćen od strane kompjajlera. Kako je preprocesor obično tjesno vezan za kompjajler, korisnik najčešće nije svjestan ove činjenice (i ne mora da bude svjestan, osim u veoma naprednim primjenama). Sve naredbe preprocesora počinju znakom “#“ koji se, inače, čita kao “heš” (engl. *hash*).

Preostaje još da objasnimo red programa koji je glasio:

```
using namespace std;
```

Ovaj programski red govori programu da koristi *imenik* (engl. *namespace*) nazvan “`std`”. Imenici su osobina jezika C++ koja je uvedena tek nedavno. Naime, pojavom velikog broja nestandardnih biblioteka različitim proizvođača bilo je nemoguće spriječiti konflikte u imenima koje mogu nastati kada dvije različite biblioteke upotrijebi isto ime za dva različita objekta ili dvije različite funkcije. Zbog toga je odlučeno da se imena svih objekata i funkcija razvrstavaju u *imenike*. Dva različita objekta mogu imati isto ime, pod uvjetom da se nalaze u različitim imenicima. Standard propisuje da se svi objekti i sve funkcije iz standardnih biblioteka moraju nalaziti u imeniku “`std`”. Pomoću navedenog programskog reda govorimo kompjajleru da želimo da sve funkcije i objekti iz ovog imenika budu vidljivi u našem programu. Bez ovog reda, objektu “`cout`“ bismo morali pristupati pomoću konstrukcije “`std::cout`”, koja znači “objekat `cout` iz imenika `std`“. Alternativno bismo umjesto prethodnog programskog reda mogli pisati i sljedeći red:

```
using std::cout;
```

Ovim bismo naglasili da želimo pristupati objektu “cout“ iz imenika “std“ (bez kasnijeg stalnog navođenja imena imenika pri svakom pristupu cout objektu), bez potrebe da “uvozimo” čitav imenik “std”.

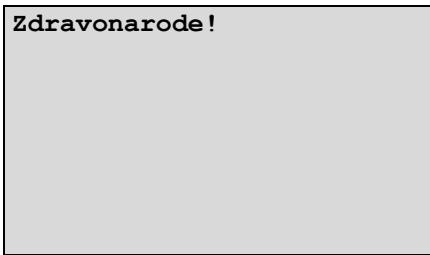
Da uočimo još neke osobine objekta “cout”, napisaćemo ponovo program koji radi istu stvar kao i program iz prvog primjera, ali na nešto drugačiji način:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Zdravo ";
    cout << "narode!";
    return 0;
}
```

Ovaj program proizvodi potpuno isti ispis kao i prethodni program, jer svaka sljedeća naredba za ispis nastavlja sa ispisom tačno od onog mjesta gdje se prethodna naredba za ispis završila. Razmak iza “o” u prvoj naredbi je bitan. Naime, i razmak je *dio stringovnog literalja*, i kao takav *bićeписан* na izlazni uređaj. Da smo izostavili taj razmak, tj. da smo glavnu funkciju napisali ovako:

```
int main()
{
    cout << "Zdravo";
    cout << "narode!";
    return 0;
}
```

tada bi ispis na ekran izgledao otprilike ovako:



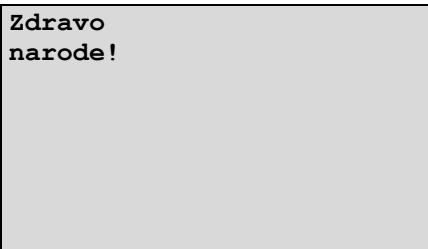
Važno je napomenuti da su svuda osim unutar stringova i šiljatih zagrada (<>), dodatni razmaci potpuno nebitni, i kompjaler ih ignorira, što omogućava programeru stiliziranje izgleda programa u svrhu poboljšanja njegove čitljivosti. Zapravo, isto vrijedi ne samo za razmace nego i za sve tzv. *bjeline* (engl. *whitespaces*), u koje osim razmaka spadaju tabulatori i oznake za kraj reda. Tako se naredbe koje čine tijelo funkcije obično pišu *neznatno uvučeno*, da se vizuelno istakne početak i kraj tijela. Razmaci unutar šiljastih zagrada *nisu dozvoljeni*, tj. sljedeća direktiva je neispravna:

```
#include < iostream >
```

Unutar stringovnih literalala mogu se naći i neki specijalni, tzv. *kontrolni* znakovi. Jedan od najkorisnijih je znak “\n” koji predstavlja znak za “novi red” (kasnije ćemo upoznati i neke druge kontrolne znakove). Tako, ako napišemo

```
cout << "Zdravo\nnarode!";
```

ispis na ekranu će biti:



Naravno, mogli smo istu stvar pisati i kao 2 naredbe:

```
cout << "Zdravo\n";
cout << "narode!";
```

Ovakav način pisanja početnicima obično bude jasniji.

Biblioteka “`iostream`“ definira objekat nazvan “`endl`”, koji je u logičkom smislu sinonim za string “`\n`” (mada u izvedbenom smislu postoje značajne razlike) tako da smo istu naredbu mogli pisati i kao:

```
cout << "Zdravo" << endl << "narode!";
```

ili je raščlaniti na dvije naredbe kao:

```
cout << "Zdravo" << endl;
cout << "narode!";
```

Važno je napomenuti da stil pisanja programa (razmaci, novi redovi, uvlačenje redova, itd.) ni na koji način ne utiču na njegovo izvršavanje. Novi red ne označava kraj naredbe, nego je kraj naredbe isključivo tamo gdje se nalazi znak tačka-zarez. Tako bi sljedeći program proizveo isti efekat kao i program iz prvog primjera:

```
#include <iostream>
using namespace std;
int main(){cout<<"Zdravo narode!";return 0;}
```

Još gore, isti rezultat bi proizveo i ovakav program:

```
#include <iostream>
using
    namespace
std;int
main(
){ cout
    <<
    "Zdravo narode!"
;return
0
; }
```

Naravno, radi čitljivosti programa treba paziti na stil pisanja. Treba napomenuti da se stringovi *ne*

smiju prelamati u više redova. Na primjer, nije dozvoljeno pisati:

```
cout << "Zdravo  
narode!" ;
```

Šta će se dogoditi ukoliko napišemo nešto slično, zavisi od konkretnog kompjerala. Neki kompjajleri će prijaviti grešku, dok će neki prihvati ovakvu naredbu, i smatrati da string sadrži znak za novi red “\n” na mjestu preloma. Čak i uz pretpostavku da je ovo dozvoljeno u Vašoj verziji kompjajlera, ne trebate koristiti tu mogućnost, jer ona nije podržana standardom.

Pretprocesorske naredbe se također moraju pisati u *jednom redu*, tako da nije dozvoljeno pisati:

```
#include  
<iostream>
```

Pored toga, pretprocesorske naredbe se, kao što ste već vjerovatno primijetili, *ne završavaju tačka-zarezom.*

2. Promjenljive i ulazni tok

Programi koji su demonstrirani u prethodnom poglavlju su *potpuno beskorisni*, jer uvijek ispisuju jedan te isti tekst na ekran. Da bismo od programa imali ikakve koristi, oni moraju biti u stanju da prihvataju podatke od korisnika, da ih obrađuju, i da prezentiraju korisniku rezultate obrade. Za ostvarivanje ovih zadataka, programski jezik C++ poput mnogih drugih programskih jezika koristi posebne objekte koji se nazivaju *promjenljive* ili *variable*.

Posmatrano na fizičkom nivou, promjenljive možemo shvatiti kao određene dijelove radne memorije zadužene za čuvanje vrijednosti podataka koji se obrađuju u programu. Svaka promjenljiva se u memoriji čuva na određenoj *adresi* (adresa je najčešće neki broj koji određuje tačno mjesto u memoriji gdje se promjenljiva čuva). Kako bi rukovanje sa promjenljivim zasnovano na upotrebi adresa bilo veoma mučno, promjenljivim se dodjeljuju *imena*, koja služe za pristup njihovom sadržaju. Na taj način je programer oslobođen potrebe da razmišlja o adresama. Stoga, na logičkom nivou, promjenljive možemo posmatrati kao *imena kojima su predstavljeni podaci koji se obrađuju*. Sljedeća slika ilustrira ovaj koncept na primjeru tri promjenljive nazvane redom "x", "starost" i "slovo", koje sadrže redom vrijednosti "5.12", "21" i "A" (posljednji primjer jasno ukazuje da vrijednost promjenljive ne mora nužno da bude broj):

x	5.12
starost	21
slovo	A

U nekim programskim jezicima ime promjenljive u potpunosti određuje promjenljivu. U jeziku C++, svaka promjenljiva pored imena mora imati i svoj *tip*. Tip promjenljive određuje vrstu podataka koji se mogu čuvati u promjenljivoj, kao i operacije koje se mogu obavljati nad tim podacima. Na primjer, ukoliko je neka promjenljiva *cjelobrojnog tipa*, u njoj se mogu čuvati samo *cijeli brojevi*, i nad njenim sadržajem se mogu obavljati samo *operacije definirane za cijele brojeve*.

U jeziku C++ se svaka promjenljiva mora *najaviti (deklarirati)* prije nego što se prvi put upotrijebi u programu. Primjer deklaracije promjenljive izgleda ovako:

*Tip promjenljive
Ime promjenljive (identifikator)*

`int broj;`

Ovim deklariramo promjenljivu nazvanu "broj" koja može sadržavati samo cjelobrojne (engl. *integer*) vrijednosti, što je određeno *tipom promjenljive* (u ovom slučaju "int"). C++ spada u jezike tzv. *stroe tipizacije*, što znači da svaka promjenljiva mora imati svoj tip, koji striktno određuje skup mogućih vrijednosti (tzv. *domen promjenljive*) koje se mogu čuvati u toj promjenljivoj. Za početak ćemo koristiti samo promjenljive cjelobrojnog tipa, dok ćemo se kroz kasnija poglavlja upoznavati i sa drugim tipovima. Vidjećemo također da programer može definirati i svoje vlastite tipove podataka.

Imena promjenljivih i funkcija su specijalan slučaj tzv. *identifikatora*. Identifikatori smiju sadržavati samo alfanumeričke znakove (tj. slova i brojke), pri čemu prvi znak mora biti slovo. Pored toga, dozvoljena su samo slova engleskog alfabet-a, tako da naša slova ("č", "ć" itd.) nisu dozvoljena u identifikatorima. Tako su imena "a", "Sarajevo", "carsija", "a123", "U2" i "x2y7" legalna, dok

imena "čaršija", "2Pac" i "7b" nisu (dakle, "burek" je legalno, ali "ćevapi" nije legalno ime promjenljive). Dalje, razmaci u identifikatorima takođe nisu dozvoljeni. Prema tome, "moj broj" nije legalno ime promjenljive. Umjesto toga, možemo koristiti ime "mojbroj" ili, još bolje, ime "MojBroj" (neki udžbenici preporučuju korištenje imena poput "mojBroj", tako da je početno slovo uvijek malo slovo).

Veoma je važno naglasiti da je C++ tzv. "*case sensitive*" jezik, što znači da pravi striktnu razliku između malih i velikih slova. Tako su "mojbroj", "MOJBROJ" i "MojBroj" tri legalna, ali *posve različita identifikatori*. Dakle, mogu postojati tri potpuno različite promjenljive sa gore pomenuitim imenima (što je, naravno, loša praksa). Kao generalno pravilo, imena svih ugrađenih naredbi, funkcija i objekata jezika C++ sadrže isključivo mala slova, dok imena mnogih konstanti (koje ćemo uskoro upoznati) definiranih u standardnim bibliotekama jezika C++ često sadrže samo velika slova.

Pored slova i cifri, identifikatori mogu sadržavati i *donju crticu* "_" (engl. *underscore*), tako da je "moj_broj" također legalan identifikator. Ovu crticu treba razlikovati od *obične crtice* "-" (engl. *hyphen*), tako da ime "moj-broj" nije legalno. Principijelno, donja crtica je ravnopravna sa slovima, tako da identifikatori mogu čak i početi donjom crticom. Drugim riječima, imena poput "_xyz" pa čak i "_____ predstavljaju potpuno legalna imena. Suvišno je i reći da je upotreba ovakvih identifikatora veoma loša praksa.

Postoje izvjesne riječi koje, mada formalno ispunjavaju sve gore postavljene uvjete, *ne mogu biti identifikatori* zbog toga što je njihovo značenje precizno utvrđeno pravilima jezika C++ i ne mogu se koristiti ni za šta drugo. Takve riječi nazivaju se *rezervirane* ili *ključne riječi*. Standard jezika C++ predviđa sljedeće ključne riječi:

and	and_eq	asm	auto
bitand	bitor	bool	break
case	catch	char	class
compl	const	const_cast	continue
default	delete	do	double
dynamic_cast	else	enum	explicit
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	not	not_eq	operator
or	or_eq	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	xor	xor_eq

Radi lakšeg uočavanja rezervirane riječi se u programima obično prikazuju **podebljano**, što je učinjeno i u dosadašnjim primjerima, i što će biti ubuduće primjenjivano u svim primjerima koji slijede (mnogi programerski editori za jezik C++ automatski podebljavaju svaku uočenu rezerviranu riječ). Dakle, nije moguće imati promjenljivu koja se zove npr. "**friend**", s obzirom da je to rezervirana riječ.

Postoji mogućnost da neki kompjajleri koriste još neke rezervirane riječi, mada se to smatra ozbiljnim kršenjem standarda od strane kompjajlera. Obično se takvim "nestandardnim" rezerviranim riječima daju neobična imena (npr. imena koja počinju znakom "_") da se smanji mogućnost konfliktova sa programima

koji bi mogli slučajno nesvjesno upotrijebiti takvu riječ kao identifikator. Tako, na primjer, Borland C++ Builder uvodi rezervirane riječi poput “`_property`”, “`_closure`”, itd. Jedan od najsigurnijih načina da izbjegnete konflikte sa rezerviranim riječima je da koristite identifikatore bazirane na riječima iz bosanskog jezika, s obzirom da su sve rezervirane riječi izvedene iz engleskog jezika (tako da možete biti sigurni da “baklava” nije ključna riječ).

Upotrebu promjenljivih ćemo ilustrirati na primjeru programa koji će prosto ponoviti na ekranu broj koji unesemo sa tastature:

```
#include <iostream>
using namespace std;
int main() {
    int broj;
    cin >> broj;
    cout << broj;
    return 0;
}
```

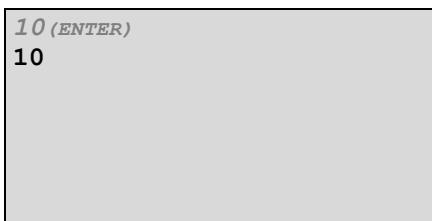
Ovdje trebamo posebno obratiti pažnju na naredbu koja glasi

```
cin >> broj;
```

“`cin`” (skraćenica od engl. *console in*) predstavlja objekat tzv. *ulaznog toka podataka* (engl. *input stream*), koji je povezan sa standardnim uređajem za unos. Standardni uređaj za unos je tipično tastatura, mada ne mora biti (C++ uopće ne podrazumijeva da računar na kojem se program izvršava mora posjedovati tastaturu, jer postoje i drugi načini za unos podataka). Znak “`>>`” predstavlja *operator izdvajanja* (engl. *extraction*) iz ulaznog toka, koji pojednostavljeno možemo čitati kao “šalji u”. Njegov smisao je suprotan u odnosu na smisao operatora umetanja “`<<`” koji se koristi uz objekat izlaznog toka “`cout`”. Razmotrimo preciznije šta se zapravo dešava. Uz pretpostavku da je standardni ulazni uređaj zaista tastatura, po nailasku na prethodnu naredbu koja zahtijeva izdvajanje promjenljive “`broj`” iz ulaznog toka “`cin`” program privremeno prekida rad i omogućava nam da unesemo neki niz znakova sa tastature, sve dok ne pritisnemo tipku ENTER. Uneseni niz znakova se pri tome čuva negdje u memoriji. Operator “`>>`” iz unesenog niza znakova izdvaja sve znake do prvog razmaka (ili do kraja unesenog niza), nakon čega izdvojene znakove interpretira kao cijeli broj koji smješta u promjenljivu “`broj`” čije ime se nalazi desno od operadora “`>>`” (interpretacija niza znakova kao cijelog broja uvjetovana je činjenicom da je “`broj`” promjenljiva cjelobrojnog tipa). Dakle, prethodna naredba može se interpretirati na sljedeći način:

Pošalji unos sa tastature interpretiran kao cijeli broj u promjenljivu nazvanu "broj".

Kada pokrenemo ovaj program, ne možemo odmah znati šta će se prikazati na ekranu, jer će ispis zavisiti od toga šta unesemo sa tastature. Zbog toga ćemo, u ovom i sličnim primjerima, prepostavljene vrijednosti unosa sa tastature prikazivati *sivim nakošenim slovima*. Tako, možemo prikazati sljedeću sliku:



Dakle, program je prihvatio unos sa tastature (u gornjem primjeru "10") i prosto ga ponovio. Primijetimo veliku razliku između na prvi pogled sličnih naredbi

```
cout << "broj";
```

i

```
cout << broj;
```

Prva naredba ispisuje *bukvalno* tekst "broj" na ekran. Druga naredba ispisuje *sadržaj* (tj. *vrijednost*) promjenljive koja se zove "broj". Pored toga, bitno je naglasiti da se sa desne strane operatora izdvajanja ">>" smije nalaziti samo *ime promjenljive* (koja mora biti prethodno deklarirana) i ništa drugo. Tako je, na primjer, sljedeća naredba potpuno besmislena:

```
cin >> "broj";
```

Strogo rečeno, kao desni operand operatora izdvajanja ">>" pored imena promjenljivih mogu se nalaziti i svi ostali objekti koji spadaju u kategoriju tzv. *l-vrijednosti* (engl. *l-values*), o kojima ćemo govoriti u sljedećem poglavlju. Međutim, za sada su jedine l-vrijednosti koje smo upoznali upravo imena promjenljivih, tako da nećemo mnogo pogriješiti ako kažemo da desni operand operatora ">>" mora biti ime promjenljive.

Interesantno je primijetiti da "cout" i "cin" *nisu rezervirane riječi*. U načelu, ukoliko ne koristimo biblioteku "iostream" nigdje u programu (što nije mnogo vjerovatno), sasvim je legalno koristiti riječi "cout" odnosno "cin" kao imena promjenljivih (kasnije ćemo vidjeti da "cout" i "cin" zapravo i jesu *promjenljive*, samo vrlo specifičnog tipa). U slučaju da koristimo biblioteku "iostream", "cout" i "cin" su već deklarirane unutar nje, tako da ne smijemo koristiti njihova imena kao identifikatore, s obzirom da C++ ne dozvoljava da se isti identifikator deklarira više puta unutar istog programa za više različitih namjena. Riječi poput "cin" i "cout" koje u jeziku C++ posjeduju unaprijed definirano značenje, ali čije je značenje u principu moguće potpuno promijeniti i koristiti ih za nešto posve drugo, nazivaju se *predefinirane riječi*.

Zavisno od kompjerala sa kojim radite, i u ovom primjeru će se vjerovatno desiti da će nakon završetka programa konzolni prozor zatvoriti, tako da nećete biti u stanju da vidite šta je program ispisao nakon unosa broja. Ubacivanje naredbe "cin.get();" koja je korištena u prethodnom primjeru kao "polurješenje" u ovom primjeru neće pomoći, jer se njeno korištenje zasniva na prepostavci da se ulazni tok neće koristiti (što ovdje očigledno nije tačno). Stoga nam je potrebno univerzalnije rješenje, koje će odložiti završetak programa sve dok recimo korisnik ne pritisne bilo koju tipku. Standard jezika C++ uopće ne predviđa nikakav način da se ovo riješi (vjerovatno zbog toga što ne predviđa da računar na kojem se program izvršava mora posjedovati tipke), tako da se moramo poslužiti *nestandardnim rješenjima*. Mada su ovakva rješenja ovisna od upotrijebljenog kompjerala i operativnog sistema, većina raspoloživih kompjajlera dolazi sa nestandardnom bibliotekom "conio.h" (od engl. *console input/output*) koja sadrži skupinu nestandardnih funkcija za rad sa *tastaturom* (a ne bilo kojim ulaznim tokom) i *ekranom* (a ne bilo kojim izlaznim tokom), dakle sa onim uređajima za koje C++ ne garantira da moraju postojati. U ovoj biblioteci nalazi se i veoma korisna funkcija "getch()" koja, između ostalog, privremeno zaustavlja izvođenje programa sve dok korisnik ne pritisne bilo koju tipku, a to je upravo ono što nam treba. Dakle, jedno *nestandardno rješenje* ovog problema može se demonstrirati kroz sljedeći program:

```
#include <iostream>
```

```

#include <conio.h>           // Nestandardno!
using namespace std;

int main() {
    int broj;
    cin >> broj;
    cout << broj;
    getch();                  // Nestandardno!
    return 0;
}

```

Napomenimo da su zgrade iza "getch" bitne, tako da naredba "getch;" (bez zagrada) neće izazvati željeni efekat, mada kompjajler neće prikazati nikakvu grešku (kasnije ćemo vidjeti da par zagrada predstavlja operaciju *poziva funkcije*, tako da u slučaju da izostavimo ove zgrade, funkcija "getch" bi bila samo *referencirana* odnosno *prozvana* ali ne i *pozvana*). U primjerima koji slijede nećemo koristiti biblioteku "conio.h" niti naredbu "getch();" da ne bismo listing programa nepotrebito opterećivali detaljima koji nisu bitni za razumijevanje samog programa, i koji su pri tome nestandardni. Međutim, ako želite da sami isprobate neke od kasnijih programa, vjerovatno ćete trebati ubaciti opisane modifikacije da bi efekat programa bio jednak očekivanom.

Već smo rekli da razmaci u programu u principu *nisu bitni*, osim unutar stringova i šiljastih zagrada. Međutim, to ne znači da smijemo izostaviti one razmake koji razdvajaju jednu riječ od druge. Tako je sljedeći program *neispravan*:

```

#include <iostream>
using namespacesd;          // Greška: riječ "namespacesd" nema smisla
intmain() {                 // Greška: riječ "intmain" nema smisla
    intbroj;                // Greška: riječ "intbroj" nema smisla
    cin>>broj;              // Nije greška: ">>" razdvaja "cin" i "broj"
    cout<<broj;              // Nije greška: "<<" razdvaja "cout" i "broj"
}

```

Do sada razmotreni primjeri programa iz ovog poglavlja su "ružni" u smislu da korisnik programa nakon njihovog pokretanja uopće ne zna šta se od njega očekuje. Svaki dobro napisani program trebao bi da bude takav da korisnik programa u svakom trenutku zna šta se od njega očekuje, i šta predstavlja ispis koji se eventualno javlja kao rezultat rada programa (programer koji je pisao program to sigurno zna, ali programer i korisnik programa često nisu ista osoba). Drugim riječima, program bi trebao biti "ljubazan prema korisniku" (engl. *user friendly*). Stoga se programi mogu učiniti "ljepšim" ako svaku naredbu ulaza proratimo odgovarajućom naredbom izlaza koja će na ekranu ispisati upute šta se od nas očekuje da unesemo. Sljedeći primjer ilustrira takav "ljubazniji" program:

```

#include <iostream>
using namespace std;

int main() {
    int broj;
    cout << "Unesi neki broj: "
    cin >> broj;
    cout << "Unijeli ste broj " << broj << endl;
    return 0;
}

```

Jedan mogući scenario izvršavanja ovog programa je sljedeći:

```
Unesi neki broj: 7 (ENTER)
Unijeli ste broj 7
```

Naravno, ovakav program je još uvijek posve beskoristan, ali barem posjeduje izvjesnu dozu komunikacije sa korisnikom programa.

Kao što je već rečeno, operator “`>>`” izdvaja znakove iz ulaznog toka samo *do prvog razmaka*, što ilustrira sljedeći primjer izvršavanja “neljubazne” verzije ovog programa:

```
10 20 30 (ENTER)
10
```

Također, ukoliko se prilikom izdvajanja brojčanih podataka iz ulaznog toka najde na znak koji *nije sastavni dio broja* (npr. na slovo), izdvajanje prestaje na tom znaku, kao u sljedećem scenariju:

```
10ab20 (ENTER)
10
```

U oba primjera, samo broj “10” je izdvojen iz ulaznog toka. Međutim, preostali unijeti podaci (niz znakova “20 30” odnosno “ab20”) i dalje su pohranjeni u memoriji i predstavljaju dio ulaznog toka. Stoga će eventualna sljedeća upotreba operatora izdvajanja nastaviti izdvajanje iz niza znakova zapamćenog u memoriji (kažemo da se nastavlja izdvajanje iz ulaznog toka). Tek kada se *ulazni tok isprazni*, odnosno kada se *istroše svi znakovi* pohranjeni u memoriji biće zatražen novi unos sa ulaznog uređaja. Neka, na primjer, imamo sljedeći program:

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

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

```

    cin >> c;
    cout << a << endl << b << endl << c << endl;
    return 0;
}

```

Primijetimo prvo da smo u ovom programu deklarirali tri cjelobrojne promjenljive a, b i c odjednom pomoću deklaracije

```
int a, b, c;
```

Potpuno isti efekat postigli bismo sa tri odvojene deklaracije:

```

int a;
int b;
int c;

```

Prikažimo dva moguća “scenarija” pri izvršavanju ovog programa, koji ilustriraju način na koji djeluje operator izdvajanja nad ulaznim tokom:

```
10 20 30 (ENTER)
10
20
30
```

```
10 20 (ENTER)
30 (ENTER)
10
20
30
```

Operator “`>>`” se također može *nadovezivati* odnosno *ulančavati* poput operatora “`<<`”, tako da smo isti program mogli napisati i kraće, na sljedeći način:

```

#include <iostream>
using namespace std;
int main() {
    int a, b, c;
    cin >> a >> b >> c;
    cout << a << endl << b << endl << c << endl;
    return 0;
}

```

Međutim, ovdje treba obratiti pažnju na jednu veoma čestu početničku grešku. Ukoliko umjesto naredbe

```
cin >> a >> b >> c;
```

slučajno napišemo naredbu poput

```
cin >> a, b, c;
```

kompajler *neće prijaviti nikakvu grešku*, s obzirom da je gornja konstrukcija *sintaksno ispravna* u jeziku C++. Međutim, ova konstrukcija ne radi ono što bi korisnik mogao očekivati. Posebno, ona će dovesti do izdvajanja samo promjenljive “a” iz ulaznog toka, dok će promjenljive “b” i “c” biti prosto ignorirane. Zašto je tako, shvatićemo kasnije kada upoznamo značenje tzv. *zarez-operatora* (engl. *comma operator*). U ovom trenutku je samo potrebno zapamtiti da ova konstrukcija *ne vrši* izdvajanje promjenljivih “a”, “b” i “c” iz ulaznog toka. Ovakvih naizgled ispravnih konstrukcija koje ne rade ono što bi na prvi pogled trebalo da rade treba se naročito čuvati, jer nas na njih kompjajler ne može upozoriti (one su u načelu sintaksno ispravne). U žargonu se takve konstrukcije obično nazivaju *zamke* (engl. *pitfalls*).

Ako uneseni niz znakova nije sadržavao niti jednu cifru, a očekivana je cifra (npr. zbog toga što zahtijevamo unos cjelobrojne promjenljive), ulazni tok će dospjeti u tzv. *neispravno stanje*, i svaki sljedeći pokušaj izdvajanja iz ulaznog toka biće ignoriran, sve dok tok ne vratimo ponovo u *ispravno stanje* pomoću naredbe “`cin.clear()`”. O ovome ćemo detaljno govoriti kasnije, kada naučimo kako možemo utvrditi da li je ulazni tok u *neispravnom stanju*, i na taj način preduzeti određenu akciju (npr. obavijestiti korisnika da je unio pogrešne podatke).

Veoma je važno napomenuti da deklaracija neke promjenljive samo obavještava kompjajler da imenovana promjenljiva *postoji*, ali ne i kolika joj je vrijednost. Na primjer, deklaracijom “`int a;`” kompjajler će biti obaviješten o postojanju promjenljive “a”, čime će *zauzeti mjesto u memoriji gdje će se čuvati njena vrijednost*. Međutim, njena *početna vrijednost* će biti *posve slučajna*, preciznije njena početna vrijednost će zavisiti od *onoga što se u tom trenutku od ranije nalazilo u memoriji na dodijeljenom mjestu*. Ta vrijednost je često nula, ali ne uvijek. Stoga će sljedeći program, kada ga pokrenemo, vjerovatno ispisati neku potpuno nepredvidljivu i besmislenu vrijednost:

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

int main() {
    int a;
    cout << a << endl;
    return 0;
}
```

Navedeni program predstavlja tipični primjer programa koji je *sintaksno ispravan*, tj. napisan je potpuno u skladu sa “pravopisnim” i “gramatičkim” pravilima jezika C++, ali sadrži suštinsku grešku *logičke* odnosno *semanticke* prirode. Ovakvih grešaka se treba dobro čuvati, jer nas kompjajler može upozoriti samo na sintaksne greške (poneki kompjajleri mogu prepoznati ovakve situacije i eventualno uputiti upozorenje, ali ne i grešku). Dakle, prilikom prevodenja prethodnog programa, kompjajler neće javiti postojanje ikakve greške.

Svaka promjenljiva koja je deklarirana imaće besmislenu vrijednost sve dok joj se eksplicitno ne dodijeli neka konkretna vrijednost. Jedan način dodjele vrijednosti smo već upoznali: izdvajanje vrijednosti iz ulaznog toka. U tom slučaju, promjenljiva dobija vrijednost na osnovu podataka u ulaznom toku (tipično podataka unijetih sa tastature). U narednom poglavlju ćemo upoznati i drugi način dodjele vrijednosti promjenljivim, korištenjem tzv. *operatora dodjele* “`=`”. Napomenimo da je korištenje

promjenljivih kojima nije dodijeljena vrijednost jedna od najčešćih programerskih grešaka (tipično u većim programima), koja obično dovodi do programa koji, zavisno od slučaja, nekad rade ispravno, a nekada ne rade (jer rezultat njihovog rada praktično zavisi od slučajne početne vrijednosti promjenljive).

Postoji mogućnost da se promjenljivoj prilikom deklaracije zada i njena početna vrijednost, tako da će njen sadržaj odmah biti dobro definiran. To se postiže tako što se iza imena promjenljive njena početna vrijednost navede unutar zagrade. Tako će se sljedeći program ponašati posve predvidljivo (ispisivaće uvijek vrijednost "5"), s obzirom da je definirano da je početna vrijednost promjenljive "a" jednaka "5". Kažemo da je promjenljiva "a" *inicijalizirana* na vrijednost "5".

```
#include <iostream>
using namespace std;
int main() {
    int a(5);
    cout << a << endl;
    return 0;
}
```

Napomenimo da je mogućnost zadavanja početne vrijednosti promjenljive u zagradama uvedena u novijim standardima jezika C++, tako da neki vrlo stari prevodioci za C++ (koje ionako više ne bi trebalo koristiti) ne podržavaju ovu sintaksu. U takvim slučajevima, inicijalizacija promjenljivih se može izvršiti na nešto drugačiji način (naslijeden iz jezika C), koji će biti opisan u sljedećem poglavlju.

Za promjenljive čija se početna vrijednost ne navodi prilikom deklaracije kažemo da su *neinicijalizirane*. Zbog problema koji mogu nastati zbog upotrebe neinicijaliziranih promjenljivih, preporučuje se da se sve promjenljive koje će se koristiti u programu obavezno inicijaliziraju, osim u slučaju kada neposredno iza same deklaracije slijedi izdvajanje te promjenljive iz ulaznog toka, nakon čega će ona definitivno dobiti sasvim određenu vrijednost.

3. Dodjeljivanje i aritmetički izrazi

U prethodnom poglavlju smo vidjeli da se promjenljivoj može dodijeliti vrijednost izdvajanjem sa ulaznog toka. Pored toga, vidjeli smo da početnu vrijednost promjenljive možemo zadati i putem inicijalizacije, prilikom njene deklaracije. Sada ćemo vidjeti da se promjenljivoj može dodijeliti vrijednost i primjenom *naredbe pridruživanja* odnosno *dodjele* (engl. *assignment statement*). Primjer naredbe pridruživanja u jeziku C++ izgleda poput:

```
broj = 5;
```

Ovdje je prepostavljeno da je promjenljiva `broj` prethodno deklarirana (u suprotnom će prevodilac javiti grešku). Znak “`=`” predstavlja *operator dodjele* (engl. *assignment operator*). Čita se kao “*postaje*”, tako da gornju naredbu treba čitati kao “*Broj postaje pet*”. Ovdje je bitno shvatiti da operator dodjele ne predstavlja *jednakost u matematskom smislu*, nego ima imperativno dejstvo, kojim se objektu sa lijeve strane operadora dodjele *dodjeljuje* vrijednost sa desne strane. Stoga, prethodnu naredbu nismo mogli napisati kao

```
5 = broj;
```

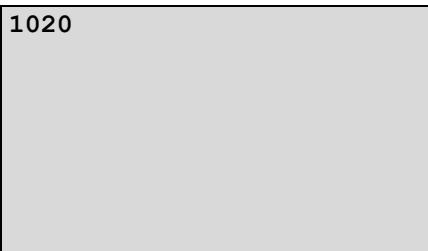
s obzirom da se objektu “5” ne može dodjeliti druga vrijednost od one vrijednosti koju on sam po sebi veći ima (“5”). Slijedi da se sa lijeve strane operadora dodjele mogu nalaziti samo objekti čija se vrijednost *može mijenjati*. To su tipično objekti iza kojih u načelu stoji određena *memorijska lokacija*, koja može prihvati vrijednost koja se dodjeljuje objektu. Takvi objekti, koji mogu stajati sa lijeve strane operadora dodjele, nazivaju se *l-vrijednosti* (engl. *lvalues*). Dakle, promjenljive jesu *l-vrijednosti*, a brojevi *nisu*. Kasnije ćemo upoznati i druge l-vrijednosti osim promjenljivih.

Svakoj promjenljivoj se, kao što joj i ime govori, može *mijenjati* vrijednost proizvoljan broj puta tokom izvršavanja programa. Slijedeći program to lijepo ilustrira:

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

int main() {
    int broj;
    broj = 10;
    cout << broj;
    broj = 20;
    cout << broj;
}
```

Rezultat izvršavanja ovog programa je sljedeći:



Kao što je i očekivano, ovaj program je ispisao brojeve “10” i “20”. Međutim, interesantno je primijetiti da su ovi brojevi slijepjeni, tako da prikaz izgleda poput broja “1020”. Zašto je tako? Razlog je veoma jednostavan: zbog toga što nigdje nije rečeno da tako ne treba da bude! Ono što ne piše u programu, neće se ni izvršiti. Računar ništa ne podrazumijeva. Već smo rekli da se prilikom ispisa na izlazni tok, svaki sljedeći ispis prosti nastavlja od mjesta gdje je prethodni ispis završio. Upravo se to i desilo u ovom programu. Da smo htjeli da broj “20” bude odvojen jednim razmakom od broja “10”, taj razmak bismo morali ispisati *eksplicitno*, tj. trebali bismo imati program poput sljedećeg:

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

int main() {
    int broj;
    broj = 10;
    cout << broj << " ";
    broj = 20;
    cout << broj << endl;
}
```

Slično, ukoliko smo željeli da broj “20” bude isписан u novom redu, mogli smo napisati program poput sljedećeg:

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

int main() {
    int broj;
    broj = 10;
    cout << broj << endl;
    broj = 20;
    cout << broj << endl;
    return 0;
}
```

U jeziku C++ pridruživanje se može obaviti *istovremeno sa deklaracijom*, kao u sljedećem primjeru:

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

int main() {
    int broj = 10;
    cout << broj << endl;
    broj = 20;
    cout << broj << endl;
    return 0;
}
```

Istu stvar smo mogli napisati i na sljedeći način, s obzirom da smo vidjeli da se početna vrijednost promjenljive može prilikom deklaracije navesti unutar zagrada iza njenog imena:

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

int main() {
    int broj(10);
```

```

    cout << broj << endl;
    broj = 20;
    cout << broj << endl;
return 0;
}

```

Ova šarolikost može u prvi mah zbuniti početnika. Da rezimiramo, postoje tri načina na koji možemo definirati cjelobrojnu promjenljivu “`broj`” čija je vrijednost “10”:

Način 1

Način 2

Način 3

Prva dva načina imaju potpuno identično dejstvo: stvaraju promjenljivu `broj` koju odmah prilikom stvaranja inicijaliziraju na vrijednost 10. Zaista, prva dva načina su pretežno sinonimi (osim u nekim relativno rijetkim izuzecima, o kojima će biti govora kasnije), pri čemu je drugi način naslijeden iz jezika C. Prvi način je uveden ekskluzivno u jeziku C++, s obzirom da je C++ uveo složenje tipove podataka koji se ne mogu inicijalizirati jednom vrijednošću, već traže više vrijednosti za inicijalizaciju, tako da je sintaksa koja koristi znak “`=`” neprikladna (npr. objekti koji predstavljaju kompleksne brojeve inicijaliziraju se sa dvije vrijednosti, koje predstavljaju realni i imaginarni dio kompleksnog broja). Također, postoje izvjesni tipovi objekata u jeziku C++, koje ćemo kasnije upoznati, kod kojih bi upotreba znaka “`=`” za inicijalizaciju bila zbumujuća. Stoga, C++ preporučuje da se za inicijalizaciju objekata uvijek koristi sintaksa sa zagradama (način 1), mada će za inicijalizaciju većine objekata koji se mogu inicijalizirati sa jednom vrijednošću (osim određenih tipova objekata kod kojih je inicijalizacija na ovaj način eksplicitno zabranjena, o čemu će kasnije biti riječi) biti prihvaćena i sintaksa kod koje se koristi znak “`=`” (način 2).

S druge strane, način 3 se donekle razlikuje od prva dva načina. Pri korištenju ovog načina, prvo se stvara *neinicijalizirana promjenljiva* “`broj`”, kojoj se *kasnije dodjelye* vrijednost “10”. Dakle, u ovom slučaju, faza dodjele vrijednosti je razdvojena od faze dodjele vrijednosti. Mada početnicima ukazivanje na ovu razliku može djelovati kao nepotrebno sitničarenje, potrebno je već na samom početku shvatiti da jezik C++ strogo razlikuje *proces inicijalizacije* (engl. *initialization*), kod kojeg se vrijednost nekog objekta (npr. promjenljive) postavlja *prilikom njenog kreiranja*, i *proces dodjele* (engl. *assignment*), kod kojeg se postavlja vrijednost objekta *koji već postoji* (i koji pri tome od ranije ima neku vrijednost, makar i nedefiniranu), pri čemu novopostavljena vrijednost *zamjenjuje prethodno postojeću vrijednost*.

U ovako jednostavnom primjeru, razlika između inicijalizacije i dodjele je zaista minorna, tako da mnogi na nju neće obraćati pažnju. Međutim, kod rada sa složenijim objektima razlike postaju izražajnije. U slučaju složenijih objekata ćemo vidjeti da način 3 može biti znatno neefikasniji od prva dva načina. Pored toga, postoje i takvi objekti za koje je *zabranjeno* da budu neinicijalizirani, pa se način 3 ne može ni primijeniti. Sve ovo će postati jasnije tek kada se upoznamo sa objektima mnogo složenije strukture nego što su cjelobrojne promjenljive. Uglavnom, sintaksa sa zagradama, iskorištena u načinu 1, uvedena je baš sa ciljem da se jasnije istakne razlika između inicijalizacije i dodjele. Sintaksa sa znakom “`=`”, korištena u načinu 2, zadržana je uglavnom radi kompatibilnosti sa jezikom C. Noviji standardi jezika C++ ne preporučuju njeno korištenje, s obzirom da se iz nje ne vidi jasno da se ne radi o *dodjeli* nego o *inicijalizaciji*, a složeniji koncepti jezika C++ zahtijevaju od programera da jasno vodi računa o razlici između ova dva pojma, kao što ćemo vidjeti kasnije kada upoznamo rad sa složenijim korisnički definiranim tipovima podataka.

Jezik C++ spada u *strog tipizirane jezike* (engl. *strongly typed languages*), što znači da svaka promjenljiva obavezno mora imati svoj tačno određeni *tip*. Tako, deklaracija

```
int broj;
```

označava da je tip promjenljive “broj” cijeli broj (engl. *integer*). Tip promjenljive služi da odredi kako vrstu vrijednosti koja može biti dodijeljena promjenljivoj, kao i vrstu operacija koje se na nju mogu primjenjivati. Tako, na primjer, za promjenljivu “broj” iz gornjeg primjera, to znači da se njoj mogu dodjeljivati samo cijelobrojne vrijednosti, i nad njom se mogu primjenjivati samo operacije definirane za cijele brojeve (a ne, na primjer, operacije definirane za skupove ili nizove znakova).

Sa desne strane operatora dodjele ne mora se uvijek nalaziti samo broj, kao u dosadašnjim primjerima. Sasvim je moguće sa desne strane operatora dodjele upotrijebiti *ime neke druge promjenljive* (koja mora imati prethodno definiranu vrijednost, da bi dodjela imala smisla), pa čak i proizvoljan *izraz* (engl. *expression*). Prilično je teško *formalno* definirati šta je to izraz. Stoga, ovdje nećemo ni pokušavati formalno definirati izraz, već ćemo pojam izraza uesti opisno, na neformalan način, pri čemu će smisao ovakvog opisa biti jasan po intuiciji. Neformalno rečeno, izraz je formula u kojoj se mogu nalaziti razni objekti (npr. promjenljive i brojevi), koji su eventualno medusobno povezani izvjesnim operatorima. Pri tome, svaka skupina objekata i operatora ne tvori nužno izraz, već za formiranje izraza postoje izvjesna sintaksna pravila (koja su, u većini slučajeva, intuitivno jasna). Čitav izraz uvijek ima neku *vrijednost*, koja nastaje kao rezultat *izračunavanja* (engl. *evaluation*) izraza. Na primjer,

2 + 3 * 5

predstavlja jedan izraz, čija je vrijednost “17” (ovdje znak “*” označava operaciju množenja). Za sada ćemo se upoznati prvo sa cijelobrojnim izrazima. Oni se mogu sastojati od sljedećih elemenata:

a) Cijelih brojeva, npr.

5 999 +999 -42

Cijeli brojevi se sastoje od cifara “0” – “9” kojima eventualno prethodi predznak “+” ili “-” (pri čemu se predznak “+” podrazumijeva u slučaju izostavljanja). Ovdje treba upozoriti na jednu čudnu konvenciju koja je (nažalost) naslijedena iz jezika C. Naime, dok se u matematici vodeće nule u broju ignoriraju (tako da je “00253” isti broj kao i “253”), u jezicima C i C++ vodeća nula označava da je broj napisan u *oktalnom brojnom sistemu* (tj. sistemu sa bazom 8). Tako je u jezicima C i C++ broj “00253” zapravo broj “171” ($2 \cdot 8^2 + 5 \cdot 8^1 + 3$). O ovome treba voditi računa, jer može dovesti do grešaka koje se teško otkrivaju. Također, jezici C i C++ omogućavaju i zadavanje cijelih brojeva u *heksadekadnom brojnom sistemu* (što može nekada biti korisno) dodavanjem prefiksa “0x” ispred broja. Pri tome se slova “A” – “F” (nebitno da li su mala ili velika) koriste kao cifre “10” – “15”. Tako, broj “0x1f2” zapravo predstavlja cijeli broj “498” u dekadnom sistemu ($1 \cdot 16^2 + 15 \cdot 16^1 + 2 \cdot 16^0$)

b) Cijelobrojnih aritmetičkih operatora, među kojima su najvažniji:

+	<i>sabiranje</i>
-	<i>oduzimanje</i>
*	<i>množenje</i>
/	<i>cijelobrojno dijeljenje</i>
%	<i>ostatak pri cijelobrojnom dijeljenju</i>

c) Cijelobrojnih funkcija, o kojima će kasnije biti riječi;

d) Zagrada, pri čemu se bez obzira na dubinu gniježdenja, koriste isključivo male zagrade.

Slijedi nekoliko primjera ispravnih cijelobrojnih izraza:

Izraz:	Vrijednost:
2 + 2	4
33 * -6	-198
48 / 5	9
7 % 3	1
(5 + 2) * 7 + 4	53
120 / (4 + 12 * (3 + 2))	1

Primijetimo da operator “/” označava *cijelobrojno dijeljenje*, tako da rezultat izraza “7/2” nije “3.5” kao u matematici, nego “3”. Operator “%” označava ostatak pri cijelobrojnem dijeljenju (engl. *remainder*), tako da je $7 \% 2 = 1$. Kod operatora “/” i “%” drugi operand ne smije biti nula, inače se javlja greška koja tipično rezultira prekidom rada programa. Također, rezultat izvođenja operatora “%” nije precizno definiran standardom jezika C++ u slučaju da je drugi operand negativan. Rezultat u tom slučaju može zavisiti od konkretne izvedbe kompjlera. Najčešće je operator “%” definiran tako da vrijedi formula

$$x \% y = x - y * (x / y)$$

bez obzira na znak operanada x i y .

Prilikom izračunavanja izraza, poštuje se prioritet operacija na način kako je uobičajeno u mathematici. Tako, operatori “*” i “/” imaju prioritet u odnosu na operatore “+” i “-“. Operator “%” ima isti prioritet kao i operatori “*” i “/“. Naravno, redoslijed izvođenja operacija može se promijeniti upotrebom zagrada na način uobičajen u mathematici (samo uz korištenje isključivo malih zagrada). U slučaju operatora istog prioriteta, redoslijed izvođenja operacija je slijeva nadesno, tako da se izraz “7 / 3 * 2” interpretira kao “(7 / 3) * 2” a ne kao “7 / (3 * 2)”.

Već je rečeno da se sa desne strane operatora dodjele mogu koristiti i izrazi, tako da su sljedeće naredbe dodjele sasvim legalne:

```
broj = 2 + 2;
broj = 7 % 3;
broj = 120 / (4 + 12 * (3 + 2));
```

Također, cijelobrojni izrazi mogu sadržavati i *cijelobrojne promjenjive*, kao što je prikazano u sljedećem demonstracionom programu:

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

int main() {
    int a, b, c;
    a = 5;
    b = 3;
    c = a * 2;
    b = (c + 3) / a;
    a = a + 7;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    return 0;
```

```
}
```

U ovom programu treba obratiti posebnu pažnju na dodjelu “ $a = a + 5$ ” koja na prvi pogled izgleda čudno. Međutim, treba imati u vidu da operator “ $=$ ” ne predstavlja *jednakost*, nego *dodjelu*, pri čemu se lijevoj strani *dodjeljuje* izračunata vrijednost izraza sa desne strane. Tako se, u ovoj naredbi, trenutna vrijednost promjenljive “ a ” sabira sa 5, nakon čega se izračunata vrijednost *smješta* ponovo u promjenljivu “ a ”. Tako, navedena dodjela ima značenje “*nova vrijednost promjenljive a postaje stara vrijednost promjenljive a plus 7*”. Drugim riječima, navedena dodjela *uvećava vrijednost promjenljive a za 7*. Tako će rezultat izvršavanje ovog programa biti:

```
12
2
10
```

Dakle, veoma je bitno da shvatimo da znak “ $=$ ” označava *dodjelu*, a ne *dvosmjernu jednakost*, odnosno *jednakost u matematičkom smislu*.

Već smo rekli da se promjenljive mogu *inicijalizirati prilikom deklaracije*, kao i da se operator “ $<<$ ” može nadovezivati, što omogućava da prethodni program napišemo *kompaktnije*:

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

int main() {
    int a(5), b(3), c(a * 2);
    b = (c + 3) / a;
    a = a + 7;
    cout << a << endl << b << c << endl;
    return 0;
}
```

U ovom primjeru je interesantna inicijalizacija promjenljive “ c ”, koja se ne vrši *brojem* nego *izrazom*, što je sasvim legalno, pod uvjetom da se u izrazu nalaze samo promjenljive koje su prethodno *inicijalizirane*, ili im je na neki drugi način *dodijeljena vrijednost*. U suprotnom će promjenljiva biti inicijalizirana izrazom čija vrijednost *nije definirana*, što je suštinski isto kao da nije ni inicijalizirana!

Deklaracije promjenljivih se *ne moraju* nužno nalaziti na početku funkcije, ali svaka promjenljiva mora biti deklarirana prije nego što se *upotrijebi u programu*. Tako je npr. sljedeći program ispravan:

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

int main() {
    int a;
    a = 5;
    int b;
    b = 3;
    int c;
    c = a * 2;
```

```

    b = (c + 3) / a;
    a = a + 7;
    cout << a << endl << b << endl << c << endl;
    return 0;
}

```

Međutim, sljedeći program je neispravan, jer se promjenljive koriste *prije njihove deklaracije*:

```

#include <iostream>
using namespace std;

int main() {
    a = 5;
    b = 3;
    c = a * 2;
    int a, b, c;
    b = (c + 3) / a;
    a = a + 7;
    cout << a << endl << b << endl << c << endl;
    return 0;
}

```

U jeziku C sve promjenljive su se morale deklarirati na samom početku, prije prve tzv. *izvršne naredbe* unutar funkcije (tj. naredbe koja ima neko dejstvo). U jeziku C++, kao što smo upravo rekli, ovo više nije slučaj. Naprotiv, u jeziku C++ se strogo preporučuje da se niti jedna promjenljiva ne deklarira *prije mesta njenog prvog korištenja*.

Svakoj promjenljivoj se može promijeniti vrijednost proizvoljan broj puta u programu, ali se ni jedna promjenljiva ne smije deklarirati više od jedanput, tako da je sljedeći program također neispravan:

```

#include <iostream>
using namespace std;

int main() {
    int a;
    a = 3;
    cout << a;
    int a;           // Dvostruka deklaracija - greška!
    a = 2;
    cout << a;
    return 0;
}

```

Isto vrijedi za sljedeći program, u kojem su obje deklaracije (sa inicijalizacijom) napisane tako da *liče na dodjelu*:

```

#include <iostream>
using namespace std;

int main() {
    int a = 3;
    cout << a;
    int a = 2;       // Dvostruka deklaracija - greška!
    cout << a;
    return 0;
}

```

}

Činjenica da operator “/” u cjelobrojnim izrazima predstavlja *cjelobrojno dijeljenje* može često dovesti do zabuna. Na primjer, pogledajmo izraze “ $a * (b / 30)$ ” i “ $(a * b) / 30$ ” gdje su “ a ” i “ b ” neke cjelobrojne promjenljive. Matematički posmatrano, ova dva izraza djeluju ekvivalentni. Međutim, oni to nisu, upravo zbog činjenice da “/” predstavlja cjelobrojno dijeljenje. Na primjer, neka promjenljive “ a ” i “ b ” imaju redom vrijednosti “90” i “75”. Tada je vrijednost prvog izraza “180”, jer je $75 / 30 = 2$ i $90 * 2 = 180$. Međutim, vrijednost drugog izraza je “225”, jer je $90 * 75 = 6750$ i $6750 / 30 = 225$. Iz istog razloga, nisu ekvivalentni ni izrazi poput “ $a / (b / 30)$ ” i “ $(a / b) * 30$ ” (za iste vrijednosti promjenljivih “ a ” i “ b ” kao u prethodnom primjeru, njihove vrijednosti su redom “45” i “30”). Stoga, koji ćemo oblik izraza koristiti, ovisi od toga šta zaista želimo da postignemo. Obično su izrazi sa cjelobrojnim dijeljenjem najsversihodniji u slučaju kada je cjelobrojno dijeljenje *posljednja* operacija koja se obavlja, tako da u većini primjena izraz poput “ $(a * b) / 30$ ” ima više smisla od izraza “ $a * (b / 30)$ ”. Specijalno se treba čuvati izraza u kojima se javlja višestruko cjelobrojno dijeljenje. Na primjer, izraz “ $a / (b / 30)$ ” za slučaj kada promjenljive “ a ” i “ b ” imaju respektivno vrijednosti “20” i “10” nije uopće definiran, jer je $10 / 30 = 0$ (tako da dobijamo dijeljenje nulom), za razliku od prividno ekvivalentnog izraza “ $(a / b) * 30$ ”, čija je vrijednost “60”!

Već je rečeno da su veoma opasne greške koje mogu nastati uslijed upotrebe neinicijaliziranih promjenljivih. Ovakve greške su posebno opasne, jer nam kompjaler ne može ukazati na njih, tj. sa aspekta kompjlera program djeluje ispravan (on to u suštini i jeste, ali samo sintaksno). Svaka promjenljiva koja se upotrijebi u bilo kojem izrazu *mora imati jasno definiranu vrijednost* da bi sam izraz bio dobro definiran. Stoga će sljedeći program ispisati neku nedefiniranu vrijednost (s obzirom da vrijednost izraza “ $a + 7$ ” ne može biti dobro definirana, s obzirom da promjenljivoj “ a ” nije pridružena nikakva konkretna vrijednost:

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

int main() {
    int a, b;
    b = a + 7;
    cout << b << endl;
    return 0;
}
```

Isto tako nipošto ne smijemo misliti da će sljedeći program ispisati vrijednost “3” po analogiji sa matematskim jednačinama (rezultat ovog programa je zapravo nepredvidljiv jer je promjenljiva “ y ” neinicijalizirana i nigdje joj se ne dodjeljuje vrijednost, a nakon druge dodjele promjenljiva “ x ” će također postati nedefinirana, jer joj se dodjeljuje nedefinirana vrijednost izraza “ $y + 2$ ”):

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

int main() {
    int x = 5, y;
    x = y + 2;
    cout << y << endl;
    return 0;
}
```

Cjelobrojnu aritmetiku ćemo dodatno ilustrirati na primjeru još jednog jednostavnog programa:

```

#include <iostream>
using namespace std;

int main() {
    int broj_1, broj_2;
    cout << "Unesi jedan broj: ";
    cin >> broj_1;
    cout << "Unesi drugi broj: ";
    cin >> broj_2;
    cout << "Zbir ovih brojeva je " << broj_1 + broj_2 << endl;
    return 0;
}

```

Mogući scenario izvršavanja ovog programa je sljedeći (pri čemu nećemo više pisati (ENTER) iza unosa da naznačimo da se unos mora završiti pritiskom na ENTER):

```

Unesi jedan broj: 7
Unesi drugi broj: 5
Zbir ovih brojeva je 12

```

Ovdje treba primijetiti da se aritmetički izrazi mogu koristiti i kao drugi operand operatora umetanja na izlazni tok “`<<`” (zapravo, bilo koji operator može primiti kao operand bilo koji izraz čiji je tip saglasan sa očekivanim tipom odgovarajućeg operanda). Možda je malo neobična činjenica (po kojoj se C++ razlikuje od mnogih drugih jezika) da je konstrukcija poput

```
cout << broj_1 + broj_2
```

također *izraz*. Njegova vrijednost nije ništa drugo već *sam objekat* “`cout`” (nešto preciznije, *referenca* na ovaj objekat, ali u ovom trenutku ne možemo objasniti šta to tačno znači). Upravo zahvaljujući ovoj činjenici je i moguće *nadovezivanje* na izlazni tok. Razmotrimo, na primjer, izraz

```
cout << broj_1 << " " << broj_2
```

Ovaj izraz se zapravo interpretira kao:

```
((cout << broj_1) << " ") << broj_2
```

Prvo se obavi “računanje” izraza u unutrašnjim zagradama, koje kao posljedicu ima ispis sadržaja promjenljive “`broj_1`”. Rezultat ovog izraza je sam objekat “`cout`”, tako da se sada prethodni izraz svodi na izraz

```
(cout << " ") << broj_2
```

Ponovo se prvo izvodi izraz u zagradi, koji ispisuje razmak, nakon čega daje objekat “`cout`” kao rezultat, tako da se izraz svodi na izraz

```
cout << broj_2
```

koji na kraju ispisuje sadržaj promjenljive “`broj_2`”. Krajnji rezultat cijelog izraza je također objekat

“cout”, ali se ovaj rezultat dalje ne koristi nizašta (tj. ignorira se). Također, vrijedi napomenuti da je rezultat izraza poput “cin >> broj” sam objekat ulaznog toka “cin”.

Operator “<<” ima niži prioritet od aritmetičkih operatora poput “+” itd. tako da se izraz

```
cout << broj_1 + broj_2
```

posve logično interpretira kao

```
cout << (broj_1 + broj_2)
```

Dakle, prvo se računa vrijednost izraza “broj_1 + broj_2” nakon čega se izračunata vrijednost šalje na izlazni tok (tipično ekran).

Jezik C++ posjeduje veliki broj operatora koji ne postoje u drugim programskim jezicima. Upoznajmo prvo operatore “+=”, “-=”, “*=”, “/=” i “%=” . Naime, kako u programiranju često postoji potreba da se neka promjenljiva poveća, smanji, pomnoži ili podijeli nečim, i da se rezultat ponovo stavi u tu istu promjenljivu, autori jezika C++ su uveli i skraćeni način da se ovakva dodjeljivanja obave:

$x += y;$	<i>znači isto što i</i>	$x = x + y;$
$x -= y;$	<i>znači isto što i</i>	$x = x - y;$
$x *= y;$	<i>znači isto što i</i>	$x = x * y;$
$x /= y;$	<i>znači isto što i</i>	$x = x / y;$
$x \%= y;$	<i>znači isto što i</i>	$x = x \% y;$

Matematički čistunci će reći da je ovakav zapis matematički konzistentniji, jer ne asocira na jednakost veličina koje ne mogu biti jednake. Tako smo matematički “sumnjivu” naredbu “ $a = a + 5$ ” iz jednog od ranijih primjera mogli zamijeniti sa “ $a += 5$ ”. Kasnije ćemo ovu mogućnost koristiti dosta često.

Još jedna od neobičnih osobina jezika C++ (po kojoj se on izrazito razlikuje od jezika kao što su BASIC, FORTRAN i Pascal) je u tome što *iskaz dodjele također predstavlja izraz*. To zapravo znači da iskaz oblika “ $a = 5$ ” ne samo što dodjeljuje promjenljivoj “ a ” vrijednost “5”, nego i on sam ima vrijednost “5”, tako da se može iskoristiti kao sastavni dio nekog složenijeg izraza. Tako je sasvim moguća naredba poput sljedeće:

```
b = 3 + 2 * (a = 5);
```

Efekat ove naredbe je identičan kao da smo napisali skupinu od sljedeće dvije naredbe:

```
a = 5;  
b = 3 + 2 * a;
```

Korištenje ove osobine se ne preporučuje, jer je ona idealan put koji vodi ka pisanju nejasnih i nečitljivih programa. Ovdje je navodimo uglavnom da bismo mogli kasnije da ukažemo na jednu veoma čestu grešku koja se može nehotično napraviti prilikom upotrebe naredbi “if” i “while”. C++ programeri obično ovu osobinu ne zloupotrebjavaju. već je koriste uglavnom kada treba dodijeliti istu vrijednost u nekoliko promjenljivih. Tada umjesto

```
a = 2; b = 2; c = 2;
```

možemo pisati

```
a = b = c = 2;
```

Ovo se naziva *višestruka dodjela* (engl. *multiple assignment*). Tačan efekat ove konstrukcije je kao da smo pisali sljedeću sekvencu:

```
c = 2; b = c; a = b;
```

a ne sekvencu:

```
a = b; b = c; c = 2;
```

Dakle, operator “=” se izvodi *s desna na lijevo*, suprotno uobičajenom toku operacija. Primijetimo da efekat gornje dvije sekvence *nije isti*. Kaže se da je operator “=” *desno asocijativan*, za razliku od većine operatora, koji su *lijevo asocijativni* (za proizvoljan operator “•” se kaže da je lijevo asocijativan ukoliko se izraz “ $x \bullet y \bullet z$ ” interpretira kao “ $(x \bullet y) \bullet z$ ”, a desno asocijativan ukoliko se interpretira kao “ $x \bullet (y \bullet z)$ ”). Naravno, za slučaj *asocijativnih* operatora lijeva i desna asocijativnost daje isti efekat (npr. svejedno je da li se “ $4 + 3 + 2$ ” interpretira kao “ $(4 + 3) + 2$ ” ili kao “ $4 + (3 + 2)$ ”), ali za slučaj neasocijativnih operatora važno je znati interpretaciju. Na primjer, bitno je znati da se izraz “ $4 - 3 - 2$ ” interpretira kao “ $(4 - 3) - 2$ ” a ne kao “ $4 - (3 - 2)$ ”. Drugim riječima, operator “–” je *lijevo asocijativan*. Svi operatori koje budemo spominjali podrazumijevano će biti lijevo asocijativni, ako posebno ne naglasimo drugačije. Pored operatora dodjele, u desno asocijativne operatore spadaju i *kombinirani operatori dodjele*, poput “ $+=$ ”, ali ta činjenica je više od teoretskog nego od praktičnog značaja, s obzirom da su veoma rijetki slučajevi da se neki od kombiniranih operatora dodjele u istom izrazu upotrijebi više od jedanput.

Ovdje treba ukazati na jednu dosta čestu grešku. Naredba

```
a = a - b - c;
```

nije ekvivalentna naredbi

```
a -= b - c;
```

kako bi neko mogao brzopledo pomisliti, jer se ova posljednja naredba zapravo interpretira kao

```
a = a - (b - c);
```

koja se zapravo svodi na naredbu

```
a = a - b + c;
```

S druge strane, naredba

```
a -= b + c;
```

ekvivalentna je polaznoj naredbi!

Činjenica da dodjela predstavlja izraz omogućuje da se unutar istog izraza izvrši dodjela neke promjenljive i njen ispis na izlazni tok (mada i ovu osobinu treba izbjegavati). Tako se, na primjer, sljedeće dvije naredbe

```
c = a + b;  
cout << c;
```

mogu “upakovati” u jednu (nepreporučljivu) naredbu oblika

```
cout << (c = a + b);
```

Zgrade su u ovom slučaju neophodne. Naime, operator dodjele “=” ima veoma nizak prioritet, niži od prioriteta gotovo svih ostalih operatora (razlog za ovo bi trebao biti jasan). Tako bi se prethodni izraz bez zagrada interpretirao kao “`(cout << c) = a + b`”, što je očita besmislica, s obzirom da “`cout << c`” nije l-vrijednost, i ne može se naći sa lijeve strane operatora dodjele.

Sljedeći veoma karakteristični operatori jezika C++ su “`++`” i “`--`”. Ovo su *unarni* a ne *binarni* operatori, odnosno primjenjuju se na samo *jedan operand*. Oni *povećavaju* odnosno *smanjuju* vrijednost operanda na koji su primjenjeni za 1. Pri tome, operand mora biti l-vrijednost (najčešće ime neke promjenljive). Tako, umjesto “`a = a + 1`” ili “`a += 1`” možemo pisati samo

```
a++;
```

Ovi operatori mogu se pisati u *prefiksnoj* odnosno u *postfiksnoj* formi, odnosno bilo *ispred* bilo *iza* operanda (promjenljive). Tako smo mogli pisati i:

```
++a;
```

Ovo dvoje ipak nije isto. Razlika je u tome *šta je vrijednost* ovih izraza, i uočljiva je samo u slučaju kada se ovi operatori upotrijebe *unutar nekog složenijeg izraza* (što također ne treba zloupotrebljavati). Naime, obje konstrukcije “`a++`” i “`++a`” povećavaju sadržaj promjenljive “`a`” za 1, ali je razlika u tome *šta je rezultat* tog izraza, tj. kako će on biti iskorišten dalje unutar nekog složenijeg izraza. Rezultat izraza “`a++`” je vrijednost promjenljive “`a`” *prije* uvećavanja, dok je vrijednost izraza “`++a`” vrijednost promjenljive “`a`” *nakon* uvećavanja. Kažemo da “`++a`” obavlja *preinkrementiranje*, dok “`a++`” obavlja *postinkrementiranje*. Na primjer, neka je vrijednost promjenljive “`a`” jednaka “5”, i neka je data naredba

```
b = 3 + 2 * (a++);
```

Efekat ove naredbe je kao da smo napisali sljedeće dvije naredbe:

```
b = 3 + 2 * a;  
a = a + 1;
```

S druge strane, efekat naredbe

```
b = 3 + 2 * (++a);
```

je isti kao da smo napisali sekvencu naredbi

```
a = a + 1;  
b = 3 + 2 * a;
```

Iako u oba slučaja imamo da je nakon izvršenja naredbe vrijednost promjenljive “`a`” jednaka “6”, vrijednosti promjenljive “`b`” će se razlikovati u prvom i drugom slučaju (u prvom slučaju će ova vrijednost biti “13”, a u drugom slučaju “15”). Interesantno je napomenuti da je prioritet operatora “`++`” veoma visok, tako da smo u oba slučaja zgrade mogli izostaviti, po cijenu da program učinimo slabije čitljivim (u slučaju kada nismo sigurni u prioritet pojedinih operatora, upotreba zagrada za specifikaciju prioriteta nije na odmet, čak i u slučajevima kada njihovo korištenje nije neophodno).

Izrazi poput izraza “`3 + 2 * (++a)`” koji *mijenjaju vrijednost* neke od promjenljivih, ili koji obavljaju bilo kakvu akciju koja ne spada u puko izračunavanje vrijednosti izraza, nazivaju se izrazi sa *bočnim efektima* (engl. *side effects*). U prethodnom izrazu, kažemo da je bočni efekat izvršen nad

promjenljivom “`a`“. Specifikacija jezika C++ kaže da se, u slučaju da se nad nekom promjenljivom primjeni bočni efekat, ta promjenljiva u izrazu smije upotrijebiti samo jedanput. U suprotnom je vrijednost izraza nedefinirana (njegova vrijednost zavisi od izvedbe kompjdera), i takvi izrazi se ne smiju upotrebljavati. Tako je npr. vrijednost izraza “`a * a++`“ i “`(a++) - (a++)`“ nedefinirana. Na primjer, u prvom slučaju, zna se da je vrijednost desnog operanda operacije množenja vrijednost promjenljive “`a`“ prije uvećanja, ali se ne zna da li je će kao lijevi operand operacije množenja biti upotrijebljena vrijednost promjenljive “`a`“ prije ili poslije uvećanja. Kao posljedica toga, ovaj izraz ima nedefiniranu vrijednost. Stoga se početnicima savjetuje da izbjegavaju formiranje složenijih izraza u kojima se javljaju bočni efekti.

Analogna priča se može ispričati i za operator “`--`“. On umanjuje vrijednost svog operanda koji također mora biti l-vrijednost (obično promjenljiva) za 1, ali u zavisnosti da li je napisan *ispred* ili *iza* promjenljive obavlja *predekrementiranje* odnosno *postdekrementiranje*. Interesantno je da je jezik C++ dobio ime upravo po operatoru “`++`”, jer je C++ *poboljšana verzija* jezika poznatog pod imenom C (a jezik C je dobio ime zahvaljujući ekscentričnosti njegovih autora koji su željeli da imaju *kratko i originalno* ime za svoj novi programski jezik; usput, jedan od njihovih ranijih jezika zvao se B).

Na kraju treba ukazati na još jednu interesantnu osobinu jezika C++, koja ga također razlikuje od većine srodnih jezika. Mnogi drugi jezici, kao što je npr. Pascal, striktno razlikuju *izraze* od *naredbi*. U njima izrazi mogu biti dio naredbi, ali nikada ne mogu predstavljati naredbu. Također, naredba ne može biti iskorištena kao izraz. Međutim, razmotrimo konstrukcije poput “`a = 3`”, “`a++`” i “`cout << a`” u jeziku C++. Sve tri konstrukcije predstavljaju izraze, ali također predstavljaju i naredbe. U jeziku C++ vrijedi još opštije pravilo nego što je prikazano u ovim primjerima: *svaki izraz može biti iskorišten i kao naredba* (obrnuto ne vrijedi, odnosno svaka naredba *ne može* biti iskorištena kao izraz). Na primjer, konstrukcija “`2 + 3`” nedvojbeno predstavlja izraz, ali principijelno može biti iskorištena i kao naredba. Tako je, sa aspekta sintakse, sasvim dopušteno u programu napisati nešto poput

`2 + 3;`

Šta radi ovakva naredba? Ništa. Činjenica je da će izraz “`2 + 3`” biti izračunat, ali kako nije rečeno šta treba uraditi sa rezultatom izračunavanja ovog izraza (5), on će prosto biti ignoriran. Da bi rezultat imao smisla, nešto s njim treba *uraditi* (npr. ispisati ga, smjestiti ga u neku promjenljivu, ili uraditi nešto treće). Slijedi da bi izraz bio smisleno upotrijebljen kao naredba, on mora sadržavati neki bočni efekat. U suprotnom, upotreba izraza kao naredbe je besmislena, mada je sintaksom jezika C++ dozvoljena, tako da kompjaler neće prijaviti nikakvu grešku na naredbu poput gore navedene (bolji kompjajleri će eventualno prijaviti *upozorenje* da navedena naredba ne radi ništa). Treba uočiti razliku između izraza poput “`a + 1`” koji ne može smisleno biti upotrijebljen kao samostalna naredba, od izraza “`a += 1`” koji može biti upotrijebljen kao samostalna naredba (s obzirom da mijenja sadržaj promjenljive “`a`”).

4. Konstante i formatirani ispis

U prethodnim poglavljima upoznali smo se sa *promjenljivim*, koje služe za čuvanje podataka koji se mijenjaju tokom rada programa. S druge strane, *konstante* se koriste za čuvanje podataka koji se, nakon trenutka njihovog definiranja, više ne mijenjaju do kraja izvođenja programa. Poput promjenljivih, i konstante se također moraju deklarirati. Deklaracija konstanti se razlikuje od deklaracije promjenljivih po tome što deklaracija konstante počinje sa ključnom riječju “**const**”, i što konstante obavezno *moraju biti inicijalizirane* (bilo navodenjem početne vrijednosti unutar zagrada, bilo pomoću znaka “=”). Slijede primjeri ispravnih deklaracija konstanti:

```
const int DinaraZaMarku(35);  
const int BrojRadnihDana = 5, BrojSedmicaUGodini = 52;
```

Deklaracija poput

```
const int broj;
```

nije dozvoljena, jer konstanti “broj” nije dodijeljena vrijednost.

Uvođenje konstanti povećava čitljivost programa i olakšava njegovu izmjenu. Na primjer, deklaracija konstante “DinaraZaMarku” prepostavlja da je kurs između konvertibilne marke i srpskog dinara takav da se za 1 KM dobija 35 dinara. Međutim, kursevi valuta su često podložni promjenama. Tako, na primjer, ukoliko se kurs između marke i dinara promijeni, dovoljno je promijeniti samo definiciju ove konstante. U suprotnom bismo morali pretražiti čitav program, i mijenjati svaku pojavu broja “35” nekom drugom vrijednošću, pod uvjetom da ta pojava broja “35” zaista predstavlja kurs između marke i dinara (jer je sasvim moguće da se na nekom mjestu u programu pojavi broj “35” iz razloga koji nemaju nikakve veze sa kursom marke i dinara). Slično, ukoliko na svakom mjestu u nekom programu u kojem nam treba broj sedmica u godini koristimo konstantu “BrojSedmicaUGodini”, program će biti mnogo čitljiviji nego ako koristimo bezlični broj “52”, jer pri površnoj analizi programa teško možemo zaključiti šta broj “52” zapravo predstavlja u kontekstu u kojem je upotrijebljen. Stoga se definicija i upotreba konstanti u programima smatra izuzetno poželjnom praksom. Inače, brojevi čiji se smisao ne vidi iz samog mesta u programu na kojem su upotrijebljeni, često se nazivaju *magični brojevi* (engl. *magic numbers*), vjerojatno po analogiji sa magičnim riječima u bajkama pomoću kojih se ostvaruju neki specijalni efekti, bez ikakvog vidljivog razloga zašto. Upotreboru konstanti smanjuje se upotreba magičnih brojeva, što olakšava razumijevanje i održavanje programa.

Pokušaj da unutar programa *promjenimo vrijednost* neke konstante smatra se greškom, tj. kompjuter bi prijavio grešku da se negdje kasnije u programu pojavi naredba poput

```
DinaraZaMarku = 120;
```

Interesantno je napomenuti da se konstante mogu inicijalizirati vrijednošću nekog izraza, tako da vrijednost konstante ne mora nužno biti poznata prije nego što započne izvršavanje programa. Na primjer, razmotrimo sljedeći programske isječak:

```
int godina_rodjenja, tekuca_godina;  
cout << "Unesi godinu rođenja: ";  
cin >> godina_rodjenja;  
cout << "Unesi tekuću godinu: ";  
cin >> tekuca_godina;
```

```
const int starost = tekuca_godina - godina_rodjenja;
```

Ovaj primjer je sasvim legalan. Prvo se od korisnika traže podaci o godini rođenja i tekućoj godini, a zatim se na osnovu ovih podataka računa starost, koja se koristi za inicijalizaciju konstante "starost". Ovdje je upotrijebljena konstanta a ne promjenljiva, s obzirom da se starost, nakon što je izračunata na osnovu ulaznih podataka, više neće mijenjati do kraja programa (glavna svrha konstanti je da omoguće kompjajleru prijavu greške u slučaju da kasnije u programu nehotično probamo promjeniti vrijednost nekog podatka koji bi, po prirodi stvari, trebao da bude nepromjenljiv). Međutim, jasno je da vrijednost ove konstante ne možemo znati prije početka rada programa, odnosno prije nego što korisnik unese podatke na osnovu kojih će biti izračunata vrijednost ove konstante.

Konstanta "starost" iz prethodnog primjera očigledno se razlikuje od ranije definiranih konstanti "BrojRadnihDana" i "BrojSedmicaUGodini", čija je vrijednost očigledno poznata i prije pokretanja programa. Takve konstante nazivaju se *prave konstante* (engl. *true constants*). Preciznije, prave konstante su one konstante koje su inicijalizirane ili *brojem* ili *konstantnim izrazom* (engl. *constant expression*), koji se definira kao izraz koji se sastoji isključivo od brojeva i drugih pravih konstanti. Konstante koje nisu prave konstante možemo nazvati *neprave konstante*, mada se one u engleskoj literaturi obično nazivaju *konstantne promjenljive* (iako je, sa lingvističkog aspekta, jasno da je ovaj termin klasični oksimoron) ili, još gore, *neizmjenljive promjenljive* (engl. *unmodifiable variables*). Da bi se u listingu programa lakše razlikovale prave konstante od nepravih konstanti i promjenljivih, mnogi programeri koriste konvenciju po kojoj se za početna slova imena promjenljivih i nepravih konstanti uvijek koriste *mala slova*, a za početna slova imena pravih konstanti *velika slova*. Pored toga, u imenima pravih konstanti se izbjegava upotreba znaka ". Ove konvencije ćemo se i mi pridržavati.

Upotrebu konstanti ćemo ilustrirati na primjeru programa koji računa iznos godišnjeg poreza poreskog obveznika po britanskom sistemu naplate poreza. Po ovom sistemu, iznos poreza se računa kao jedna trećina godišnjeg prihoda korisnika umanjena za poresku naknadu, pri čemu se naknada sastoji od fiksne lične naknade od 5000 funti i naknade za djecu od 1000 funti po svakom djetetu. Očigledno su ulazni podaci neophodni za rad programa iznos godišnjeg prihoda korisnika, i broj njegove djece:

```
#include <iostream>
using namespace std;
int main() {
    const int LicnaNaknada(5000), NaknadaZaDjecu(1000);
    int prihod, broj_djece;
    cin >> prihod;
    cin >> broj_djece;
    // Oporezovani prihod je prihod umanjen za iznos naknada
    int oporezovani_prihod = prihod - LicnaNaknada
        - broj_djece * NaknadaZaDjecu;
    // Porez se računa kao cijeli broj funti
    int porez = oporezovani_prihod / 3;
    cout << prihod << oporezovani_prihod << porez << prihod - porez;
    return 0;
}
```

Neka je godišnji prihod nekog korisnika sa četvero djece 11000 funti. Mogući scenario izvršavanja ovog programa je sljedeći:

```
11000  
4  
11000200066610334
```

Ovdje odmah primijećujemo dvije stvari. Prvo, program je "ružan" jer kad ga pokrenemo, ne znamo šta se od nas traži dok ne analiziramo listing programa. Drugo, mada želimo da dobijemo rezultate koji konkretno iznose "11000", "2000", "666" i "10334", dobili smo sljepljeni broj "11000200066610334" koji nam ništa ne govori. To ne treba da nas čudi, jer nigdje nismo naveli da rezultate treba razdvojiti (već smo vidjeli da se ovo razdvajanje treba eksplisitno "isprogramirati"). U nastavku ćemo razmotriti razne načine kako ovaj program "upristojiti".

Kao što već znamo, problem sljepljenog ispisa rezultata u prethodnom programu, ako ne želimo da svaki podatak ispisujemo u novom redu, lako možemo riješiti eksplisitnim traženjem da se ispiše po jedan razmak između svaka dva rezultata, tj. pisanjem naredbe poput:

```
cout << prihod << " " << oporezovani_prihod << " " << porez  
<< " " << prihod - porez << endl;
```

Sada bi rezultat izvršavanja programa za iste ulazne podatke izgledao ovako:

```
11000  
4  
11000 2000 666 10334
```

što već ima smisla.

Bolju kontrolu nad ispisom podataka možemo postići zadavanjem *širine polja predviđene za ispis* podataka. Ovo zadavanje postiže se pomoću naredbe oblika

```
cout.width(širina)
```

gdje je "širina" proizvoljan cijeli broj, ili izraz koji ima cjelobrojnu vrijednost. Tako, ako upotrijebimo sljedeće naredbe:

```
cout << "Poreska dažbina iznosi: ";  
cout.width(8);  
cout << porez;
```

ispis će izgledati ovako:

```
Poreska dažbina iznosi:      666
```

Ovdje treba obratiti pažnju da "8" nije broj razmaka koji se ispisuju ispred podatka, nego *broj mjesta koje će zauzeti podatak*. Kako u našem slučaju podatak (broj "666") ima 3 cifre, on se dopunjuje sa 5

dodatnih razmaka ispred, tako da će ukupna širina ispisa biti 8 mesta. Ovo se obično koristi kada želimo da ispišemo skupinu podataka (čiju širinu unaprijed ne znamo) poravnati udesno (kao u primjeru koji slijedi). Nažalost, naredba “`cout.width`” djeluje samo na prvi sljedeći ispis, nakon čega se opet sve vraća na normalu. Tako, da uljepšamo izlaz iz prethodnog programa moramo pisati sljedeći niz naredbi:

```
cout << "Prihod:           "
cout.width(5);
cout << prihod << endl << endl;
cout << "Oporezovani prihod: ";
cout.width(5);
cout << oporezovani_prihod << endl;

cout << "Poreska dažbina:    ";
cout.width(5);
cout << porez << endl;
cout << "Čisti prihod:        "
cout.width(5);
cout << prihod - porez << endl;
```

Ovaj isječak će proizvesti sljedeći ispis:

Prihod:	11000
Oporezovani prihod:	2000
Poreska dažbina:	666
Čisti prihod:	10334

Ovim smo, bez sumnje, uljepšali ispis time što smo poravnali ispis udesno (primijetimo da smo, radi lakše procjene neophodne širine polja za ispis podataka, sve tekstove proširili razmacima na istu dužinu). Međutim, ovo uljepšavanje smo “skupo platili” dosadnim ponavljanjem naredbe “`cout.width`” i potrebom da “prekidamo” tok na objekar “`cout`”. Srećom, biblioteka “`iomanip`” (skraćeno od engl. *input-output manipulators*) posjeduje tzv. *manipulatore* (odnosno *manipulatorske objekte*) poput “`setw`”. Manipulatori su specijalni objekti koji se *šalju* na izlazni tok (pomoću operatora “`<<`”) sa ciljem podešavanja osobina izlaznog toka. Njihovom upotrebotom, širinu ispisa možemo postavljati “u hodu”, bez prekidanja toka, pomoću naredbi poput:

```
cout << "Prihod:           " << setw(5) << prihod << endl << endl;
cout << "Oporezovani prihod: " << setw(5) << oporezovani_prihod << endl;
cout << "Poreska dažbina:    " << setw(5) << porez << endl;
cout << "Čisti prihod:        " << setw(5) << prihod - porez << endl;
```

ili čak poput sljedeće *jedne jedine* naredbe (bez ikakvog prekidanja toka):

```
cout << "Prihod:           " << setw(5) << prihod << endl << endl
<< "Oporezovani prihod: " << setw(5) << oporezovani_prihod << endl
<< "Poreska dažbina:    " << setw(5) << porez << endl
<< "Čisti prihod:        " << setw(5) << prihod - porez << endl;
```

Primijetimo da smo u svim navedenim primjerima unutar navodnika umetali razmake, bez obzira na to što ćemo kasnije dodatno podešavati širinu ispisa brojčanim rezultata upotrebotom naredbe “`cout.width`” ili pomoću manipulatora “`setw`”. Ovo smo učinili da bismo lakše mogli proizvesti ispis koji je poravnat

uz posljednju cifru rezultata. Naravno da smo isti efekat mogli postići i bez umetanja razmaka uz prethodno pažljivo proračunavanje širina za svaki od brojčanih podataka. Tako smo isti ispis mogli postići pomoću sljedeće naredbe, u kojoj stringovne konstante ne sadrže razmake:

```
cout << "Prihod:" << setw(18) << prihod << endl << endl  
<< "Oporezovani prihod:" << setw(6) << oporezovani_prihod << endl  
<< "Poreska dažbina:" << setw(9) << porez << endl  
<< "Čisti prihod:" << setw(12) << prihod - porez << endl;
```

Očigledno je ovakvo rješenje mnogo manje elegantno od prethodnog rješenja u kojem se može smatrati da "tekstualni dio" ispisa i "brojčani dio" ispisa u svakom redu zauzimaju istu širinu, pri čemu je ta širina u tekstualnom dijelu ispisa ostvarena dopunjavanjem stringovnih konstanti do iste fiksne dužine, a u brojčanom dijelu ispisa slanjem manipulatora "setw(5)" na izlazni tok.

Konačno, uljepšana verzija programa za računanje poreza i oporezovanog prihoda mogla bi izgledati poput sljedećeg:

```
#include <iostream>  
#include <iomanip>  
using namespace std;  
int main() {  
    const int LicnaNaknada(5000), NaknadaZaDjecu(1000);  
    int prihod, broj_djece;  
  
    cout << "Unesi god. prihod: ";  
    cin >> prihod;  
    cout << "Unesi broj djece: ";  
    cin >> broj_djece;  
    cout << endl;  
  
    int oporezovani_prihod = prihod - LicnaNaknada  
        - broj_djece * NaknadaZaDjecu;  
    int porez = oporezovani_prihod / 3;  
  
    cout << "Prihod: " << setw(5) << prihod << endl << endl  
        << "Oporezovani prihod: " << setw(5) << oporezovani_prihod << endl  
        << "Poreska dažbina: " << setw(5) << porez << endl;  
        << "Čisti prihod: " << setw(5) << prihod - porez << endl;  
  
    return 0;  
}
```

Mogući scenario izvršavanja ovog programa prikazan je na sljedećoj slici:

```
Unesi god. prihod: 11000  
Unesi broj djece: 4  
  
Prihod: 11000  
  
Oporezovani prihod: 2000  
Poreska dažbina: 666  
Čisti prihod: 10334
```

Za slučaj kada je zadana širina ispisa manja od minimalne neophodne širine potrebne da se ispiše rezultat, naredba ”cout.width” (odnosno manipulator ”setw”) se ignorira.

5. Druge vrste cjelobrojnih tipova

U dosadašnjim izlaganjima, sve promjenljive koje smo deklarirali imale su tip “**int**”, i to je bio jedini tip podataka sa kojim smo se do sada susreli. Rekli smo da tip “**int**” predstavlja cjelobrojni tip podataka, ali to je zapravo samo dio priče, s obzirom da jezik C++ posjeduje više različitih cjelobrojnih tipova podataka, pri čemu je “**int**” samo jedan od njih. Tačnije, jezik C++ posjeduje četiri osnovna tipa cjelobrojnih podataka, koji se redom nazivaju “**char**”, “**short**”, “**int**” i “**long**”. Ovi tipovi se razlikuju po tome koliko memoriskog prostora računar rezervira za promjenljive koje su deklarirane tim tipom. Efektivno, ovo se odražava na opseg vrijednosti koje se mogu zapamtiti u promjenljivim kojima je dodijeljen neki od ovih tipova. Napomenimo da je za precizno razumijevanje izlaganja koji slijede neophodno da čitatelj odnosno čitateljka posjeduju izvjesna predznanja o pojmovima kao što su bit, bajt, itd. kao i načinu zapisivanja brojeva u računarskoj memoriji.

Standard jezika C++, što je prilično čudno, ne predviđa koliko tačno memorije zauzimaju promjenljive pojedinih tipova. Jedino se garantira da promjenljive tipa “**char**” zauzimaju tačno jedan bajt, a da promjenljive ostalih cjelobrojnih tipova uvijek zauzimaju cijeli broj bajtova. Pored toga, još se garantira da vrijedi

$$\text{veličina}(\mathbf{char}) \leq \text{veličina}(\mathbf{short}) \leq \text{veličina}(\mathbf{int}) \leq \text{veličina}(\mathbf{long})$$

Kod većine kompjlera koji se danas koriste, promjenljive tipa “**short**” zauzimaju 2 bajta, dok promjenljive tipa “**long**” zauzimaju 4 bajta, a u nekim verzijama kompjlera 8 bajta. Najšarolikija je situacija sa tipom “**int**”. Do nedavno su preovladavali kompjajleri kod kojih promjenljive tipa “**int**” zauzimaju 2 bajta, dok danas pretežno susrećemo kompjajlere kod kojih promjenljive tipa “**int**” zauzimaju 4 bajta u memoriji. To praktično znači da su kod većine današnjih kompjajlera tipovi “**int**” i “**long**” sinonimi (dok su donedavno tipovi “**int**” i “**short**” bili sinonimi). Da bismo stekli osjećaj o najmanjim i najvećim vrijednostima koje se mogu smjestiti u promjenljive odgovarajućih tipova, sljedeća tabela daje zavisnost između zauzeća memorije u bajtima i najmanje odnosno najveće vrijednosti koja se može smjestiti u promjenljivu koja zauzima navedenu količinu memorije:

<u>Količina memorije</u>	<u>Najmanja vrijednost</u>	<u>Najveća vrijednost</u>
1 bajt	-128	+127
2 bajta	-32768	+32767
4 bajta	-2147483648	+2147483647
8 bajta	-9223372036854775808	+9223372036854775807

Na primjer, pretpostavimo da koristimo kompjajler kod kojeg promjenljive tipa “**short**” zauzimaju 2 bajta a promjenljive tipa “**long**” 4 bajta (najčešći slučaj), i neka su date sljedeće deklaracije:

```
short a, b;  
long c;
```

Tada će promjenljive “a” i “b” moćiće da prime opseg brojeva od -32768 do +32767, dok će promjenljiva “c” moći da primi znatno širi opseg brojeva u rasponu od -2147483648 do +2147483647. Pošto je tip “**int**” tipično najdiskutabilniji po pitanju opsega koji prihvata, ukoliko želimo da program koji pišemo bude što prenosiviji, u smislu da što manje ovisi od osobina upotrijebljenog kompjajlera, dobra je praksa sve cjelobrojne promjenljive za koje znamo da neće uzimati veliki opseg vrijednosti (npr. promjenljive koje opisuju dan, mjesec i godinu nečijeg rođenja) deklarirati kao promjenljive tipa

“**short**”, a promjenljive za koje očekujemo da mogu uzimati veće vrijednosti (npr. cijene) deklarirati kao promjenljive tipa “**long**”.

Pomoću operatora ”**sizeof**” može se saznati koliko bajta zauzima pojedini tip na Vašem konkretnom C++ kompjalu. Ovaj operator kao argument prihvata ime nekog tipa ili izraz, a daje kao rezultat broj bajta koje zauzima navedeni tip, ili rezultat navedenog izraza (nakon obavljenog izračunavanja). Kako se koristi ovaj operator, lako se vidi iz sljedećeg primjera:

```
cout << "Tip char zauzima " << sizeof(char) << "bajta\n";
cout << "Tip short zauzima " << sizeof(short) << "bajta\n";
cout << "Tip int zauzima " << sizeof(int) << "bajta\n";
cout << "Tip long zauzima " << sizeof(long) << "bajta\n";
```

Ukoliko se kao argument operatora ”**sizeof**” koristi ime neke promjenljive (ili, općenitije, neki izraz), zgrade *nisu neophodne*, tako da se umjesto ”**sizeof(a)**” može pisati samo ”**sizeof a**” (ukoliko je argument *ime tipa*, zgrade su uvijek neophodne). Međutim, s obzirom da operator ”**sizeof**” ima *veoma visok prioritet*, zgrade su gotovo uvijek neophodne ukoliko nas zanima veličina nekog *izraza*. Tako, na primjer, ”**sizeof a+b**” neće biti shvaćeno kao ”**sizeof(a+b)**”, što je vjerovatno bila namjera programera, nego kao ”**sizeof(a)+b**”.

Umjesto ”**short**” i ”**long**” može se pisati i ”**short int**” i ”**long int**”, da se istakne da se radi o ”kratkim” i ”dugim” cjelobrojnim tipovima. Dakle, legalne su i deklaracije poput sljedećih:

```
short int a;
long int b;
```

Kod nekih verzija kompjajlera kod kojih promjenljive tipa ”**long**” zauzima 4 bajta, uveden je i tip ”**long long**” (ili ”**long long int**”) čije promjenljive zauzimaju 8 bajta, npr.

```
long long veliki_broj;
```

Ipak, treba voditi računa da ovaj tip nije predviđen standardom.

Postavlja se posve prirodno pitanje zašto su uvedena četiri cjelobrojna tipa, a ne samo jedan. Ako tip ”**char**”, o kojem ćemo detaljno govoriti kasnije, privremeno ostavimo po strani, razlozi su uglavnom historijske prirode:

- 1) Memorija je nekada bila jako skupa pa se vodilo računa o utrošku svakog bajta. Ako je potrebno memorirati 100000 podataka od kojih svaki zauzima opseg od 1 – 100, posve je neracionalno rezervirati 4 bajta za svaki podatak (i potrošiti 400000 bajta memorije), kada je dovoljno rezervirati samo bajt po podatku, čime je utrošak memorije svega 100000 bajta. Memoriju je, naravno, važno štedjeti, ali ne treba ni pretjerivati (primjer nepomišljene štednje memorije doveo je do pojave čuvenog milenijskog baga Y2K).
- 2) Računari su prije sporije obradivali veće brojeve nego manje. U doba 8-bitnih procesora to je bilo izrazito izraženo. To je u suštini tačno i danas, ali sa današnjim 32-bitnim i 64-bitnim procesorima računari uglavnom obrađuju brojeve dužine do 4 bajta istom brzinom neovisno od njihove stvarne veličine (N-bitni procesori mogu brojeve dužine do N bita obrađivati ”u jednom potezu”). Dakle, danas nema osobitih ”brzinskih” zahtjeva koji bi zahtijevali da cjelobrojne promjenljive budu kraće od 4 bajta.

Pomoću prefiksa “**`unsigned`**” moguće je naznačiti da promjenljiva neće uzimati negativne vrijednosti, čime se dvostruko povećava raspon pozitivnih vrijednosti koje promjenljiva može primiti. Tako, ako izvršimo deklaraciju

```
unsigned int a;
```

promjenljiva “`a`” će, na kompjleru kod kojih promjenljive tipa “**`int`**” zauzimaju 4 bajta, moći primiti vrijednosti iz opsega $0 - 4294967295$ umjesto iz opsega $-2147483648 - 2147483647$.

Ako iza riječi “**`unsigned`**” slijedi riječ “**`int`**”, riječ “**`int`**” je moguće izostaviti, tako da je valjana i sljedeća deklaracija:

```
unsigned a;
```

Suprotan prefiks prefikuksu “**`unsigned`**” je prefiks “**`signed`**” kojim se navodi da će promjenljiva *pamtiti i predznak*, ali je ovaj prefiks suvišan, jer se podrazumijeva ako se izostavi. Ipak, radi jasnoće, dozvoljeno je pisati i deklaraciju poput

```
signed int a;
```

Činjenica da svi cjelobrojni tipovi posjeduju *ograničen opseg* može dovesti do neugodne pojave nazvane *prekoračenje* (engl. *overflow*). Naime, ukoliko promjenljivoj dodijelimo vrijednost izvan dozvoljenog opsega, ili ako prilikom izvođenja računskih operacija dođe do prekoračenja dozvoljenog opsega vrijednosti, rezultat neće biti tačan. Možemo zamisliti da umjesto *brojnjog pravca* poznatog u matematici, imamo *brojni krug*, u kojem iza “najvećeg” broja ponovo dolazi “najmanji” broj. Ilustrirajmo to na primjeru tipa “**`unsigned short`**” na kompjleru kod kojeg promjenljive tipa “**`short`**” zauzimaju 2 bajta (istu pretpostavku ćemo koristiti u svim narednim izlaganjima). Tako, sa promjenljivim ovog tipa imamo sljedeći “račun”:

$$\begin{aligned} 65532 + 1 &= 65533 \\ 65532 + 2 &= 65534 \\ 65532 + 3 &= 65535 \\ 65532 + 4 &= 0 \\ 65532 + 5 &= 1 \\ 65533 + 6 &= 2 \end{aligned}$$

itd. Matematičari bi rekli da se račun vrši “po modulu 2^N ” gdje je N broj bita promjenljive (16 u našem slučaju). Iz ovoga slijedi da je:

$$\begin{aligned} 65536 &\equiv 0 \\ 65537 &\equiv 1 \end{aligned}$$

itd. Namjerno pišemo “ \equiv ” umjesto “ $=$ ” da naznačimo da se ne radi o jednakosti u matematskom smislu. Matematičari će u ovakovom “poređenju” prepoznati zapravo *kongurenciju po modulu 2^N* .

Slično vrijedi i “podbacimo” li ispod donje granice dozvoljenog opsega (engl. *underflow*). Tako, uz tip “**`unsigned short`**” imamo:

$$\begin{aligned} 3 - 2 &= 1 \\ 3 - 3 &= 0 \\ 3 - 4 &= 65535 \end{aligned}$$

$$3 - 5 = 65534$$

Dakle, smisao računara za “matematiku” je pomalo “čudan”. U ovo se možemo uvjeriti ako napišemo sljedeću sekvencu naredbi:

```
unsigned int a;
a = -2;
cout << a;
```

Nakon gornjeg razmatranja neće nam biti previše čudno zašto će računar kao rezultat ispisati broj “65534”. Ako Vam sve ovo djeluje “uvrnuto” i zbunjujuće, poštujte sljedeća pravila:

- NEMOJTE dodjeljivati promjenljivoj vrijednosti koje su izvan dozvoljenog opsega;
- NEMOJTE dozvoliti da rezultat neke operacije premaši opseg promjenljive u koju se rezultat smješta.

Sretna je okolnost da kod većine današnjih kompjajlera promjenljive tipa “**int**”, koje se najčešće koriste, zauzimaju 4 bajta, tako da opseg vrijednosti koje se u njih mogu smjestiti uglavnom zadovoljavaju potrebe većine primjena. Znatno je teža situacija bila u vrijeme kada su promjenljive tipa “**int**” zauzimale 2 bajta, tako da je njihovo korištenje (umjesto npr. promjenljivih tipa “**long**”) dovodilo do ozbiljnih problema. Da bismo se bolje upoznali sa pojmom prekoračenja (koja nekada može zaista da ima veoma neugodne posljedice), korisno je razmotriti i sljedeće primjere, koji zahtijevaju malo veće udubljivanje (ako Vam izloženi primjeri djeluju “šokantni”, nemojte misliti da je to “ružno svojstvo” samo jezika C++: od istog problema “pate” i svi drugi programski jezici). Šta se dešava kada se prekrši drugo pravilo, pokazuje sljedeći primjer:

```
short int a, b, c;
a = 20000; b = 20000;
c = a + b;
cout << c;
```

Pokušajte sami odgonetnuti zašto je krajnje dejstvo ove sekvence naredbi ispis broja “–25536”. Manje je očigledno zašto sljedeća sekvenca daje isti (pogrešan) rezultat:

```
short int a, b;
a = 20000; b = 20000;
cout << a + b;
```

pa čak i sljedeća sekvenca:

```
short int a, b;
long int c;
a = 20000; b = 20000;
c = a + b;
cout << c;
```

U čemu je problem? C++ podrazumijeva da je rezultat izraza *onog tipa kakav je tip najvećeg (po opsegu) od argumenata koji učestvuju u izrazu*. U sva tri primjera oba argumenta su tipa “**short int**” pa je i rezultat tipa “**short int**” (neovisno od toga što se u trećem primjeru rezultat smješta u promjenljivu tipa “**long int**”). Stoga, u sva tri slučaja dolazi do prekoračenja, jer rezultat izraza “20000 + 20000” premašuje opseg tipa “**short int**”.

Bitno je napomenuti da se i sami cijeli brojevi u C++ programima posmatraju kao da su tipa “**int**”,

ukoliko po svojoj vrijednosti upadaju u opseg tipa “**int**”. Ovo može stvoriti poseban problem kod kompjlera kod kojih tip “**int**” zauzima 2 bajta. Na primjer, kod takvih kompjlera će čak i naredba

```
cout << 20000 + 20000;
```

dovesti do pogrešnog rezultata (-25536), s obzirom da se operandi tretiraju kao da su tipa “**int**”, tako da dolazi do prekoračenja. S druge strane, naredba

```
cout << 35000 + 5000;
```

daje ispravan rezultat, s obzirom da prvi argument (35000) ne može stati u 2 bajta, pa se ne tretira kao tip “**int**” nego kao prvi sljedeći tip po veličini (na razmatranom kompjleru, to je tip “**unsigned int**”, s obzirom da je $35000 < 65535$). Stoga je rezultat sabiranja $35000 + 5000$ tipa “**unsigned int**”. Kako 40000 lijepo može da stane u tip “**unsigned int**”, dobija se tačan rezultat.

Ovakvi problemi mogu da budu uzrok gadnih frustracija. Na primjer, čak i kod kompjlera kod kojih tip “**int**” zauzima 32 bajta, naredba

```
cout << 2000000000 + 2000000000;
```

neće dati tačan rezultat. Da bi se izbjegli problemi ovog tipa, jezik C++ uvodi konvenciju da se dodavanjem sufiksa “**L**”, “**U**” ili “**UL**” na broj eksplicitno signalizira da broj treba tretirati kao tip “**long**”, “**unsigned**”, odnosno “**unsigned long**” (umjesto velikih mogu se koristiti i mala slova, tako da su sufiksi “**l**”, “**u**” i “**ul**” također legalni, samo ih nije dobro koristiti, pogotovo što se malo slovo “**l**” po izgledu gotovo ne razlikuje od cifre “**1**”). Tako je broj “**20000**” tipa “**int**”, ali je broj “**20000L**” tipa “**long**”. Stoga će naredba

```
cout << 20000L + 20000;
```

ispisati tačan rezultat, čak i kod kompjlera kod kojih tip “**int**” zauzima 2 bajta (umjesto “**20000L**” pomoglo bi i “**20000U**”). Iz istog razloga, naredba

```
cout << 2000000000UL + 2000000000;
```

proizvodi tačan rezultat (zbir 4000000000 lijepo može stati u tip “**unsigned long**”). Sufiks “**LL**” može se koristiti za forsiranje tipa “**long long**” kod kompjlera koji poznaju taj tip.

U nekim slučajevima je potrebno izvršiti *eksplicitnu pretvorbu jednog tipa u drugi*. Stoga, jezik C++ dopušta da se eksplicitno promijeni tip neke promjenljive, konstante, broja ili čitavog izraza korištenjem *operatora za promjenu tipa* (engl. *type-casting operator*). Ovaj operator se koristi na jedan od dva navedena načina:

(ime tipa) izraz
ime tipa(izraz)

Prvi način je naslijeden iz jezika C, dok je drugi način uveden u jeziku C++. U drugom načinu, sintaksa upotrebe operatora za promjenu tipa podsjeća na sintaksu za *poziv funkcije*, tako da govorimo o tzv. *funkcijskoj* ili *konstruktorskoj notaciji*. Ipak, funkcionska (konstruktorska) notacija se može koristiti samo ako se ime tipa sastoji od *samo jedne riječi*. Na primjer, obje konstrukcije “**(long) a**” i “**long(a)**” su dozvoljene i ravnopravne, međutim konstrukcija “**unsigned long(a)**” nije legalna, a konstrukcija “**(unsigned long) a**” jeste. Stoga će, na primjer, sljedeća naredba

```
cout << long(20000) + 20000;
```

ili, ekvivalentno, naredba

```
cout << (long)20000 + 20000;
```

ispisati ispravan rezultat "40000" čak i na kompjajlerima kod kojih tip "**int**" zauzima 2 bajta. Naravno, ovdje je umjesto "**long**(20000)" ili "(**long**)20000" bilo lakše pisati "20000L". Pravu primjenu operatora za pretvorbu tipa dobija tek kada se primijeni na neku *promjenljivu* ili *izraz*. Na primjer, sljedeća sekvenca naredbi demonstrira kako ispravno pomnožiti dvije promjenljive tipa "**short int**", pri čemu rezultat, koji ne može da stane u tip "**short int**", želimo smjestiti u promjenljivu tipa "**long int**":

```
short int a, b;  
long int c;  
a = 20000; b = 10;  
c = (long)a * b;  
cout << c;
```

Ovdje smo pomoću "*type-casting*" operatorka "(**long**)" pretvorili promjenljivu "a" u tip "**long**" (odnosno "**long int**") čime smo postigli da rezultat bude tačan.

Na kraju, za matematički orientirane čitatelje i čitateljke, daćemo *gotovu matematičku formulu* kojom se može predvidjeti rezultat neke operacije u slučaju da dođe do prekoračenja. Neka je x tačna (matematička) vrijednost nekog broja, izraza, itd. i neka je N broj bita koji zauzima tip izraza koji se izračunava. Neka izraz $\lfloor x \rfloor$ označava cijeli dio broja x , tj. najveći cijeli broj koji je manji od ili jednak x (npr. $\lfloor 3.14 \rfloor = 3$ i $\lfloor -1.32 \rfloor = -2$). Tada je rezultat u slučaju prekoračenja

$$x - 2^N \lfloor x / 2^N \rfloor$$

ako je tip izraza *bez znaka*, odnosno

$$x - 2^N \lfloor (x + 2^{N-1}) / 2^N \rfloor$$

ako je tip izraza *sa znakom*. Npr. provjerimo zašto smještanje broja -2 u promjenljivu tipa "**unsigned int**" ($N = 16$, bez znaka) daje rezultat 65534:

$$-2 - 2^{16} \lfloor -2 / 2^{16} \rfloor = -2 - 2^{16} \lfloor -2^{15} \rfloor = -2 - 2^{16} \cdot (-1) = -2 + 65536 = 65534$$

kao i zašto sabiranje brojeva 20000 i 20000 korištenjem tipa "**short int**" ($N = 16$, sa znakom) umjesto očekivanog rezultata 40000 daje rezultat -25536 :

$$\begin{aligned} 40000 - 2^{16} \lfloor (40000 + 2^{15}) / 2^{16} \rfloor &= 40000 - 2^{16} \lfloor 72768 / 65536 \rfloor = 40000 - 2^{16} \cdot 1 = \\ &= 40000 - 65536 = -25536 \end{aligned}$$

6. Realni tipovi podataka

U prethodnom poglavlju smo se detaljno upoznali sa cjelobrojnim tipovima podataka. Međutim, postoje mnogi problemi koje nije moguće riješiti korištenjem samo cjelobrojnih vrijednosti. Stoga jezik C++ uvodi *realne tipove podataka*, koje omogućavaju pamčenje (doduše, aproksimativno) vrijednosti koje se iskazuju pomoću realnih brojeva. Jezik C++ predviđa tri tipa za pamčenje realnih vrijednosti: “**float**”, “**double**” i “**long double**”. Razlika između ovih tipova je u tome što promjenljive tipa “**float**” zauzimaju manje memorije, ali imaju manji opseg i manju preciznost (mogu zapamtitи manje tačnih cifara), dok promjenljive tipa “**double**” zauzimaju više memorije, ali nude veći opseg i veću preciznost. Tip “**long double**” omogućuje izuzetno veliki opseg i preciznost, i namijenjen je uglavnom za složenije inžinjerske proračune. Najviše se koristi tip “**double**”. Napomenimo da se prefiks “**unsigned**” ne može koristiti uz realne promjenljive, odnosno one su uvijek *sa znakom*. Slijede neki primjeri realnih promjenljivih (tj. promjenljivih realnog tipa):

```
float temperatura;
double duzina, sirina, povrsina;
```

Realnim promjenljivim mogu biti pridružene *realne vrijednosti* (preciznije, prividno realne vrijednosti, jer je njihov opseg i broj tačnih cifara koje se pamte *ograničen*). Realni izrazi mogu sadržavati *realne brojeve* (odnosno *realne brojčane konstante*), za koje je karakteristično da imaju *decimalnu tačku*, i, opcionalno, cjelobrojni *eksponent*, koji se piše iza oznake “E” ili “e”, pri čemu “xEy” (ili “xeY”) zapravo znači “ $x \cdot 10^y$ ”. Slijedi nekoliko primjera ispravno napisanih cijelih brojeva:

15.423	
3.0	
3.	(Ovo je isto kao i 3.0)
-0.1234	
-.1234	(Ovo je isto kao i -0.1234)
55E6	(Ovo znači $55 \cdot 10^6$)
-3.8E3	
12.0e-3	

Obratimo pažnju na sljedeće: “3” je *cijeli broj*, dok je “3.0” (ili samo “3.”) *realan broj*, iako im je vrijednost ista. Kakvog ovo ima smisla, vidjećemo uskoro.

U primjerima koji slijede ćemo prepostavimo da imamo sljedeće deklaracije promjenljivih:

```
int brojac;
double poluprecnik;
```

kao i definiciju realne konstante

```
const double PI(3.141592654);
```

(podsjetimo se da je smisao *konstanti* u tome da nas kompjajler upozori ako kasnije u programu nehotice pokušamo da promijenimo njenu vrijednost naredbom poput “`PI = 4`”). Realni izrazi mogu sadržavati realne operande (brojeve i promjenljive), ali mogu sadržavati i *cjelobrojne operande*, koji se u tom slučaju *automatski konvertiraju* u realne. Na primjer, kada se izračunava izraz

```
2 * PI * poluprecnik
```

cijeli broj “2” se automatski konvertira u realnu vrijednost “2 . 0” (odnosno “2 . ”). Iz istog razloga, cijeli broj može bez problema biti dodijeljen realnoj promjenljivoj, na primjer:

```
poluprecnik = 3;
```

Ovaj tip automatske konverzije, kod kojeg se uži tip (po skupu mogućih vrijednosti) konvertira u širi tip, nazivamo *promocija*. Moguća je i obrnuta dodjela, tj. dodjela u kojoj se realna vrijednost dodijeljuje cjelobrojnoj promjenljivoj. U tom slučaju se dešava *automatsko odsjecanje decimala*. Npr. ako imamo sljedeću deklaraciju:

```
int brojac;
```

nakon naredbe

```
brojac = PI * 10;
```

vrijednost promjenljive “*brojac*” će biti “31”. Iako se i ovdje dešava *automatska konverzija tipa*, da bi program bio čitkiji (a i da bismo bili sigurni šta radimo), dobra je praksa da konverziju tipa zatražimo *eksplicitno* pomoću *type-casting* operatora (napomenimo da će neki kompjaleri, u slučaju da konverziju tipa ne zatražimo eksplisitno, prijaviti *upozorenje*, ali ne i *grešku*). Na primjer:

```
brojac = (int)(PI * 10);
```

Zašto je izraz “*PI * 10*” u zagradi? Stvar je u tome što *type-casting* operator ima veoma visok prioritet, veći od operacije množenja, tako da ako izostavimo zagrade, kao u naredbi

```
brojac = (int)PI * 10;
```

rezultat će biti “30”, jer tada *type-casting* djeluje samo na konstantu “*PI*” čime se dobija rezultat “3”, koji se nakon toga množi sa 10, tako da je krajnji rezultat “30”. Alternativno, možemo koristiti i funkciju notaciju *type-casting* operatora, i pisati

```
brojac = int(PI * 10);
```

Realni izrazi mogu se formirati na *sličan način* kao i cjelobrojni izrazi, koristeći aritmetičke operatore. Osnovni operatori koji se koriste za realnu aritmetiku su sljedeći:

+	<i>sabiranje</i>
-	<i>oduzimanje</i>
*	<i>množenje</i>
/	<i>dijeljenje</i>

Treba obratiti pažnju da operator “%” *nije definiran* za operande realnog tipa.

Pri pretvaranju matematskih izraza u C++ potreban je izvjestan oprez, naročito kada pretvaramo izraze sa razlomcima. Na primjer, ako u jeziku C++ želimo napisati sljedeći izraz:

$$\frac{a+b}{c+\frac{d}{e+f}}$$

ne možemo samo pisati znak “/” umjesto razlomačke crte, tj. pisati

```
a + b / c + d / e + f
```

jer će, radi prioriteta operacija (dijeljenje ima prioritet), ovo biti shvaćeno kao:

$$a + \frac{b}{c} + \frac{d}{e} + f$$

Ispravno bi bilo pisati:

$$(a + b) / (c + d / (e + f))$$

Slijedi primjer jednog jednostavnog programa koji ilustrira realnu aritmetiku:

```
#include <iostream>
using namespace std;
int main() {
    const double PI(3.141592654);
    double poluprecnik;
    cin >> poluprecnik;
    double obim = 2 * PI * poluprecnik;
    double povrsina = PI * poluprecnik * poluprecnik;
    cout << poluprecnik << " " << obim << " " << povrsina << endl;
    return 0;
}
```

Jedan mogući scenario izvršavanja ovog programa je sljedeći:

```
3.2
3.2 20.106193 32.169909
```

Naravno, ovaj program možemo učiniti “ljubaznijim”, kao u sljedećem primjeru:

```
#include <iostream>
using namespace std;
int main() {
    const double PI(3.141592654);
    double poluprecnik;
    cout << "Unesi poluprečnik: "
    cin >> poluprecnik;
    double obim = 2 * PI * poluprecnik;
    double povrsina = PI * poluprecnik * poluprecnik;
    cout << "Obim je " << obim << endl
        << "Površina je " << povrsina << endl;
    return 0;
}
```

Mogući scenario izvršavanja ovog programa je sljedeći:

```
Unesi poluprečnik: 3.2
Obim je 20.106193
Površina je 32.169909
```

Primijetimo da, iako smo morali da iz imena promjenljivih izbacimo naša slova, nema razloga da to radimo i unutar stringova, koji ionako služe samo za uljepšavanje ispisa. Također, bitno je shvatiti da imena promjenljivih ne moraju da imaju veze sa njihovom ulogom u programu (ona samo pomažu *nama* da lakše povežemo šta nam predstavljaju pojedine promjenljive u programu). Potpuno isti efekat dao bi i slijedeći program (koji se ne preporučuje):

```
#include <iostream>
using namespace std;
int main() {
    const double PI(3.141592654);
    double x;
    cout << "Unesi poluprečnik: "
    cin >> x;
    double y = 2 * PI * x;
    double z = PI * x * x;
    cout << "Obim je " << y << endl
        << "Površina je " << z << endl;
    return 0;
}
```

Što je još gore, isti efekat (i tačne rezultate) bi dao i ovakav program:

```
#include <iostream>
using namespace std;
int main() {
    const double tezina(3.141592654);
    double povrsina;
    cout << "Unesi poluprečnik: "
    cin >> povrsina;
    double poluprecnik = 2 * tezina * povrsina;
    double obim = tezina * povrsina * povrsina;
    cout << "Obim je " << poluprecnik << endl
        << "Površina je " << obim << endl;
    return 0;
}
```

U ovom programu smo potpuno pogrešno imenovali promjenljive (npr. imenom “povrsina” nazivamo poluprečnik, konstantu “PI” smo nazvali “tezina” itd.), ali to računaru *ni najmanje ne smeta*. Njemu smisao imena promjenljivih ne znači *ama baš ništa*. To može smetati samo *onome ko pokušava da shvati program*. Naravno da je pisanje ovakvih programa (osim ako nam nije cilj da namjerno pišemo neshvatljive programe) *vrlo loša praksa*.

Jedna interesantna stvar vezana je za operaciju dijeljenja. Operator “/” obavlja dvije različite funkcije (*cjelobrojno dijeljenje i dijeljenje*) u ovisnosti od toga šta su njegovi operandi. Tako ako su oba operanda *cjelobrojnog tipa*, rezultat je također cijeli broj. Ako je *makar jedan* operand realnog tipa, rezultat će također biti realan broj. Tako, dejstvo naredbe

```
cout << a / b;
```

ovisi od toga kakvog su tipa promjenljive “a” i “b”. Iz istog razloga naredba

```
cout << 11 / 4;
```

rezultira ispisom broja “2”, a naredba

```
cout << 11. / 4;
```

rezultira ispisom broja “2.75”. Tačka unutar broja “11.” proglašila je prvi operand za realan broj, čime je i čitav rezultat realan broj.

Prirodno se postavlja pitanje kako bismo mogli ispisati *decimalni* rezultat dijeljenja 2 *cjelobrojne* promjenljive “a” i “b”. Odgovor leži u primjeni *type-casting* operatora, čime ćemo promijeniti tip jednog od operanada u “**double**”:

```
cout << (double)a / b;
```

Razumije se da smijemo koristiti i funkciju (konstruktorsku) notaciju:

```
cout << double(a) / b;
```

Međutim, ne smijemo pisati

```
cout << double(a / b);
```

jer će u tom slučaju prvo biti izračunat *cjelobrojni* količnik “a / b”, nakon čega će dobijeni (*cjelobrojni*) rezultat biti pretvoren u realnu vrijednost, čime nećemo postići ono što smo željeli.

Na sličan način, ako je potrebno da *necjelobrojni* rezultat dijeljenja dvije *cjelobrojne* promjenljive “a” i “b” dodijelimo *realnoj* promjenljivoj “c”, takođe moramo koristiti *type-casting* operator:

```
c = (double)a / b;
```

Samo “c = a / b” nije dovoljno, jer kako su “a” i “b” *cjelobrojni*, rezultat njihovog dijeljenja je *također cjelobrojan*, mada se rezultat smješta u realnu promjenljivu. O ovom fenomenu već smo govorili ranije kod opisa pojave prekoračenja kod *cjelobrojne aritmetike*.

Već je rečeno da operator “%” *ne radi sa realnim operandima*. Ako nam je iz bilo kojeg razloga neophodno da simuliramo izraz oblika “a % b” pri čemu je makar jedan od operanada “a” ili “b” realan broj, možemo se poslužiti činjenicom da je ovaj izraz ekvivalentan izrazu “a - b * int(a / b)” koji je sasvim dobro definiran i u slučaju da su “a” ili “b” (ili oba) realni operandi.

Pri ispisu realnih vrijednosti također postoji mogućnost zadavanja željenu širinu ispisa. Međutim, ovdje postoji i mogućnost zadavanja *broja decimalnih mjeseta*. Da bismo ovo vidjeli, razmotrimo sljedeći primjer. Ako se u program iz prethodnog primjera unese ulazna vrijednost “3”, naredba

```
cout << poluprecnik << " " << obim << " " << povrsina << endl;
```

ispisaće nešto poput:

3 18.849556 28.274334

Naredba ”cout.width”, koju smo već upoznali, djeluje i na realne promjenljive i izraze, tako da će sljedeća sekvenca naredbi:

```
cout.width(8);
cout << poluprecnik;
cout.width(12);
cout << obim;
cout.width(12);
cout << povrsina << endl;
```

ispisati nešto poput:

3 18.849556 28.274334

Slično kao i pri ispisu cijelobrojnih vrijednosti, ako u program uključimo i biblioteku ”iomanip”, možemo koristiti manipulator ”setw” za podešavanje širine ispisa ”u hodu”, tj. možemo pisati

```
cout << setw(8) << poluprecnik << setw(12) << obim << setw(12)
<< povrsina << endl;
```

Međutim, rekli smo da je kod ispisa realnih vrijednosti, moguće podešavati i željeni broj tačnih cifara prilikom ispisa. Ovo se obavlja pomoću naredbe ”cout.precision”. Tako će, na primjer, sekvenca

```
cout.precision(5);
cout << PI;
```

ispisati konstantu ”PI” zaokruženu na 5 tačnih cifara, odnosno na 4 decimale (tj. ”3.1416”). Naredba ”cout.precision” može se kombinirati sa naredbom ”cout.width”, tako da će sekvenca naredbi

```
cout.precision(5);
cout.width(10);
cout << PI;
```

ispisati konstantu ”PI” zaokruženu na 4 decimale, dopunjenu razmacima tako da ukupna širina ispisa bude 10 mjesta (tj. biće dodana još 4 razmaka ispred):

3.1416

Kao i naredba ”cout.width”, tako i naredba ”cout.precision” djeluje samo na prvi slijedeći ispis. Manipulator ”setprecision”, definiran u biblioteci ”iomanip”, omogućava podešavanje željenog broja decimala ”u hodu”, npr. pomoću naredbe poput:

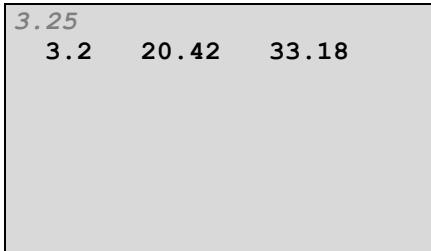
```
cout << setprecision(5) << PI;
```

Tako, ako u ranije navedeni (”neljubazni”) primjer programa za proračun obima i površine kruga, ubacimo sljedeću naredbu za ispis:

```
cout << setw(5) << setprecision(2) << poluprecnik << setw(8)
<< setprecision(4) << obim << setw(8) << setprecision(4)
```

```
<< povrsina << endl;
```

tada će jedan od mogućih scenarija izvršavanja ovog programa biti sljedeći:



Veoma veliki i veoma mali brojevi ispisuju se u *eksponencijalnoj notaciji* (sa slovom “e”, za koje smo već rekli da označava “puta 10 na”). Tako je efekat naredbi:

```
cout << 173286.25 * 442381.16 << endl;
cout << 1.0 / 250000 << endl;
```

sljedeći ispis:

```
7.665857e+10          (tj. 7.665857·1010)
4e-06                  (tj. 4·10-6)
```

Realnu aritmetiku ćemo ilustrirati još jednim kratkim programom, koji učitava iznos novca izražen u engleskim funtama, zatim kursnu razliku engleske funte i BH konvertibilne marke, i na izlazu daje odgovarajući iznos izražen u konvertibilnim markama.

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

int main() {
    double funte, kursna_razlika;
    cout << "Unesi iznos novca u funtama: ";
    cin >> funte;
    cout << "Unesi kursnu razliku: ";
    cin >> kursna_razlika;
    double marke = funte * kursna_razlika;
    cout << funte << " funti = " << marke << " maraka\n";
}
```

Do sada smo bili ograničeni samo na četiri osnovne računske operacije (“+”, “-”, “*” i “/”). Uključivanjem zaglavljiva biblioteke “cmath” u program (u ranijim standardima jezika C++ kao i u jeziku C, zaglavljivo ove biblioteke se zvalo “math.h”), dobijamo mogućnost korištenja velikog broja raznih matematičkih funkcija, kao što su kvadratni korijen, trigonometrijske funkcije, itd. Ovdje ćemo navesti neke najvažnije funkcije iz ove biblioteke:

<code>fabs (x)</code>	Apsolutna vrijednost od x
<code>pow (x, y)</code>	Stepenovanje, x^y
<code>sqrt (x)</code>	Kvadratni korijen od x
<code>exp (x)</code>	Specijalni slučaj stepenovanja, e^x
<code>log (x)</code>	Logaritam po bazi e od x , $\ln x$
<code>log10 (x)</code>	Logaritam po bazi 10 od x , $\log x$
<code>sin (x)</code>	Sinus od x ($\sin x$), argument je u radijanima

$\cos(x)$	Kosinus od x ($\cos x$), argument je u radijanima
$\tan(x)$	Tangens od x ($\operatorname{tg} x$), argument je u radijanima
$\operatorname{asin}(x)$	Glavna vrijednost arkus sinusa od x ($\operatorname{arcsin} x$), rezultat je u radijanima
$\operatorname{acos}(x)$	Glavna vrijednost arkus kosinusa od x ($\operatorname{arccos} x$), rezultat je u radijanima
$\operatorname{atan}(x)$	Glavna vrijednost arkus tangensa od x ($\operatorname{arctg} x$), rezultat je u radijanima
$\sinh(x)$	Hiperbolni sinus od x , sh x
$\cosh(x)$	Hiperbolni kosinus od x , ch x
$\tanh(x)$	Hiperbolni tangens od x , th x

Sve navedene funkcije vraćaju kao rezultat realan broj. Jedno karakteristično svojstvo standarda ISO C++98 jezika C++ je da je svaka od ovih funkcija izvedena u *tri verzije*, koje daju rezultat tipa “**float**”, “**double**” ili “**long double**”, u zavisnosti da li je njihov argument tipa “**float**”, “**double**” ili “**long double**” respektivno. Tako će “`sin(a)`” dati rezultat tipa “**float**” ukoliko je promjenljiva “`a`” tipa “**float**”, rezultat tipa “**double**” ukoliko je promjenljiva “`a`” tipa “**double**”, itd. Na taj način preciznost računanja funkcije ovisi od preciznosti njenog argumenta (podsjetimo se da su promjenljive tipa “**float**” manje precizne od promjenljivih tipa “**double**”). U ranijim standardima jezika C++ nije bilo ovako, nego su sve realne funkcije vraćale rezultat tipa “**double**”.

Ova novouvedena osobina standarda ISO C++98 pruža izvjesne prednosti, ali je dovela i do izvjesnih komplikacija. Naime, ukoliko se kao argument neke od ovih funkcija upotrijebi *cjelobrojna vrijednost*, dolazi do *nejasnoće* odnosno *dvosmislenosti* (engl. *ambiguity*) koja uzrokuje grešku pri prevodenju. Naime, jasno je da se ova cjelobrojna vrijednost treba pretvoriti u realnu vrijednost prije proslijđivanja funkciji, ali je nejasno u *kakvu* realnu vrijednost (tj. da li tipa “**float**”, “**double**” ili “**long double**”). Naime, od tipa odabrane pretvorbe ovisi i tačnost sa kojim će vrijednost funkcije biti izračunata. Ovaj problem se rješava tako što se izvrši *eksplicitna pretvorba* cjelobrojnog argumenta u neki realni tip. Na primjer, ukoliko je “`a`” cjelobrojna promjenljiva, naredba poput

```
cout << sqrt(a);
```

doveće do prijave greške (nejasan poziv). Ovaj problem rješavamo eksplisitnom pretvorbom. Drugim riječima, zavisno od željene tačnosti računanja, upotrijebićemo neku od sljedeće tri konstrukcije:

```
cout << sqrt((float)a);
cout << sqrt((double)a);
cout << sqrt((long double)a);
```

Primijetimo da su se prve dvije konstrukcije mogle napisati i upotrebom funkcijске notacije *type-casting* operatora, na primjer:

```
cout << sqrt(float(a));
cout << sqrt(double(a));
```

Međutim, u trećem slučaju, funkcijска notacija se ne bi mogla koristiti, jer se ime tipa “**long double**” sastoji od *dvije riječi*.

Bitno je naglasiti da se za klasično napisane realne brojeve (tj. realne brojne konstante) smatra po konvenciji da su tipa “**double**”. Tako se pri pozivu funkcije “`sqrt`” u primjeru

```
cout << sqrt(3.42);
```

ne javlja nejasnoća, s obzirom da se smatra da je broj “`3.42`” tipa “**double**”, tako da se poziva “**double**”

verzija funkcije "sqrt". Ova konvencija se može izmijeniti dodavanjem sufiksa "F" ili "L" na broj, tako da je broj "3.42F" tipa "float", dok je broj "3.42L" tipa "long double". S druge strane, poziv funkcije "sqrt" u primjeru

```
cout << sqrt(3);
```

je *nejasan*, s obzirom da se cijelobrojna vrijednost "3" treba pretvoriti u *neki* realni tip, ali je nejasno *koji*. Ovaj primjer treba jasno razlikovati od primjera

```
cout << sqrt(3.);
```

koji je savršeno jasan (poziva se "double" verzija funkcije "sqrt"). Treba napomenuti da ovaj problem nejasnoće nije bio prisutan u starijim standardima jezika C++ (niti u jeziku C), s obzirom da su, prema ranijim standardima, sve realne funkcije postojale samo u "double" verziji (preciznije rečeno, problem je uveden u novoj verziji biblioteke "cmath" – stara verzija biblioteke sa zaglavljem "math.h" nije posjedovala ovaj problem).

Pored gore pomenutih matematičkih funkcija, postoji i funkcija "abs", koja se ne nalazi u biblioteci "cmath", nego u biblioteci "cstdlib". Ova funkcija je jako srodna funkciji "fabs", ali se ona primjenjuje na argumente *cjelobrojnog tipa*, i tada kao rezultat vraća također *cjelobrojni tip*. Na primjer, ukoliko je "a" cjelobrojnog tipa, tada je "abs(a)" sasvim nedvosmislen izraz, čija je vrijednost također cjelobrojnog tipa. Poznavanje ove činjenice može biti značajno u nekim slučajevima. Na primjer, ukoliko su "a" i "b" cjelobrojne promjenljive, operator "/" u izrazu "abs(a) / b" obavlja *cjelobrojno dijeljenje*, s obzirom da su oba njegova operanda cjelobrojnog tipa! S druge strane, sve realne funkcije (uključujući i funkciju "fabs") uvijek vraćaju rezultat realnog tipa, čak i u slučajevima kada se rezultat može iskazati u vidu cijelog broja. Na primjer, rezultat izraza "sqrt(9.)" je realnog tipa (realni broj "3."), mada bi se mogao iskazati kao cijeli broj "3".

Ilustraciju formiranja realnih izraza demonstriraćemo na nekoliko primjera iz sljedeće tabele, u kojoj je sa lijeve strane napisan izraz u matematskoj formi, a sa desne strane odgovarajući izraz zapisan u sintaksi jezika C++. Pri tome se podrazumijeva da su sve upotrijebljene promjenljive prethodno deklarirane kao promjenljive realnog tipa:

Matematički zapis:

$$\frac{a^3 - 3}{x^4 + y^4}$$

$$\frac{a \cdot b}{c}$$

$$\frac{a}{b \cdot c}$$

$$\frac{a}{b} \cdot c$$

$$3 \cdot \left\{ 4 + \frac{x}{2 + \frac{1}{x+1}} \cdot [(2x-3):(x-4)] \right\}$$

C++ zapis:

(pow(a, 3) - 3) / (pow(x, 4) + pow(y, 4))

(a*b) / c *ili samo* a*b/c

a / (b * c)

(a/b) * c *ili samo* a/b*c

3 * (4 + x / (2 + 1 / (x + 1)) * ((2 * x - 3) / (x - 4)))

$$\begin{aligned}
 & 1 + \sqrt{\frac{x+1}{x-1}} && 1 + \text{sqrt}((x+1)/(x-1)) \\
 & 1 + \frac{\sqrt{x+1}}{x-1} && 1 + \text{sqrt}(x+1)/(x-1) \\
 & 2 + \sqrt{3\sqrt{\frac{x}{x+1}} + 2} && 2 + \text{sqrt}(3 * \text{sqrt}(x/(x+1)) + 2) \\
 & \frac{2}{1 + |x + |y - 1|} && 2 / (1 + \text{fabs}(x + \text{fabs}(y - 1)))
 \end{aligned}$$

Primijetimo da u izrazima poput “ $a * b / c$ ” zgrade oko produkta nisu neophodne (ali neće ni smetati), jer se operacije koje su istog prioriteta (npr. množenje i dijeljenje) izvode redom slijeva nadesno.

Kao još jednu ilustraciju upotrebe matematičkih funkcija, napisaćemo program koji nalazi i ispisuje rješenja kvadratne jednačine oblika $ax^2 + bx + c = 0$ (uz prepostavku da su ona realna), pri čemu se koeficijenti a , b i c unose sa tastature. Obratite pažnju na zgrade, koje su neophodne da se definira ispravan redoslijed operacija!

```

#include <iostream>
#include <cmath>

using namespace std;

int main() {
    double a, b, c;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;
    double d = b * b - 4 * a * c;
    double x1 = (-b - sqrt(d)) / (2 * a), x2 = (-b + sqrt(d)) / (2 * a);
    cout "x1 = " << x1 << endl << "x2 = " << x2 << endl;
    return 0;
}
  
```

Mogući scenario izvršavanja ovog programa je sljedeći:

```

a = 2
b = 10
c = 12
x1 = -3
x2 = -2
  
```

Interesantno je zapitati se šta će se dogoditi ukoliko unesemo takve koeficijente za koje ne postoji realno rješenje (u tom slučaju će se funkciji za računanje kvadratnog korijena “`sqrt`” kao argument proslijediti negativan broj), ili ukoliko se za vrijednost koeficijenta a unese nula (u tom slučaju dolazi do dijeljenja nulom). Standard jezika C++ ne specificira precizno šta se treba dogoditi u takvim slučajevima. Starije verzije kompjerala za C++ uglavnom su u takvim slučajevima izazivale prekid rada programa (uz

prijavu neke greške), pri čemu postoje neki načini (ovisni od konkretnog kompjajlera) za presretanje ovakvih grešaka. Međutim, noviji C++ kompjajleri obično u takvim slučajevima generiraju kao rezultat neke specijalne objekte, za koje se također smatra da su realnog tipa. Na primjer, pri dijeljenju nulom, kao i u drugim slučajevima kada je rezultat beskonačan po modulu, kao rezultat operacije se generira objekat “ ∞ ”, odnosno “beskonačno” (engl. *infinity*), dok se, u slučaju dijeljenja nule nulom, poziva funkcije nad argumentom izvan njenog domena (npr. funkcije “ \log ” nad negativnim argumentom) i sličnim operacijama čiji rezultat nije definiran generiraju kao rezultat objekat koji se naziva *ne-broj* (engl. *not-a-number* ili *NaN*). Nad objektom “ ∞ ” definirane su neke operacije (npr. definirano je da je $1/\infty = 0$), dok je rezultat bilo koje operacije čiji je makar jedan argument ne-broj ponovo ne-broj (npr. ako promjenljiva “*a*” sadrži ne-broj, rezultat izraza “*a + b*” je ne-broj, šta god sadržavala promjenljiva “*b*”). Prilikom ispisa, objekat “ ∞ ” se obično ispisuje kao tekst “`INF`”, dok se ne-brojevi obično ispisuju kao tekst “`NAN`” ili “`IND`” (od engl. *indeterminate*).

Ranije smo vidjeli da je prilikom korištenja cjelobrojnih tipova glavni problem *prekoračenje*. Realni tipovi imaju sasvim dovoljan opseg da se kod njih problem prekoračenja uglavnom ne javlja. Međutim, kod realnih tipova je prisutan problem tzv. *gubljenja cifara*. Razumijevanje izlaganja koja slijede zahtijeva od čitatelja ili čitateljke izvjesno poznavanje načina kako se realni brojevi zapisuju u računarskoj memoriji. Naime, realne vrijednosti u jeziku C++ se zapisuju u memoriji u obliku tzv. *pokretnog zareza* (engl. *floating point*, odakle i potiče ime tipa “`float`”). Ovaj zapis se zasniva na činjenici da se svaki realan broj x različit od 0 može na jedinstven način zapisati u obliku $\pm m \cdot 2^e$, gdje je e cijeli broj nazvan (binarni) eksponent, a m realan broj u opsegu $0.5 \leq m < 1$ nazvan *mantisa*. Tada se broj x pamti u memoriji kao par (m, e), pri čemu se broj m pamti tako da se pamti određen unaprijed specificiran broj cifara iza zareza koji nastaju kada se m pretvori u binarni zapis. Bitno je primjetiti da na taj način može doći do gubitka tačnosti, s obzirom da mantisa može imati i beskonačno mnogo cifara iza zareza.

Tipovi “`float`”, “`double`” i “`long double`” zapravo se razlikuju upravo po tome koliko se bita rezervira u memoriji za pamćenje eksponenta i mantise (jedan bit se uvijek rezervira za znak). Većina implementacija jezika C++ koristi IEEE 754 standard, koji predviđa sljedeće dužine:

<u>Tip:</u>	<u>Eksponent:</u>	<u>Mantisa:</u>	<u>Ukupna dužina promjenljive:</u>
<code>float</code>	8 bita	23 bita	4 bajta
<code>double</code>	11 bita	52 bita	8 bajta
<code>long double</code>	15 bita	64 bita	10 bajta

Ove podatke možete i sami provjeriti na Vašoj varijanti C++ kompjajlera pomoću operatora “`sizeof`”. Tako će naredba

```
cout << sizeof(long double);
```

vrlo vjerovatno (tj. na većini današnjih kompjajlera) ispisati broj 10.

U praksi, ovi podaci zapravo određuju maksimalni opseg i preciznost (tj. maksimalni broj tačnih cifara) pojedinih tipova. Za IEEE 754 standard imamo:

<u>Tip:</u>	<u>Opseg (po modulu):</u>	<u>Broj tačnih cifara:</u>
<code>float</code>	$1.5 \cdot 10^{-45} - 1.7 \cdot 10^{38}$	7 – 8
<code>double</code>	$5.0 \cdot 10^{-324} - 1.7 \cdot 10^{308}$	15 – 16
<code>long double</code>	$3.4 \cdot 10^{-4932} - 1.1 \cdot 10^{4932}$	19 – 20

U slučaju prekoračenja rezultata računanja po modulu, kao rezultat se obično generira objekat “ ∞ ”, dok se u slučaju podbačaja rezultata po modulu ispod minimalno dozvoljene vrijednosti kao rezultat obično generira nula. Međutim, vidimo da je opseg realnih tipova dovoljno velik za praktično sve primjene, tako da nas pri radu sa njima problem prekoračenja ne treba da zabrinjava.

Razmotrimo sada problem *gubljenja cifara*, koji smo ranije najavili. Ovaj problem se javlja kada pokušamo da u promjenljivu stavimo više tačnih cifara nego što njen tip dopušta. Pogledajmo sljedeću sekvencu naredbi:

```
float a = 327653225, b = 327653213;
cout << a - b;
```

Ova sekvenca ispisuje kao rezultat nulu, iako bi rezultat trebao da bude broj “12”. Brojevi smješteni u promjenljive tipa “**float**” su jednostavno imali “previše cifara” za taj tip, tako da je kompjuter nakon njihovog pretvaranja u binarni sistem izvršio zaokruživanje, čime su one postale jednake. Da se uvjerimo u to, možemo napisati sljedeću naredbu:

```
cout << a << " " << b;
```

Generalan zaključak je da problemi nastaju kad god se oduzimaju dva velika, a međusobno bliska broja. Rezultat je tada ili pogrešan, ili sadrži znatno manji broj tačnih cifara u odnosu na brojeve koji se oduzimaju. Sličan problem se javlja ako pokušamo da saberemo veoma veliki i veoma mali broj. Na primjer:

```
float a = 327653225;
float b = a + 1;
```

Nakon ove sekvence naredbi, varijable “a” i “b” će biti iste, iako ne bi trebale da budu. Tip “**float**” ne može da zapamti dovoljno cifara da bi razlika od jedne jedinice mogla da bude registrirana u broju koji ima devet cifara. Oba problema biće riješena ako umjesto tipa “**float**” koristimo tip “**double**”, ali ni on nije svemoguć: sa brojevima koji imaju više od 16 tačnih cifara ne možemo očekivati tačan račun. Ipak, može se uzeti da tip “**double**” uglavnom pokriva područje praktičnih primjena. Zbog toga, ako memoriski zahtjevi nisu veoma strogi, tip “**float**” treba izbjegavati i umjesto njega koristiti tip “**double**”.

Ni korištenje tipa “**double**” ne rješava nas svih problema. Problem leži u tome što se mantisa broja pamti sa konačnim brojem bita (binarnih cifara), a nerijetko se dešava da mantisa ima beskonačno mnogo cifara. Naročito je problematično što neki brojevi koji u dekadnom brojnom sistemu imaju konačno mnogo decimala, dobijaju beskonačno mnogo decimala nakon pretvaranja u binarni brojni sistem. Na primjer, vrijedi

$$0.2 = (0.001100110011\dots)_2$$

Kao posljedica ovoga, slijedi da računar ne može tačno da zapamti niti tako jednostavan broj kao što je “0.2”. Na primjer, razmotrimo sljedeći programski isječak:

```
double a(0.3), b(0.4), c(0.5);
cout << a * a + b * b - c * c;
```

Mada bi rezultat ovog programa po svakoj logici trebao da bude nula, on zbog opisanog problema neće biti nula, već neki izuzetno mali broj. Nešto je jasniji sljedeći primjer, koji također veoma lijepo ilustrira problem gubljenja tačnosti:

```
double a(123.0), b(321.0);
cout << a * b / b - a / b * b;
```

Ovo je još jedan primjer koji bi trebao ispisati nulu, ali neće. U čemu je problem? Izrazi “ $a * b / b$ ” i “ $a / b * b$ ” nakon uvrštavanja konkretnih vrijednosti “ a ” i “ b ” postaju “ $123.0 * 321.0 / 321.0$ ” i “ $123.0 / 321.0 * 321.0$ ”. U prvom slučaju, prvo se računa produkt “ $123.0 * 321.0$ ” koji daje rezultat “ 39483.0 ”, a koji se zatim dijeli sa “ 321.0 ”, što na kraju daje tačan rezultat “ 123.0 ”. Međutim, u drugom slučaju, prvo se računa količnik “ $123.0 / 321.0$ ” koji ima beskonačno mnogo decimala (“ $0.38317757009\dots$ ”). Ovaj rezultat će svakako morati biti zaokružen na određen broj decimala. Nakon množenja zaokruženog rezultata sa “ 321.0 ”, neće se dobiti tačno “ 123.0 ” kao rezultat, što na kraju dovodi do greške u konačnom rezultatu.

Primjeri poput navedenih, čest su uzrok veoma ozbiljnih i frustrirajućih problema u programima koji rade sa realnim tipovima. Kasnije ćemo posebno govoriti o tome kakve posljedice ova činjenica može imati na problematiku poređenja realnih vrijednosti.

7. Kompleksni tipovi

Za pisanje matematičkih ili inžinjerski orijentiranih programa često je potrebno posjedovati mogućnost rada sa *kompleksnim brojevima*. Dugo vremena, standard jezika C++ nije predviđao da C++ mora poznavati kompleksne brojeve. Međutim, kao što ćemo kasnije vidjeti, C++ je fleksibilan jezik koji omogućava naknadno definiranje novih tipova podataka (uključujući i tzv. korisničke tipove podataka, koje kreira sam korisnik, odnosno programer). Stoga su mnoge implementacije kompjajlera za C++, poput AT&T C++ i Turbo C++ kompjajlera, nudili biblioteke (nestandardne) koje definiraju kompleksne tipove podataka i rad sa njima. Kako su ove biblioteke bile nestandardne, rad sa njima se razlikovao od implementacije do implementacije. Međutim, standard ISO C++98 predviđa postojanje *standardne* biblioteke “`complex`”, čijim uključivanjem dobijamo mogućnost rada sa kompleksnim brojevima na sasvim precizno definiran način.

Kompleksni brojevi definiraju se pomoću deklaracije koja sadrži riječ “`complex`”. Međutim, kompleksni brojevi spadaju u tzv. *izvedene tipove* (engl. *derived types*). Naime, u matematici je kompleksni broj formalno definiran kao par realnih brojeva. Međutim, pošto u jeziku C++ imamo nekoliko tipova za opis realnih brojeva (“`float`”, “`double`” i “`long double`”), kompleksne brojeve je moguće izvesti iz svakog od ovih tipova. Sintaksa za deklaraciju kompleksnih brojeva je

```
complex<osnovni_tip> popis_promjenljivih;
```

gdje je *osnovni_tip* tip iz kojeg izvodimo kompleksni tip. Na primjer, deklaracijom oblika

```
complex<double> z;
```

deklariramo kompleksnu promjenljivu “*z*” čiji su realni i imaginarni dio tipa “`double`”. Kompleksni tipovi se mogu izvesti čak i iz nekog od cijelobrojnih tipova. Na primjer,

```
complex<int> gauss;
```

deklarira kompleksnu promjenljivu “*gauss*” čiji realni i imaginarni brojevi mogu biti samo cijelobrojne vrijednosti tipa “`int`”. Inače, kompleksni brojevi čiji su realni i imaginarni dio cijeli brojevi imaju veliki značaj u teoriji brojeva i algebri, gdje se nazivaju *Gaussovi cijeli brojevi* (to je i bila motivacija da ovu promjenljivu nazovemo “*gauss*”).

Bitno je primijetiti da “`complex`” nije ključna riječ, za razliku od npr. “`int`” i “`double`”. Naime, “`complex`” je *predefinirana riječ*, definirana u istoimenoj biblioteci, za razliku od “`int`” i “`double`” koje čine srž samog jezika C++. Drugim riječima, ukoliko nismo u program uključili biblioteku “`complex`”, riječ “`complex`” možemo u programu koristiti za bilo koju drugu svrhu (npr. možemo neku promjenljivu nazvati tim imenom). Ne treba posebno napominjati da to ipak nije dobra praksa.

Poput svih drugih promjenljivih, i kompleksne promjenljive se mogu inicijalizirati prilikom deklaracije. Kompleksne promjenljive se tipično inicijaliziraju parom vrijednosti u zagradama, koje predstavljaju realni odnosno imaginarni dio broja. Na primjer, deklaracija

```
complex<double> z1(2, 3.5);
```

deklarira kompleksnu promjenljivu “*z1*” i inicijalizira je na kompleksnu vrijednost “ $(2, 3.5)$ ”, što bi se u algebarskoj notaciji moglo zapisati i kao “ $2 + 3.5i$ ”. Pored toga, moguće je inicijalizirati kompleksnu promjenljivu realnim izrazom odgovarajućeg realnog tipa iz kojeg je razmatrani kompleksni tip izведен

(npr. promjenljiva tipa “`complex<double>`” može se inicijalizirati izrazom tipa “`double`”, npr. realnim brojem), kao i kompleksnim izrazom istog tipa (npr. drugom kompleksnom promjenljivom istog tipa). Tako su, na primjer, uz pretpostavku da je promjenljiva “`z1`” deklarirana pomoću prethodne deklaracije, također legalne i sljedeće deklaracije:

```
complex<double> z2(12.7), z3(z1);
```

U ovakvim slučajevima, inicijalizaciju je moguće izvršiti i pomoću znaka “`=`”, na primjer:

```
complex<double> z2 = 12.7, z3 = z1;
```

Međutim, u slučaju kompleksnog tipa inicijalizacija navođenjem vrijednosti u zagradama i pomoću znaka “`=`” nije potpuno identična, nego je inicijalizacija navođenjem vrijednosti u zagradama nešto generalnija. Naime, pri tom tipu inicijalizacije moguće je mijesati objekte tipova “`complex<float>`”, “`complex<double>`” i “`complex<long double>`” u smislu da je sasvim dozvoljeno promjenljivu tipa “`complex<double>`” inicijalizirati izrazom tipa “`complex<float>`” i obrnuto (pri čemu u prvom slučaju dolazi do automatske promocije, a u drugom slučaju do reduciranja tačnosti). Međutim, pri inicijalizaciji pomoću znaka “`=`” podržana je samo promocija ali ne i odsjecanje tačnosti, tako da npr. pomoću znaka “`=`” nije moguće izvršiti inicijalizaciju promjenljive tipa “`complex<float>`” izrazom tipa “`complex<double>`” (postoje neki praktični razlozi zbog kojeg je ovo ograničenje uvedeno).

Sa aspekta inicijalizacije, kompleksni tipovi izvedeni iz realnih tipova i kompleksni tipovi izvedeni iz cjelobrojnih tipova ponašaju se kao potpuno različiti tipovi, tako da je npr. nemoguće inicijalizirati promjenljivu tipa “`complex<double>`” izrazom (npr. drugom promjenljivom) tipa “`complex<int>`”. Slično vrijedi i za kompleksne tipove izvedene iz različitih cjelobrojnih tipova (npr. tipovi “`complex<int>`” i “`complex<long int>`” su nesaglasni po pitanju inicijalizacije. Mada ova ograničenja izgledaju posve nepotrebna, ona su posljedica nekih tehničkih aspekata vezanih za detalje kako su kompleksni tipovi uopće implementirani. Srećom, potreba za ovakvim nesaglasnim inicijalizacijama javlja se veoma rijetko. Nešto kasnije u ovom poglavlju biće ilustrirani neki načini da se ova ograničenja zaobiđu, ukoliko je to zaista potrebno.

Treba još napomenuti da se kompleksne promjenljive, u slučaju da inicijalizacija nije eksplisitno navedena, *automatski inicijaliziraju na nulu*. To znači da će promjenljive “`z`” i “`gauss`” iz ranijih primjera biti automatski inicijalizirane na (kompleksnu) nulu, odnosno na kompleksan broj “`(0,0)`”. Podsjetimo se da se ovakva automatska inicijalizacija ne dešava u slučaju promjenljivih tipa “`int`”, “`double`”, itd. Ovo je posljedica činjenice da je kompleksni tip korisnički definiran izvedeni tip, za koje je moguće definirati postupak automatske inicijalizacije (što je učinjeno u implementaciji biblioteke “`complex`”), što na žalost nije moguće definirati za ugrađene tipove, kao što su “`int`” itd.

Vidjeli smo kako je moguće *inicijalizirati* kompleksnu promjenljivu na neku kompleksnu vrijednost. Međutim, često je potrebno nekoj već deklariranoj kompleksnoj promjenljivoj *dodijeliti* neku kompleksnu vrijednost. Ovo možemo uraditi pomoću konstrukcije

```
complex<osnovni_tip> (realni_dio, osnovni_tip)
```

koja kreira kompleksni broj odgovarajućeg tipa, sa zadanim realnim i imaginarnim dijelom, koji predstavljaju izraze realnog tipa, ili nekog tipa koji se može promovirati u realni tip (npr. cjelobrojnog tipa). Na primjer, za promjenljivu “`z`” iz ranijih primjera, moguće je izvršiti dodjelu

```
z = complex<double>(2.5, 3.12);
```

Sasvim je moguće kompleksnim promjenljivim dodjeljivati realne vrijednosti, pa čak i cjelobrojene

vrijednosti, pri čemu dolazi do automatske konverzije tipa (promocije). Inače, interesantno je da je *dodjeljivanje* podložno mnogo manjim ograničenjima u odnosu na *inicijalizaciju* (ne smijemo zaboraviti da su dodjeljivanje i inicijalizacija dvije različite stvari). Tako je promjenljivoj ma kojeg kompleksnog tipa moguće dodijeliti izraz ma kojeg drugog kompleksnog, realnog ili cjelobrojnog tipa, pri čemu u zavisnosti od slučaja, dolazi ili do promocije, ili do reduciranja tačnosti. Na primjer, moguće je promjenljivoj tipa “`complex<double>`” dodijeliti vrijednost promjenljive “`complex<int>`” bez obzira što slična inicijalizacija nije moguća. Moguća je čak i obrnuta dodjela, pri kojoj će doći do reduciranja tačnosti, u dosta drastičnom obliku, koji podrazumijeva odsjecanje decimala.

Razumije se da je inicijalizaciju neke kompleksne promjenljive moguće izvršiti i konstrukcijom poput navedene, tj. umjesto deklaracije

```
complex<double> z1(2, 3.5);
```

principijelno je moguće izvršiti i deklaraciju poput

```
complex<double> z1 = complex<double>(2, 3.5);
```

Međutim, druga deklaracija, pored toga što je po formi komplikovanija, manje je efikasna od prve. U prvoj deklaraciji, promjenljiva “`z1`” se prilikom kreiranja (stvaranja) inicijalizira na kompleksnu vrijednost “`(2, 3.5)`”. U drugom slučaju, konstrukcija “`complex<double>(2, 3.5)`” kreira pomoćnu kompleksnu vrijednost “`(2, 3.5)`” koja se zatim koristi da se njom inicijalizira promjenljiva “`z1`”. Drugim riječima, u drugoj deklaraciji imamo jednu kreaciju više (pomoćna kompleksna vrijednost i sama promjenljiva), što je neefikasnije od prve varijante. U načelu, dobar kompjajler može prilikom prevođenja optimizirati drugu varijantu tako da se ona svede na prvu varijantu, ali on to nije dužan da učini (to je stvar “inteligencije” upotrijebljenog kompjajlера). Stoga je uvjek bolje koristiti konstrukcije koje su po samoj svojoj prirodi efikasnije, neovisno od upotrijebljenog kompjajlера.

Treba obratiti pažnju na jednu čestu početničku grešku. Izraz čiji je oblik

```
(x, y)
```

gdje su `x` i `y` neki izrazi sintaksno je *ispravan* u jeziku C++, ali *ne predstavlja* kompleksni broj čiji je realni dio `x`, a imaginarni dio `y`. O značenju ovog izraza ćemo govoriti kasnije. Za sada je bitno navesti da tip ovog izraza nije kompleksan tip, već je njegov tip istog tipa kao tip izraza `y`. Stoga je naredba poput

```
z = (2, 3.5);
```

sintaksno *posve ispravna*, ali ne radi ono što bi korisnik mogao očekivati (tj. da izvrši dodjelu kompleksne vrijednosti `(2, 3.5)` promjenljivoj “`z`”). Šta će se dogoditi? Izraz “`(2, 3.5)`” ima izvjesnu *realnu* vrijednost (koja, u konkretnom slučaju, iznosi “`3.5`”), koja će, nakon odgovarajuće promocije, biti dodijeljena promjenljivoj “`z`”. Drugim riječima, vrijednost promjenljive “`z`” nakon ovakve dodjele biće broj “`3.5`”, odnosno, preciznije, kompleksna vrijednost “`(3.5, 0)`”!

Nad kompleksnim brojevima i promjenljivim moguće je obavljati četiri osnovne računske operacije “`+`”, “`-`”, “`*`” i “`/`”. Uz pomoć ovih operatora tvorimo *kompleksne izraze*. Također, za kompleksne promjenljive definirani su i operatori “`+=`”, “`-=`”, “`*=`” i “`/=`”, ali *ne* i operatori “`++`” i “`--`”. Pored toga, gotovo sve matematičke funkcije, poput “`sqrt`”, “`sin`”, “`log`” itd. definirane su i za kompleksne vrijednosti argumenata (ovdje nećemo ulaziti u to šta zapravo predstavlja sinus kompleksnog broja, itd.). Kao dodatak ovim funkcijama, postoje neke funkcije koje su definirane isključivo za kompleksne vrijednosti argumenata. Na ovom mjestu vrijedi spomenuti sljedeće funkcije:

<code>real (z)</code>	Realni dio kompleksnog broja z ; rezultat je <i>realan</i> broj (odnosno, realnog je tipa)
<code>imag (z)</code>	Imaginarni dio kompleksnog broja z ; rezultat je <i>realan</i> broj
<code>abs (z)</code>	Apsolutna vrijednost (modul) kompleksnog broja z ; rezultat je <i>realan</i> broj
<code>arg (z)</code>	Argument kompleksnog broja z (u <i>radijanima</i>); rezultat je <i>realan</i> broj
<code>conj (z)</code>	Konjugovano kompleksna vrijednost kompleksnog broja z

Kompleksni brojevi se mogu ispisivati slanjem na izlazni tok pomoću operatora “`<<`”, pri čemu se ispis vrši u vidu uređenog para “ (x, y) ” a ne u nešto uobičajenije algebarskoj notaciji “ $x + yi$ ”. Također, kompleksne promjenljive se mogu učitavati iz ulaznog toka (npr. sa tastature) koristeći isti format zadavanja kompleksnog broja. Pri tome, kada se očekuje učitavanje kompleksne promjenljive iz ulaznog toka, moguće je unijeti realni ili cijeli broj, koji će biti automatski konvertiran u kompleksni broj.

Iz izloženog slijedi da su, uz prepostavku da su npr. “`a`”, “`b`” i “`c`” promjenljive deklarirane kao promjenljive tipa “`complex<double>`”, sljedeći izrazi posve legalni:

```
a + b / conj (c)
a + complex<double>(3.2, 2.15)
(a + sqrt(b)) / complex<double>(0, 3)) * log(c - complex<double>(2, 1.25))
```

Također, pri izvođenju svih aritmetičkih operacija, dozvoljeno je miješati operande kompleksnog tipa i cjelobrojnog odnosno realnog tipa, pri čemu je poželjno da odgovarajući realni tip bude isti tip iz kojeg je izveden odgovarajući kompleksni tip (u suprotnom, u određenim situacijama može biti problema sa miješanjem tipova, na koje će nas kompjajler upozoriti). Stoga, ukoliko su, pored gore navedenih promjenljivih, deklarirane i realne promjenljive “`x`” i “`y`”, kao i cjelobrojne promjenljive “`p`” i “`q`”, legalni su i izrazi poput sljedećih:

```
a + b * y
a * real(c) - b * imag(c)
p + c - complex<double>(x + y, x - y)
(a + b / x) * (c - complex<double>(p, q)) / p
```

Trebamo se čuvati izraza poput

```
a + (3.2, 2.15)
```

koji, mada su sintaksno ispravni, ne rade ono što bi korisnik mogao očekivati, s obzirom na već spomenuti tretman izraza oblika “ (x, y) ”.

Bez obzira na navedenu fleksibilnost, koja omogućava miješanje tipova u izrazima, zbog izvjesnih tehničkih razloga nije dozvoljeno miješanje različitih kompleksnih tipova u istom izrazu. Na primjer, ukoliko je promjenljiva “`a`” tipa “`complex<double>`” a promjenljiva “`b`” tipa “`complex<float>`”, izraz poput “`a + b`” nije legalan. Ovaj problem možemo riješiti na dva načina. Prvi način je da na osnovu poznavanja realnog i imaginarnog dijela jednog operanda kreiramo odgovarajuću kompleksnu vrijednost čiji je tip saglasan sa drugim operandom, odnosno da napišemo nešto poput

```
a + complex<double>(real(b), imag(b))
```

Sličan trik možemo upotrijebiti ukoliko želimo inicijalizirati neku promjenljivu kompleksnog tipa izvedenog iz realnog tipa nekom promjenljivom kompleksnog tipa izvedenog iz realnog tipa. Na primjer, mada smo vidjeli da je inicijalizacija poput sljedeće zabranjena:

```
complex<int> gauss(3, 2);
complex<double> z(gauss);
```

sljedeća inicijalizacija je sasvim legalna:

```
complex<int> gauss(3, 2);
complex<double> z(real(gauss), imag(gauss));
```

Drugi način za rješavanje problema miješanja kompleksnih tipova u izrazima je da pomoću *type-casting* operatora izvršimo pretvorbu jednog tipa u drugi. Na primjer, možemo pisati izraz poput sljedećeg:

```
a + (complex<double>)b
```

Alternativno, možemo koristiti i funkciju notaciju za pretvorbu tipa:

```
a + complex<double>(b)
```

Prvi način je ipak nešto generalniji, jer sve pretvorbe između kompleksnih tipova nisu definirane (ponovo iz tehničkih razloga). Na primjer, nije definirana pretvorba cjelobrojnih kompleksnih u realne kompleksne tipove (npr. iz tipa “`complex<int>`” u “`complex<double>`”) pomoću *type-casting* operatora. Ukoliko su Vam ova pravila o miješanju tipova isuviše komplikirana i konfuzna (a mora se priznati da jesu), najlakši način da ih izbjegnete je da u istom programu ne miješate različite vrste kompleksnih tipova, ili ako već koristite različite kompleksne tipove, da ih ne miješate unutar istog izraza.

Vrijedi napomenuti još i funkciju “`polar`”. Ova funkcija ima formu

```
polar(r, φ)
```

koja daje kao rezultat kompleksan broj čiji je modul r , a argument $φ$ (u *radijanima*). Ovdje su r i $φ$ neki izrazi realnog tipa. Ova funkcija je veoma korisna za zadavanje kompleksnih brojeva u trigonometrijskom obliku. Na primjer,

```
z = polar(5.12, PI/4);
```

Ovaj primjer podrazumijeva da imamo prethodno definiranu realnu konstantu “`PI`”. Treba znati da funkcija “`polar`” daje rezultat koji je tipa “`complex<double>`”. Ova informacija je korisna zbog prethodno navedenih ograničenja vezanih za miješanje tipova prilikom inicijalizacija i upotrebe aritmetičkih operatora. Na primjer, prethodna dodjela je valjana kojeg god je kompleksnog tipa promjenljiva “`z`”. Međutim, inicijalizacija

```
complex<float> z(polar(5.12, PI/4));
```

je legalna, a inicijalizacija

```
complex<float> z = polar(5.12, PI/4);
```

nije. Naime, već smo rekli da se promjenljiva tipa “`complex<float>`” može inicijalizirati izrazom tipa “`complex<double>`” navođenjem vrijednosti u zagradama, ali ne i pomoću znaka “`=`”.

Sljedeći primjer prikazuje program za rješavanje kvadratne jednačine, uz upotrebu kompleksnih promjenljivih:

```
#include <iostream>
```

```

#include <cmath>
#include <complex>

using namespace std;

int main() {
    double a, b, c;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;
    complex<double> d, x1, x2;
    d = b * b - 4 * a * c;
    x1 = (-b - sqrt(d)) / (2 * a);
    x2 = (-b + sqrt(d)) / (2 * a);
    cout "x1 = " << x1 << "\nx2 = " << x2 << endl;
    return 0;
}

```

Mogući scenario izvršavanja ovog programa je sljedeći (uz takve ulazne podatke za koje rješenja nisu realna, tako da bi se sličan program koji koristi samo realne promjenljive podatke ili “srušio”, ili bi dao *ne-brojeve* kao rezultat):

```

a = 3
b = 6
c = 15
x1 = (-1,-2)
x2 = (-1,2)

```

“Nezgoda” ovakvog rješenja je što se kompleksne promjenljive uvijek ispisuju kao *uređeni parovi*, pa čak i kad je imaginarni dio jednak nuli. Ovo možemo vidjeti u sljedećem scenariju:

```

a = 2
b = 10
c = 12
x1 = (-3,0)
x2 = (-2,0)

```

Ove probleme ćemo moći riješiti tek kada upoznamo naredbu “**if**” koja omogućava da se izračunavanje obavlja različitim postupcima ovisno od toga da li je neki uvjet ispunjen ili nije (u našem slučaju da li je diskriminanta jednačine pozitivna ili negativna).

U ovom programu je interesantno primijetiti da je promjenljiva “**d**”, koja čuva diskriminantu kvadratne jednačine, deklarirana kao kompleksna, bez obzira što znamo da je diskriminanta realan broj. Ukoliko ovo ne učinimo, program neće raditi ispravno. Naime, problem nastaje kod funkcije “**sqrt**”. Ova funkcija, poput gotovo svih matematičkih funkcija, daje rezultat *onog tipa kojeg je tipa njen argument*. Tako, ukoliko je njen argument tipa “**double**”, rezultat bi također trebao da bude tipa “**double**”. Slijedi

da funkcija "sqrt" može kao rezultat dati kompleksan broj samo ukoliko je njen argument kompleksnog tipa. U suprotnom se prosto smatra da je vrijednost funkcije "sqrt" za negativne realne argumente *nedefinirana* (bez obzira što se rezultat smješta u promjenljivu kompleksnog tipa). Ova situacija je analogna već ranije opisanoj situaciji u kojoj je npr. produkt dvije promjenljive tipa "**short int**" i sam tipa "**short int**", bez obzira što se rezultat smješta npr. u promjenljivu tipa "**long int**". Na funkciju "sqrt" sa *realnim argumentom* (i *realnim rezultatom*) i funkciju "sqrt" sa *kompleksnim argumentom* (i *kompleksnim rezultatom*) treba gledati kao na *dvije posve različite funkcije*, koje se slučajno *isto zovu*. Naime, i u matematici je poznato da neka funkcija nije određena samo preslikavanjem koje obavlja, već i svojim *domenom* i *kodomenu* (npr. funkcija $x \rightarrow x^2$ definirana za operande x iz skupa \mathbf{N} i funkcija $x \rightarrow x^2$ definirana za operande x iz skupa \mathbf{R} *nisu ista funkcija*). Stogo rečeno, čak i za slučaj realnih operanada, postoje *tri verzije* funkcije "sqrt" (za argumente tipa "**float**", "**double**" i "**long double**"). Pojava da može postojati više funkcija istog imena za različite tipove argumenata (koje mogu raditi čak i posve različite stvari ovisno od tipa argumenta) naziva se *preklapanje* (ili *preopterećivanje*) *funkcija* (engl. *function overloading*). O ovoj pojavi ćemo detaljnije govoriti kasnije, kada se upoznamo sa načinom definiranja vlastitih funkcija.

Ukoliko ipak želimo da promjenljiva "d" bude tipa "**double**", što je s obzirom na prirodu njenog sadržaja mnogo logičnije, računanje kompleksne vrijednosti korijena možemo forsirati tako što ćemo prilikom računanja korijena (tj. poziva funkcije "sqrt") eksplicitno izvršiti konverziju promjenljive "d" u kompleksni tip pomoću *type-casting* operatora, tj. pisanjem naredbi poput

```
x1 = (-b - sqrt((complex<double>) d)) / (2 * a);
```

odnosno, u funkcijskoj notaciji,

```
x1 = (-b - sqrt(complex<double>(d))) / (2 * a);
```

Principijelno, promjenljive "a", "b" i "c" također možemo deklarirati kao kompleksne promjenljive, npr. tipa "*complex<double>*". To će nam omogućiti da možemo rješavati i kvadratne jednačine čiji su koeficijenti kompleksni brojevi (ne zaboravimo da kompleksne brojeve unosimo kao uredene parove). Na primjer, ako izvršimo izmjene u prethodnom programu da sve promjenljive postanu tipa "*complex<double>*", moći ćemo riješiti jednačine poput $(2 + 3i)x^2 - 5x + 7i = 0$ (probajte riješiti ovu jednačinu "pješke", vidjećete da nije baš jednostavno):

```
a = (2,3)
b = -5
c = (0,7)
x1 = (0.912023,-2.01861)
x2 = (-0.142792,0.864761)
```

Treba ipak napomenuti da nije nimalo mudro deklarirati kao kompleksne one promjenljive koje to zaista ne moraju biti. Naime, kompleksne promjenljive troše više memorije, rad sa njima je daleko sporiji nego rad sa realnim promjenljivim, i što je najvažnije, pojedine operacije sasvim uobičajene za realne brojeve (npr. poređenje, koje ćemo uskoro upoznati) nemaju smisla za kompleksne brojeve.

8. Znakovni tipovi

Kada smo govorili o cjelobrojnim tipovima podataka, rekli smo da u ove tipove također spada i tip “**char**”. Promjenljive ovog tipa “**char**” uvijek zauzimaju tačno 1 bajt. Pod uvjetom da se koristi računarska arhitektura kod koje bajt ima 8 bita (zvuči nevjerovatno, ali postoje i računarske arhitekture kod kojih ovo nije slučaj), dozvoljeni opseg vrijednosti koje se mogu smjestiti u promjenljivu tipa “**char**” (ukoliko ne želimo da dođe do prekoračenja) iznosi od -128 do +127, odnosno od 0 do 255 ako koristimo i prefiks “**unsigned**”. Prema tome, sljedeće konstrukcije su sasvim legalne:

```
char a(30), b, c;  
b = 20;  
c = a + b;
```

Međutim, promjenljive tipa “**char**”, imaju nešto što ih bitno razlikuje od ostalih cjelobrojnih promjenljivih: one se *drugacije ponašaju* prilikom *slanja na izlazni tok* (npr. ispisa na ekran) odnosno *čitanja iz ulaznog toka* (npr. *unosa sa tastature*). Na primjer, isprobajmo sljedeći program:

```
#include <iostream>  
  
using namespace std;  
  
int main() {  
    char broj(65);  
    cout << broj << endl;  
    return 0;  
}
```

Iako očekujemo da će ovaj program ispisati broj “65”, on će umjesto toga na većini današnjih računara ispisati slovo “A”. Da bismo ovo razjasnili, moramo se malo osvrnuti na to kako se *znakovi* (slova, itd.) čuvaju u računarskoj memoriji. Kako je poznato da se u računarskoj memoriji mogu pamtitи samo *brojevi* (i to binarni), dogovoreno je da se svi znakovi u memoriji čuvaju pomoću odgovarajuće *brojne šifre* ili *kôda*. Danas najviše korištena šifra (kôd) je tzv. ASCII kôd, koji predviđa šifre prema sljedećoj tabeli (šifre su navedene u zagradi iza znaka):

<i>razmak</i> (32)	!	(33)	"	(34)	#	(35)	\$	(36)	%	(37)
& (38)	'	(39)	((40))	(41)	*	(42)	+	(43)
,	-	(44)	.	(45)	.	(46)	/	(47)	0	(48)
2 (50)	3	(51)	4	(52)	5	(53)	6	(54)	7	(55)
8 (56)	9	(57)	:	(58)	;	(59)	<	(60)	=	(61)
> (62)	?	(63)	@	(64)	A	(65)	B	(66)	C	(67)
D (68)	E	(69)	F	(70)	G	(71)	H	(72)	I	(73)
J (74)	K	(75)	L	(76)	M	(77)	N	(78)	O	(79)
P (80)	Q	(81)	R	(82)	S	(83)	T	(84)	U	(85)
V (86)	W	(87)	X	(88)	Y	(89)	Z	(90)	[(91)
\ (92)]	(93)	^	(94)	_	(95)	`	(96)	a	(97)
b (98)	c	(99)	d	(100)	e	(101)	f	(102)	g	(103)
h (104)	i	(105)	j	(106)	k	(107)	l	(108)	m	(109)
n (110)	o	(111)	p	(112)	q	(113)	r	(114)	s	(115)
t (116)	u	(117)	v	(118)	w	(119)	x	(120)	y	(121)
z (122)	{	(123)		(124)	}	(125)	~	(126)		

Sada možemo razjasniti neobični ispis koji je demonstriran prethodnim programom. Prilikom ispisa

promjenljivih (i općenito izraza) tipa “**char**”, umjesto njene *brojne vrijednosti*, ispisuje se *znak čija je šifra jednaka toj vrijednosti*. Vidimo da šifri “65” odgovara znak “A”, što objašnjava prikazani ispis.

Nema potrebe za eksplicitnim poznavanjem šifri pojedinih znakova. U jeziku C++ je definirano da umjesto navođenja šifre nekog znaka, možemo taj znak pisati između dva apostrofa (npr. 'A'). Dakle, apostrof predstavlja “šifru od”. Sljedeći program je, prema tome, mnogo jasniji:

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

int main() {
    char znak('A');
    cout << znak << endl;
    return 0;
}
```

U ovom programu smo namjerno promijenili ime promjenljive u “*znak*”, da izbjegnemo zabunu. Treba naglasiti da podaci 'A' i 65 ipak nisu potpuni sinonimi. Njihova je *vrijednost* ista, ali je tip podatka 'A' tip “**char**”, dok je tip podatka 65 tip “**int**”. Kako se svi podaci tipa “**char**” ispisuju u formi *znakova* (engl. *characters*), to će naredba

```
cout << 'A';
```

ispisati slovo “A”, a ne broj 65.

Promjenljive tipa “**char**” obično se koriste da čuvaju jednu znakovnu vrijednost, po čemu je tip “**char**” i dobio ime. Promjenljive tipa “**char**” nazivamo *znakovne promjenljive* (engl. *character variables*). Primjeri smislenih znakovnih deklaracija su:

```
char slovo;
char inicial, ch1, ch2;
```

Treba izbjegavati direktnu dodjelu brojnih vrijednosti promjenljivim tipa “**char**”, mada je takva dodjela posve legalna (preciznije, podržane su automatske konverzije ostalih cjelobrojnih tipova u tip “**char**”, i obrnuto). Njima se obično dodjeljuje vrijednost pomoću pridruživanja poput:

```
ch1 = 'a';
ch2 = '*';
ch1 = ch2;
```

Znak između dva apostrofa naziva se i *znakovna konstanta*. Primjeri ispravnih znakovnih konstanti su (pazite: i razmak je znak):

```
'A'      '7'      '+'
' '      '?'
```

Izuzeći nastaju kada treba napisati neke znakove koji imaju neko specijalno značenje između apostrofa, npr. sam apostrof. Znakovna konstanta koja predstavlja *apostrof* piše se ovako:

```
'\''
```

dok se znakovna konstanta koja predstavlja znak “\” (naopaka kosa crta, engl. *backslash*) piše ovako:

```
'\\\''
```

Dakle, naopaka kosa crta se *udvaja*. Isto pravilo vrijedi i unutar stringova (tj. unutar teksta između dva navodnika). Razlog za ovo je što naopaka kosa crta ima specijalno značenje sa ciljem da označi neke specijalne akcije unutar stringova. Tako, vidjeli smo ranije da oznaka “\n” označava *prelaz u novi red*. Ovakvih specijalnih akcija ima još. Na primjer, “\b” označava *pomjeranje kurzora za jedno mjesto ulijevo*, dok “\r” označava *pomjeranje kurzora na početak reda*. Zbog toga, znak “\” često nazivamo znakom “bijega” (engl. *escape character*). Pošto kompjajler ne može znati da li u slučaju kada napišemo znak “\” znak iza njega predstavlja oznaku akcije ili normalni znak, usvojena je konvencija da se znak “\” udvaja u slučaju da se treba tretirati kao takav, a ne u kontekstu neke specijalne akcije (u tom slučaju, “specijalna akcija” naredena drugom pojmom znaka “\” je upravo tretiranje ovog znaka kao takvog, a ne kao znaka za “bijeg”). Na primjer, ukoliko želimo da ispišemo sljedeći tekst

Ta igra se nalazi u C:\DOS\GAMES folderu

ispravna naredba kojom ćemo ispisati ovaj tekst glasi:

```
cout << "Ta igra se nalazi u C:\\DOS\\\\GAMES folderu\\n";
```

a ne

```
cout << "Ta igra se nalazi u C:\\DOS\\GAMES folderu\\n";
```

U drugom slučaju, kompjajler će smatrati da znakovi “\” predstavljaju znakove za “bijeg”, dok znakovi koji slijede iza njih (“D” odnosno “G”) određuju odgovarajuće akcije. S obzirom da ne postoje nikakve definirane akcije određene slovima “D” ili “G”, kompjajler će znak “\” ignorirati, tako da bi ispis najvjerojatnije izgledao ovako:

Ta igra se nalazi u C:DOSGAMES direktoriju

Još jedan karakterističan primjer nastaje ukoliko želimo da se unutar stringa nađe znak “navodnik”. Naime, ispred njega također moramo staviti prefiks “\”. Tako, ako želimo da na ekranu ispišemo tekst

Bio sam u "Grand" hotelu

moraćemo upotrebiti naredbu

```
cout << "Bio sam u \\\"Grand\\\" hotelu\\n";
```

a ne naredbu

```
cout << "Bio sam u "Grand" hotelu\\n";
```

jer bi ova druga naredba bila pogrešno interpretirana i dovela bi do sintaksne greške (razmislite zašto).

Za razliku od većine drugih programskih jezika, u jeziku C++ nije zabranjeno dodijeliti znakovnu konstantu promjenljivim koji nisu tipa “**char**”, s obzirom da tip “**char**” u suštini spada u cjelobrojne tipove (za razliku od drugih jezika, kod kojih je znakovni tip posve neovisan od cjelobrojnih tipova). Tako je sljedeća sekvenca naredbi sasvim legalna (i ispisuje broj “65” na ekran):

```
int broj;  
broj = 'A';  
cout << broj;
```

Čak je dozvoljeno vršiti aritmetiku sa promjenljivim tipa “**char**”, što je nezamislivo u većini programskih jezika. Npr. nakon naredbi

```
ch1 = 'B';
ch2 = ch1 + 3;
```

vrijednost promjenljive “**ch2**” biće 'E', odnosno broj “69” (pogledajte tablicu ASCII kôdova da vidite zašto). Ovu osobinu jezika C++ treba koristiti razumno. S jedne strane, konstrukcije poput “**ch1++**” mogu biti korisne. Smisao ove konstrukcije je da promijeni sadržaj promjenljive “**ch1**” tako da sadrži *sljedeći znak* (u smislu poretka ASCII kôdova) u odnosu na znak koji trenutno sadrži. S druge strane, mada je iz tablice ASCII kôdova vidljivo da je '**#**' + '**A**' = '**d**', ovakva “aritmetika” zaista ne vodi ka “normalnim” programima. Ipak, treba napomenuti da je rezultat binarne aritmetičke operacije nad podacima tipa “**char**” također tipa “**char**” samo ukoliko su oba operanda tipa “**char**”, inače je tipa koji odgovara tipu drugog operanda. Na primjer, naredba

```
cout << 'A' + 1;
```

neće ispisati znak “B” nego broj “66”, s obzirom da je izraz “'A' + 1” tipa “**int**” (s obzirom da je broj “1” tipa “**int**”). Naravno, upotreboom operatora za konverziju tipa možemo specifizirati šta tačno želimo. Tako će, na primjer, naredba

```
cout << (char) ('A' + 1);
```

odnosno

```
cout << char('A' + 1);
```

zaista ispisati znak “B”. Dosta interesantna situacija nastaje u slučaju sekvenci naredbi poput

```
ch1 = 'B';
ch2 = ch1 + 3;
cout << ch2;
```

gdje su “**ch1**” i “**ch2**” znakovne promjenljive. Naime, izraz “**ch1 + 3**” je tipa “**int**”, s obzirom da je broj “3” tipa “**int**”. Međutim, vrijednost ovog izraza (koja iznosi “69”) automatski se pretvara u tip “**char**” pri smještanju u promjenljivu “**ch2**” (koja je tipa “**char**”), tako da kao krajnji rezultat ipak dobijamo ispis znaka “E”.

Već je rečeno da ako se na izlazni tok pošalje objekat tipa “**char**” (znakovna promjenljiva, znakovna konstanta ili znakovni izraz), umjesto brojne vrijednosti biće isписан odgovarajući znak. Ovu konvenciju moguće je promijeniti *eksplicitnom promjenom tipa*, što smo i ranije koristili. Tako će, npr. naredba

```
cout << (int) 'A';
```

odnosno naredba

```
cout << int('A');
```

ispisati broj “65”, jer smo eksplicitno promijenili tip objekta 'A' iz tipa “**char**” u tip “**int**”. Naravno, moguća je i obrnuta konverzija. Tako će naredba

```
cout << (char) 65;
```

odnosno naredba

```
cout << char(65);
```

ispisati slovo "A".

Pored činjenice da se objekti tipa "**char**" drugačije ponašaju od objekata ostalih cjelobrojnih tipova prilikom slanja na izlazni tok, promjenljive (ili općenitije l-vrijednosti) tipa "**char**" se drugačije ponašaju i prilikom unosa sa ulaznog toka podataka. Sljedeći program to najbolje ilustrira:

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

int main() {
    char znak;
    cout << "Unesi neki znak: ";
    cin >> znak;
    cout << "Unijeli ste znak " << znak << endl
        << "Njegova šifra je " << (int)znak << endl;
    return 0;
}
```

Mogući scenario izvršavanja ovog programa je:

```
Unesi neki znak: A
Unijeli ste znak A
Njegova šifra je 65
```

Dakle, pri čitanju promjenljive tipa "**char**" iz ulaznog toka računar ne očekuje da unesemo *broj* nego *znak*, a računar u odgovarajuću promjenljivu smješta *njegovu šifru*. Ovo je precizan opis šta se tačno dešava, a nećemo puno pogriješiti ako kažemo da se u promjenljivu *smješta znak*.

Prilikom čitanja promjenljivih tipa "**char**" pomoću operatora ">>", iz ulaznog toka se izdvaja *samo jedan znak*, dok će preostali znakovi biti izdvojeni prilikom narednih čitanja. Sljedeća sekvenca naredbi ilustrira ovu pojavu:

```
char prvi, drugi, treci, cetvrti;
cin >> prvi;
cin >> drugi;
cin >> treci;
cin >> cetvrti;
cout << prvi << drugi << treci << cetvrti << endl;
```

Nije teško shvatiti zašto je jedan od mogućih scenarija izvršavanja ovog programa slijedeći:

```
abcdefg (ENTER)  
abcd
```

Naravno, potpuno isti efekat bismo postigli i sljedećim programom:

```
char prvi, drugi, treci, cetvrti;  
cin >> prvi >> drugi >> treci >> cetvrti;  
cout << prvi << drugi << treci << cetvrti << endl;
```

Da bismo bolje shvatili kako tačno funkcionira izdvajanje znakovnih promjenljivih iz ulaznog toka, slijedi još jedan primjer mogućeg scenarija izvršavanja istog programa:

```
ab (ENTER)  
cdef (ENTER)  
abcd
```

Također je moguć i sljedeći scenario:

```
a (ENTER)  
b (ENTER)  
c (ENTER)  
d (ENTER)  
abcd
```

Prilikom čitanja, eventualni razmaci se ignoriraju, tako da je i sljedeći scenario moguć:

```
a b cd e (ENTER)  
abcd
```

Ukoliko iz bilo kojeg razloga želimo da izdvojimo znak iz ulaznog toka, bez ikakvog ignoriranja, *ma kakav on bio* (uključujući razmake, specijalne značke tipa označke novog reda, itd.) možemo koristiti poziv funkcije “`cin.get()`”. Ova funkcija (koju smo već koristili za neku drugu svrhu) izdvaja sljedeći znak iz ulaznog toka, ma kakav on bio, i vraća kao rezultat njegovu šifru (ako je ulazni tok prazan, ova funkcija zahtijeva da se ulazni tok napuni svežim podacima, npr. unosom novih podataka sa tastature, što zapravo

objašnjava efekat ove funkcije koji smo koristili u prvom poglavlju). Treba znati da i oznaka novog reda (odnosno znak '\n') kao i ostali "specijalni" znaci imaju svoje šifre (koje u ASCII tablici zauzimaju vrijednosti ispod broja "32"). Na primjer, oznaka novog reda ima ASCII šifru "10". Da bismo shvatili kako djeluje funkcija "cin.get()", razmotrimo sljedeću sekvencu naredbi:

```
char prvi, drugi, treci, cetvrti;
prvi = cin.get();
drugi = cin.get();
treci = cin.get();
cetvrti = cin.get();
cout << prvi << drugi << treci << cetvrti << endl;
```

Jedan mogući scenario izvršavanja ovog programa je sljedeći:

```
ab (ENTER)
cdef (ENTER)
ab
c
```

Da bismo shvatili prikazani scenario, moramo analizirati šta nakon izvršenja unosa u prikazanom scenariju zapravo sadrže promjenljive "prvi", "drugi", "treci" i "cetvrti". Promjenljive "prvi" i "drugi" sadrže znakove "a" i "b" (tj. njihove šifre), promjenljiva "treci" sadrži *oznaku za kraj reda* (odnosno njenu šifru "10"), dok promjenljiva "cetvrti" sadrži znak "c", koji slijedi iza oznake kraja reda. Sada je sasvim jasno zbog čega ispis ovih promjenljivih daje prikazani ispis. Razmotrimo još jedan scenario izvršavanja istog programa:

```
a b cd e (ENTER)
a b
```

U ovom primjeru, promjenljive "drugi" i "cetvrti" sadrže *razmak* (odnosno, njegovu šifru).

Srodna funkciji "cin.get()" je funkcija "cin.peek()" koja daje kao rezultat šifru *sljedećeg znaka* koji bi bio izdvojen iz ulaznog toka, ali *ne vrši njegovo izdvajanje*. Obje funkcije će biti mnogo korisnije nakon što upoznamo naredbe ponavljanja koje će nam omogućiti da iščitamo čitav ulazni tok znak po znak, bez obzira koliko on znakova sadrži. Ipak, na ovom mjestu nije na odmet navesti da tip rezultata koji ove funkcije vraćaju nije "**char**" nego "**int**". Stoga, naredba poput

```
cout << cin.get();
```

neće ispisati sljedeći *znak* izdvojen iz ulaznog toka, već njegovu *šifru* (u vidu broja).

Sljedeći primjer programa ilustrira jedan korisan primjer upotrebe aritmetike sa znakovnim promjenljivim:

```

#include <iostream>
using namespace std;

int main() {
    char ch1, ch2;
    cout << "Unesi veliko slovo: ";
    cin >> ch1;
    ch2 = ch1 + 'a' - 'A';
    cout << "Malo slovo je: " << ch2;
    return 0;
}

```

Slika demonstrira izvršavanje ovog programa:

```

Unesi veliko slovo: H
Malo slovo je: h

```

Ako ne razumijete kako ovaj program radi, proučite malo tablicu ASCII kôdova. Primijetimo da će program dati besmislene rezultate ako kao ulaz *ne unesete* veliko slovo. Također, primijetimo da se isti program mogao napisati i ovako:

```

#include <iostream>
using namespace std;

int main() {
    char ch;
    cout << "Unesi veliko slovo: ";
    cin >> ch;
    cout << "Malo slovo je: " << ch + 'a' - 'A' << endl;
    return 0;
}

```

Ovdje treba obratiti pažnju da je izraz “`ch + 'a' - 'A'`” tipa “`char`”.

Uključivanjem zaglavlja biblioteke “`cctype`” (koje se u ranijim verzijama jezika C++ zvalo “`ctype.h`”) dobijamo nekoliko interesantnih funkcija za pretvaranje znakova koji odgovaraju malim slovima u velika, i obrnuto. Tako, funkcije “`tolower`” i “`toupper`” primaju kao argument znakovni izraz, koji konvertiraju u malo odnosno veliko slovo respektivno. Tako bi se prethodni program mogao napisati i na sljedeći način:

```

#include <iostream>
#include <cctype>

using namespace std;

int main() {
    char ch;
    cout << "Unesi veliko slovo: ";
    cin >> ch;
    cout << (char)tolower(ch);
    cout << endl;
}

```

```
    return 0;  
}
```

Eksplisitna konverzija u tip “**char**” je potrebna zbog činjenice da je rezultat funkcija “`tolower`” i “`toupper`” tipa “**int**” (tako da ove funkcije zapravo vraćaju kao rezultat šifru znaka nakon obavljene pretvorbe u malo odnosno veliko slovo). Također je interesantno da obje funkcije vraćaju znak neizmijenjenim ukoliko znak ne predstavlja malo odnosno veliko slovo (tako da prethodni program neće ispisati besmislice ukoliko ne unesemo veliko slovo). Ovo čini ove dvije funkcije nešto manje efikasnim nego što bi mogle biti (jer se gubi vrijeme na provjeru da li je znak koji se pretvara ispravan). Ukoliko smo sigurni da znak koji pretvaramo predstavlja veliko odnosno malo slovo, za njegovu konverziju u malo odnosno veliko slovo možemo koristiti funkcije “`_tolower`” odnosno “`_toupper`” iz iste biblioteke. Ove funkcije su neznatno efikasnije od “`tolower`” odnosno “`toupper`”, ali daju besmislene rezultate u slučaju da znak koji pretvaramo nije veliko odnosno malo slovo.

Ranije smo spominjali mogućnost zadavanja širine ispisa uz pomoć funkcije “`cout.width`” ili manipulatora “`setw`”. Pri tome smo naveli da se prazan prostor koji je potreban da se ispisi raširi do željene širine ispunjava prazninama (odnosno razmacima). Pomoću funkcije “`cout.fill`” može se zadati *znak* koji će se umjesto razmaka koristiti za ispunjavanje ove praznine. Ova funkcija kao argument prihvata znak koji će biti korišten (argument može biti bilo koji znakovni izraz, pa čak i cijelobrojni izraz, koji će biti konvertiran u znakovni tip). Na primjer, ukoliko izvršimo niz naredbi

```
cout.width(10);  
cout.fill('*');  
cout << 23;
```

biće ispisano nešto poput

*****23

Isti efekat možemo postići uz pomoć manipulatora “`setfill`”, odnosno naredbe poput

```
cout << setw(10) << setfill('*') << 23;
```

Za korištenje manipulatora “`setfill`” (kao i za korištenje svih ostalih manipulatora), potrebno je u program uključiti biblioteku “`iomanip`”.

Prilikom rada sa ulaznim tokom vidjeli smo da bez obzira da li koristimo operator izdvajanja “`>>`” ili funkciju “`cin.get()`”, podaci koji nisu izdvojeni iz ulaznog toka ostaju u njemu, i biće izdvojeni prilikom sljedeće upotrebe ulaznog toka. Ovo je u nekim slučajevima povoljno. Međutim, često želimo da budemo sigurni da će neka naredba za unos podataka poput “`cin >> a`” uvijek prihvati “svježe” podatke sa tastature, a ne neke podatke koji su eventualno preostali u ulaznom toku prilikom prethodnih unosa. Za tu svrhu možemo koristiti funkciju “`cin.ignore`”. Ova funkcija prihvata *dva argumenta razdvojena zarezom*, od kojih je prvi cijelobrojnog tipa, a drugi znakovnog tipa. Ova funkcija uklanja znakove iz ulaznog toka, pri čemu se uklanjanje obustavlja ili kada se ukloni onoliko znakova koliko je zadano prvim argumentom, ili dok se ne ukloni znak zadan drugim argumentom. Na primjer, naredba

```
cin.ignore(50, '.');
```

uklanja znakove iz ulaznog toka dok se ne ukloni 50 znakova, ili dok se ne ukloni znak ‘.’. Ova naredba se najčešće koristi da *kompletno isprazni* ulazni tok. Za tu svrhu, treba zadati naredbu poput

```
cin.ignore(10000, '\n');
```

Ova naredba će uklanjati znakove iz ulaznog toka ili dok se ne ukloni 10000 znakova, ili dok se ne ukloni oznaka kraja reda '\n' (nakon čega je zapravo ulazni tok prazan). Naravno, kao prvi parametar smo umjesto 10000 mogli staviti neki drugi veliki broj (naš je cilj zapravo *samo* da uklonimo sve znakove dok ne uklonimo oznaku kraja reda, ali moramo nešto zadati i kao prvi parametar). Opisana tehnika je iskorištena u sljedećoj sekvenci naredbi:

```
int broj1, broj2;
cout << "Unesi prvi broj: ";
cin >> broj1;
cin.ignore(10000, '\n');
cout << "Unesi drugi broj: ";
cin >> broj2;
```

Izvršenjem ovih naredbi ćemo biti sigurni da će se na zahtjev "Unesi drugi broj:" zaista tražiti unos broja sa tastature, čak i ukoliko je korisnik prilikom zahtjeva "Unesi prvi broj:" unio odmah dva broja. Treba napomenuti da će naredba "cin.ignore" u slučaju da se izlazni tok isprazni prije nego što je dostignut neki od dva moguća kriterija za njeno završavanje, očekivati od korisnika unos novih znakova sa ulaznog toka, i to bez ikakve naznake šta se dešava.

Ranije smo rekli da je dužina "**char**" promjenljivih uvijek tačno 1 bajt. To je zbog toga što u 1 bajt memorije može stati tipično 256 različitih kombinacija, što je dovoljno da se pokriju sva "normalna" mala i velika slova, cifre i standardni interpunkcijski znakovi. Primijetimo da ASCII standard nije definirao šifre za naša slova (što ne treba mnogo da čudi), tako da su na području bivše Jugoslavije nastajale razne "zakrpe" sa ciljem da se i našim slovima dodijele odgovarajuće šifre. Problem je u tome što te razne "zakrpe" nisu saglasne međusobno (tj. za isto slovo, npr. "Š" jedan "standard" predviđa jednu šifru, a drugi drugu), tako da nisu rijetke pojave da se u nekom dokumentu koji sadrži naša slova ona izgube (ili pretvore u besmislene znakove) prilikom prenosa na neki drugi računar. Cijeli problem je u tome što izvorni i odredišni računar ne koriste iste šifre za memoriranje naših slova. Danas je u razvoju novi standard šifri nazvan UNICODE koji predviđa ne samo naša slova, nego i sva slova grčkog, arapskog, hebrejskog, indijskog, kineskog i svih drugih alfabetova koji se koriste na svijetu. Ukratko, UNICODE standard predviđa preko 30000 različitih znakova. Da bi se podržao rad sa znakovima zapisanih u UNICODE standardu, nedavno je u jezik C++ pored tipa "**char**" uveden i tip "**wchar_t**" koji omogućava pamćenje znakova po UNICODE standardu. UNICODE standard zahtijeva 2 bajta za pamćenje jednog znaka, pa je logično očekivati da promjenljive tipa "**wchar_t**" zauzimaju 2 bajta memorije. Ovo jeste slučaj u većini raspoloživih kompjajlera za C++, mada standard nije precizirao tačno koliko promjenljive tipa "**wchar_t**" trebaju zauzimati memorije. Jedino je definirano da one moraju zauzimati *barem 2 bajta*, kao i da njihova veličina mora biti jednaka veličini nekog od cijelobrojnih tipova na istom kompjajleru.

Dobra stvar UNICODE standarda je u tome što je on saglasan sa ASCII standardom: oni znakovi koji već postoje u ASCII standardu zadržavaju iste šifre. Tako, slovo "A" i u UNICODE standardu ima šifru 65, kao i u ASCII standardu. Stoga nema ništa loše u deklaraciji poput

```
wchar_t znak = 'A';
```

S druge strane, ukoliko želimo da definiramo znakovnu konstantu koja će eksplisitno biti po UNICODE standardu, ispred prvog apostrofa trebamo staviti prefiks "L", na primjer:

```
wchar_t nase_slovo = L'Š';
```

Nažalost, treba primijetiti da je za rad sa znakovima po UNICODE standardu potrebna izvjesna podrška operativnog sistema, koja je u trenutku pisanja ove skripte na većini raspoloživih operativnih sistema bila

dosta traljava. Zbog toga su još uvijek promjenljive tipa “**wchar_t**” prilično ograničene upotrebne vrijednosti.

9. Logički izrazi i operatori

Do sada smo isključivo koristili *aritmetičke izraze*, u kojima su se pojavljivali brojevi, brojčane promjenljive, i aritmetičke operacije poput sabiranja, itd. Njihov rezultat je uvijek bio neka *brojčana vrijednost*. Sada ćemo se upoznati sa *logičkim izrazima*. To su izrazi za koje jedino možemo utvrditi da su *tačni* (engl. *true*) ili *netačni* (engl. *false*). Drugim riječima, jedine vrijednosti koje ima smisla pripisati logičkim izrazima su vrijednosti “tačno” odnosno “netačno”. Logički izrazi se još nazivaju i *uvjeti* (engl. *conditions*).

Najjednostavniji uvjeti formiraju se *poređenjem dvije vrijednosti*, koje možemo izvršiti korištenjem *relacionalnih operatora* (koji spadaju u grupu tzv. *logičkih operatora*). Jezik C++ poznaje sljedeće relacione operatore:

<code>==</code>	<i>jednako</i>
<code>!=</code>	<i>nije jednako (različito)</i>
<code><</code>	<i>manje od</i>
<code>></code>	<i>veće od</i>
<code><=</code>	<i>manje od ili jednako</i>
<code>>=</code>	<i>veće od ili jednako</i>

Na primjer, uvjet “`2 > 3`” je *netačan* (tj. njegova vrijednost je “netačno”), dok je uvjet “`1 <= 4`” *tačan* (njegova vrijednost je “tačno”). Uvjeti mogu sadržavati *promjenljive*, i tada njihova tačnost ovisi od trenutnih vrijednosti promjenljivih upotrijebljenih unutar uvjeta. Na primjer, uz pretpostavku da su deklarirane sljedeće promjenljive:

```
int stanje_kase, porez;
double broj;
char inicijal;
```

možemo formirati uvjete poput sljedećih:

```
stanje_kase == 100
porez < 1000
broj > 5.2
inicijal == 'P'
```

Prvi uvjet je tačan samo ukoliko je vrijednost promjenljive “`stanje_kase`” jednaka 100, inače je netačan. Analogno vrijedi za preostala tri uvjeta. Naročito obratite pažnju na razliku između operatora “`==`” sa značenjem “da li je jednako” i operatora dodjele “`=`” sa značenjem “postaje”. Na ovu razliku ćemo detaljno ukazati nešto kasnije jer je ona *Ahilova peta* jezika C++ i čest je uzrok veoma ozbiljnim greškama u programima, koje se teško uočavaju.

Najveću primjenu logički izrazi imaju za formiranje naredbi grananja i naredbi ponavljanja, koje ćemo upoznati u narednim poglavljima. Na ovom mjestu ćemo razmotriti *logičke promjenljive* (ili *Booleove promjenljive*, u čast Georga Boolea, osnivača matematičke logike), koje pamte da li je neki uvjet bio ispunjen ili nije. Da bismo vidjeli kako se formiraju ovakve promjenljive, moramo prvo vidjeti šta je tačno vrijednost nekog uvjeta. U jeziku C vrijedi konvencija da je vrijednost svakog tačnog uvjeta jednaka jedinici, dok je vrijednost svakog netačnog uvjeta jednaka nuli (drugim riječima, vrijednosti “tačno” i “1” odnosno vrijednosti “netačno” i “0” su poistovjećene). Dugo vremena (sve do pojave ISO C++98 standarda) ista konvencija je vrijedila i u jeziku C++. Međutim standard ISO C++98 uveo je dvije nove

ključne riječi “**true**” i “**false**” koje respektivno predstavljaju vrijednosti “tačno” odnosno “netačno”. Tako je vrijednost svakog tačnog izraza “**true**”, a vrijednost svakog netačnog izraza “**false**”. Uvedena je i ključna riječ “**bool**” kojom se mogu deklarirati promjenljive koje mogu imati samo vrijednosti “**true**” odnosno “**false**”. Na primjer, ako imamo sljedeće deklaracije:

```
bool u_dugovima, punoljetan, polozio_ispit;
```

tada su sasvim ispravne sljedeće dodjele (uz pretpostavku da također imamo deklarirane brojčane promjenljive “stanje_kase” i “starost”):

```
u_dugovima = stanje_kase < 0;
punoljetan = starost >= 18;
polozio_ispit = true;
```

Za logičke izraze kažemo da su logičkog tipa, odnosno “**bool**”. Međutim, kako je dugo vremena vrijedila konvencija da su logički izrazi numeričkog tipa, preciznije cjelobrojnog tipa “**int**” (s obzirom da im je pripisivana cjelobrojna vrijednost 1 ili 0), uvedena je automatska pretvorba logičkih vrijednosti u numeričke i obratno, koja se vrši po potrebi, i koja omogućava miješanje aritmetičkih i logičkih operatora u istom izrazu, na isti način kako je to bilo moguće i prije uvođenja posebnog logičkog tipa. Pri tome vrijedi pravilo da se, u slučaju potrebe za pretvorbom logičkog tipa u numerički, vrijednost “**true**” konvertira u cjelobrojnu vrijednost “1”, dok se vrijednost “**false**” konvertira u cjelobrojnu vrijednost “0”. U slučaju potrebe za obrnutom konverzijom, nula se konvertira u vrijednost “**false**” dok se *svaka numerička vrijednost (cjelobrojna ili realna) različita od nule* (a ne samo jedinica) konvertira u vrijednost “**true**”. Posljedice ove činjenice razmotrićemo u poglavljima koji slijede. Na ovom mjestu ćemo navesti jedan jednostavan primjer. Razmotrimo sljedeći programski isječak:

```
bool a;
a = 5;
cout << a;
```

Ovaj isječak će ispisati na ekran vrijednost “1”. Pri dodjeli “`a = 5`” izvršena je automatska konverzija cjelobrojne vrijednosti “5” u logičku vrijednost “**true**”. Konverzija je izvršena zbog činjenice da je odredište (promjenljiva “`a`”) u koju smještamo vrijednost logičkog tipa. Prilikom ispisa na ekran, logička vrijednost “**true**” konvertira se u cjelobrojnu vrijednost “1”, tako da pri ispisu dobijamo jedinicu. Naravno, ovaj primjer je potpuno vještački formiran, ali pomaže da se lakše shvati šta se zapravo dešava.

Zbog činjenice da je podržana automatska dvosmjerna konverzija između numeričkih tipova i logičkog tipa, moguće je kombinirati aritmetičke i logičke operatore u istom izrazu. Na primjer, ukoliko se kao neki od operanada operatorka sabiranja “+” upotrijebi logička vrijednost (ili logički izraz), ona će biti konvertirana u cjelobronu vrijednost, s obzirom da operatorka sabiranja nije prirodno definiran za operande koji su logičke vrijednosti. Stoga je izraz

```
5 + (2 < 3) * 4
```

potpuno legalan, i ima vrijednost “9”, s obzirom da se vrijednost izraza “`2 < 3`” koja iznosi “true” konvertira u vrijednost “1” prije nego što se na nju primjeni operacija množenja. Ova osobina se često može korisno upotrijebiti. Na primjer, uz pretpostavku da su “`a`” i “`b`” cjelobrojne promjenljive, naredba

```
a = a + (b < 5);
```

ili, ekvivalentno, naredba

```
a += b < 5;
```

uvećava sadržaj promjenljive “a” za 1 pod uvjetom da je vrijednost promjenljive “b” manja od 5, inače je ostavlja nepromijenjenom (s obzirom da tada izraz “ $b < 5$ ” ima vrijednost “**false**”, koja se konvertira u nulu). Razlog zašto smo u prvom slučaju upotrijebili zagrdu a u drugom nismo, uvidjećemo uskoro (u pitanju je prioritet odgovarajućih operatora).

U nekim slučajevima na prvi pogled nije jasno da li se treba izvršiti pretvorba iz logičkog u numerički tip ili obrnuto. Na primjer, neka je “a” logička promjenljiva čija je vrijednost “**true**”, a “b” cjelobrojna promjenljiva čija je vrijednost “5”. Postavlja se pitanje da li je uvjet “ $a == b$ ” tačan. Odgovor zavisi od toga kakva će se pretvorba izvršiti. Ako se vrijednost promjenljive “a” pretvori u cjelobrojnu vrijednost “1”, uvjet neće biti tačan. S druge strane, ako se vrijednost promjenljive “b” pretvori u logičku vrijednost “**true**”, uvjet će biti tačan. U jeziku C++ uvijek vrijedi pravilo da se u slučaju kada je moguće više različitih pretvorbi, uvijek *uži tip* (po opsegu vrijednosti) pretvara u *širi tip*. Dakle, ovdje će logička vrijednost promjenljive “a” biti pretvorena u cjelobrojnu vrijednost, tako da uvjet neće biti tačan.

S obzirom na automatsku konverziju koja se vrši između logičkog tipa i numeričkih tipova, promjenljive “*u_dugovima*”, “*punoljetan*” i “*polozio_ispit*” iz jednog od ranijih primjera mogle su se deklarirati i kao obične cjelobrojne promjenljive (tipa “**int**”). Do uvođenja tipa “**bool**”, tako se i moralo raditi. Međutim, takvu praksu danas treba strogo izbjegavati, jer se na taj način povećava mogućnost zabune, i program čini nejasnjim. Stoga, ukoliko je neka promjenljiva zamišljena da čuva samo logičke vrijednosti, nju treba deklarirati upravo kao takvu.

Već smo upoznali na desetine različitih operatora u jeziku C++. Iz matematike je jasno da množenje i dijeljenje ima veći prioritet u odnosu na sabiranje i oduzimanje, ali se postavlja pitanje *kakav je prioritet ostalih operatora*. Na primjer, može se postaviti pitanje da li će se izraz “ $a + b < c + d$ ” interpretirati kao

$$(a + b) < (c + d)$$

ili kao

$$a + (b < c) + d$$

Iako su obe varijante u jeziku C++ principijelno *dozvoljene*, druga varijanta izgleda *nelogično*. Jezik C++ je dodijelio takve prioritete operatorima da je (osim u slučajevima kada upotrebom zagrada naznačimo drugačije) obično *logičnija* varijanta ispravna (mada je sam pojam “logičan” dosta upitan). U jeziku C++ postoji čak 17 *nivoa prioriteta*. Da ne bi bilo zabune, ovdje navodimo potpunu tablicu prioriteta svih C++ operatora (mnoge od njih nismo još upoznali, a neke nećemo ni upoznati):

Prioritet:	Operatori:
-------------------	-------------------

1. (najviši)	<code>::</code>	<code>:: (unarni)</code>					
2.	<code>()</code>	<code>[]</code>	<code>-></code>	<code>::</code>	<code>++ (postfiks)</code>	<code>-- (postfiks)</code>	
3.	<code>!</code>	<code>~</code>	<code>+ (unarni)</code>	<code>- (unarni)</code>	<code>& (unarni)</code>	<code>* (unarni)</code>	
	<code>++ (prefiks)</code>	<code>-- (prefiks)</code>	<code>new</code>	<code>delete</code>	<code>sizeof</code>	<code>typeid</code>	
4.	<code>. *</code>	<code>->*</code>	<i>(tip)</i>				
5.	<code>*</code>	<code>/</code>	<code>%</code>				
6.	<code>+</code>	<code>-</code>					
7.	<code><<</code>	<code>>></code>					
8.	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>			
9.	<code>==</code>	<code>!=</code>					
10.	<code>&</code>						

11.	\wedge
12.	$ $
13.	$\&\&$
14.	$\ $
15.	$? :$
16.	$=$ $+=$ $-=$ $*=$ $/=$ $\%=$ $\&=$ $^=$ $ =$ $<<=$ $>>=$
17. (najniži)	,

Množenje i dijeljenje imaju prioritet 5, a sabiranje i oduzimanje prioritet 6. Znaci “+” i “–“ spomenuti pod prioritetom 3 (uz napomenu “unarni” u zagradi) predstavljaju operatore za predznak (npr. “–7”, “+3” ili “–(2*a)”). Operatori “+” i “–“ su primjeri operatora koji postoje i u unarnoj i u binarnoj varijanti. Takvi su također i operatori “*”, “&“ i “: :” (u binarnoj varijanti, operator “*” predstavlja množenje, dok ćemo se sa značenjem njegove unarne varijante upoznati kasnije). Kao što smo ranije vidjeli, unarni operatori “++” i “––“ postoje u prefiksnoj i postfiksnoj verziji (“++a“ odnosno “a++“), koje imaju neznatno različite prioritete.

Treba primijetiti da svi relacioni operatori (poput “<” itd.) imaju prioritet niži od aritmetičkih operatora (poput “+”, “–“ itd.), pa se, zbog toga, izrazi poput

$a + b < c + d$

grupiraju “ispravno” kao

$(a + b) < (c + d)$

Operatori dodjele (“=”, “+=” itd.) imaju *veoma nizak prioritet* (što je sasvim logično), tako da će ranije spomenute dodjele poput

`u_dugovima = stanje_kase < 0;`

biti ispravno shvaćene kao

`u_dugovima = (stanje_kase < 0);`

a ne kao

`(u_dugovima = stanje_kase) < 0;`

Šta bi radila ova druga varijanta? Ona bi izvršila dodijelu promjenljivoj “u_dugovima” vrijednost promjenljive “stanje_kase”, a zatim ispitala da li je rezultat te dodjele manji od nule, i ovisno od toga, vratila rezultat “true” ili “false”. Ovo *vrlo vjerovatno* nije ono što smo željeli, tako da kompjaler, posve logično, podrazumijeva prvu varijantu. Naravno, ukoliko je baš to ono što želimo, uvijek možemo upotrijebiti zagrada da *eksplicitno* naznačimo redoslijed izvođenja operacija.

Jedini operator koji ima niži prioritet od dodjele je *zarez*. Značenje zareza kao operatorka uvidjećemo u poglavlju o naredbama ponavljanja. Na ovom mjestu ćemo samo reći da je tako nizak prioritet operatorka “zarez” u skladu sa intuitivnim shvatanjem da konstrukcije poput

`int a = 5, b = 10;`

ne treba da budu “besmisleno” grupirane kao

```
int a = (5, b) = 10;
```

S druge strane, operatori “`++`” i “`--`” i u prefiksnoj i u postfiksnoj formi imaju *veoma visok prioritet*, tako da se “otkačena” naredba, koju smo pisali u jednom od ranijih poglavlja

```
b = 3 + 2 * (a++);
```

mogla pisati i bez zagrade kao

```
b = 3 + 2 * a++;
```

Naravno, ako Vas već “boli glava” od razmišljanja o prioritetima operatora, ništa loše nema u tome da sami pomoći zagrada eksplisitno odredite kakav redoslijed operacija želite, čak i ako se takav redoslijed podrazumijeva i bez zagrada.

Izbor prioriteta pojedinih operatora nije baš “najsretnije” odabran. Na primjer, ako želimo da se “svojim očima” uvjerimo kolika je vrijednost izraza “`2 < 3`”, imaćeemo problema ako napišemo naredbu

```
cout << 2 < 3;
```

Naime, operator “`<<`” kojim vršimo ispis na ekran (napomenimo ovo nije jedina funkcija ovog operatora) ima veći prioritet u odnosu na operator “`<`”, pa se gornja naredba besmisleno grupira kao

```
(cout << 2) < 3;
```

Posljednji izraz je sintaksno neispravan, jer se rezultat izraza “`cout << 2`” (koji zapravo predstavlja sam objekat “`cout`”, što u ovom trenutku nije bitno) ne može upotrijebiti kao lijevi operand operatora “`<`”. Rješenje ovog problema je, naravno, u korištenju zagrada:

```
cout << (2 < 3);
```

Složeniji logički izrazi mogu se formirati korištenjem logičkih operatora. Programski jezik C++ poznaje sljedeće logičke operatore:

<code>&&</code>	konjukcija (logičko “i”)
<code> </code>	disjunkcija (logičko “ili”)
<code>!</code>	logička negacija

Ovi operatori kao svoje operative očekuju logičke vrijednosti, ali će prihvati i numeričke vrijednosti, koje će tom prilikom biti pretvorene u logičke, po već opisanim pravilima. Interpretacija ovih operatora predstavljena je sljedećom tablicom:

<code>false && false = false</code>	<code>false false = false</code>	<code>!false = true</code>
<code>false && true = false</code>	<code>false true = true</code>	<code>!true = false</code>
<code>true && false = false</code>	<code>true false = true</code>	
<code>true && true = true</code>	<code>true true = true</code>	

Logički operatori se najčešće koriste za formiranje *kombiniranih uvjeta*. Na primjer, sljedeća dva uvjeta koriste logičke operatore:

```
starost >= 16 && starost < 65
```

```
starost < 16 || starost >= 65
```

Prvi od ova dva izraza je tačan ukoliko je vrijednost promjenljive “starost” veća ili jednaka 16 i manja od 65, a drugi je tačan ukoliko je vrijednost promjenljive “starost” manja od 16 ili veća ili jednaka 65. Obratite pažnju da su ova dva izraza uvijek *suprotna po vrijednosti* (ako je prvi tačan, drugi je netačan, i obrnuto). Također, u oba izraza dodatne zagrade nisu potrebne, zbog toga što operatori “**&&**” i “**||**” imaju niži prioritet od operatora poređenja. Naravno, ništa ne bi bilo loše da smo napisali:

```
(starost >= 16) && (starost < 65)  
(starost < 16) || (starost >= 65)
```

Uvođenje dodatnih zagrada je korisna praksa ako nismo sigurni u prioritet operacija, a ono nam također pomaže i da “olakšamo život” onima koji pokušavaju da shvate šta i kako program radi.

Neko bi se mogao zapitati da li bi se prvi od prethodna dva uvjeta mogao napisati na sljedeći način, koji je uobičajen u matematici, a koji je podržan u nekim specijalističkim matematičkim programskim jezicima (npr. u programskom paketu *Mathematica*):

```
16 <= starost < 65
```

Odgovor je određan. Na žalost, gornji izraz je sintaksno *potpuno ispravan*, ali ne obavlja onu funkciju koju bismo očekivali. Zapravo, prethodni izraz je *uvijek tačan*, bez obzira na vrijednost promjenljive “starost”! Da bismo ovo pokazali, razmotrimo kako se ovaj izraz tačno interpretira. S obzirom da su operatori “**<=**” i “**<**” istog prioriteta, ovaj izraz se izračunava redom, slijeva nadesno, odnosno interpretira se kao

```
(16 <= starost) < 65
```

Šta se sad dešava? Vrijednost izraza “ $16 \leq \text{starost}$ ” je “**true**” ili “**false**”, zavisno kakva je vrijednost promjenljive “starost”. Ta vrijednost se sada upoređuje sa brojem 65, pri čemu dolazi do njene pretvorbe iz logičke vrijednosti u cjelobrojnu vrijednost “0” ili “1”. Međutim, ma koja od ove dvije vrijednosti manja je od 65, tako da je cijelokupan izraz na kraju tačan! Kao pouku, možemo izvesti zaključak da gotovo nikada nema smisla u istom izrazu koristiti više od jednog relacionog operatora, osim u slučaju kada se između njih nalazi neki od logičkih operatora “**&&**” ili “**||**”.

Operator “**!**” može se koristiti za *negiranje* uvjeta, ali zbog njegovog veoma visokog prioriteta, dodatne zagrade su skoro uvijek neophodne. Na primjer:

```
!(starost > 65)
```

Ovaj uvjet je tačan samo ukoliko vrijednost promjenljive “starost” *nije* veća od 65. Međutim, da smo izostavili zagrade, tj. da smo napisali uvjet

```
!starost > 65
```

on bi bio interpretiran *pogrešno*. Naime, operator “**!**” bio bi primijenjen samo na promjenljivu “starost”. Kako je ona cjelobrojna promjenljiva, a operator “**!**” očekuje logičku vrijednost, njena vrijednost bi bila konvertirana u “**true**” odnosno “**false**” (zavisno da li joj je vrijednost različita od nule ili jednaka nuli). Nakon negacije, ta vrijednost bi postala “**false**” odnosno “**true**” respektivno. Konačno, ta bi se vrijednost poredila sa brojem 65. Kako se logička vrijednost ne može direktno porediti sa brojem, ona bi bila pretvorena u brojčanu vrijednost “0” ili “1”, koja bi se poredila sa brojem 65.

Rezultat poređenja bi u svakom slučaju bio netačan, tako da bi konačna vrijednost ovog uvjeta bila “**false**”. To sigurno nije ono što smo željeli.

Treba razmotriti još neke greške koje početnici mogu učiniti pri korištenju logičkih operatora. Zamislimo da želimo da formiramo uvjet koji je tačan ako i samo ako je vrijednost promjenljive “*a*” jednaka 2 ili 3. Ispravno bi bilo napisati sljedeću konstrukciju:

```
a == 2 || a == 3
```

Međutim, početnik bi, ponesen *analogijom sa govornim jezikom*, mogao napisati sljedeći uvjet:

```
a == 2 || 3
```

Problem je što je ovaj uvjet *sintaksno ispravan*, ali ne radi ono što treba (zapravo, on je uvijek *tačan*). Naime, prvi operand operatora “**||**” je izraz “*a == 2*”, koji je tačan ili netačan, ovisno od vrijednosti promjenljive “*a*” (operator “**==**” ima veći prioritet u odnosu na operator “**||**”), dok je njegov drugi operand cijeli broj “3”. Kako “3” nije logička vrijednost, ona se konvertira u logičku vrijednost “**true**”. Kako je vrijednost disjunkcije jednaka “**true**” ukoliko barem jedan od operanada ima vrijednost “**true**”, to i cijeli izraz ima vrijednost “**true**”, neovisno od vrijednosti promjenljive “*a*”. Još besmisleniju interpretaciju čemo dobiti ukoliko napišemo:

```
a == (2 || 3)
```

Ovaj izraz će biti tačan ako i samo ako je vrijednost promjenljive “*a*” jednaka jedinici (razmislite zašto). Problem je, u suštini, veoma sličan problemu koji smo imali pri interpretaciji sintaksno ispravnog ali logički nekorektnog izraza “*16 <= starost < 65*”. Ako Vam sve ovo djeluje konfuzno, zapamtite barem sljedeće: ne pravite ovakve greške.

Standard ISO C++98 jezika C++ predvio je da se kao alternativa za oznake operatora “**&&**”, “**||**” i “**!**” mogu koristiti i ključne riječi “**and**”, “**or**” i “**not**”. Tako smo maloprije napisane uvjete mogli napisati i na sljedeći način:

```
starost >= 16 and starost < 65  
starost < 16 or starost >= 65  
not(starost > 65)
```

Ipak, ovaj način pisanja nije uobičajen. Prvo, uveden je u jezik C++ tek nedavno, a drugo, smatra se da nije “u duhu” jezika C++.

Budite na oprezu: pored logičkih operatora “**&&**” i “**||**”, jezik C++ posjeduje i operatore “**&**” i “**|**” sa *sasvim drugačijim značenjem* (koji uopće nisu logički operatori). Tako ako greškom napišete izraz

```
starost >= 16 & starost < 65
```

kompajler neće prijaviti nikakvu grešku, jer se također radi o legalnom sintaksno ispravnom izrazu, ali koji ne predstavlja ono što smo (vjerovatno) zamislili. Ovdje se opet susrećemo sa problemom da prilikom izvršavanja programa računar nažalost ne izvršava *ono što smo zamislili* nego *ono što smo napisali*.

Interesantna osobina operatora “**&&**” i “**||**” je da se u nekim slučajevima njihov desni operand *uopće ne izračunava*. Ukoliko pretpostavimo da “*x*” i “*y*” predstavljaju neke logičke ili numeričke izraze, tačna interpretacija izraza “*x && y*” i “*x || y*” je sljedeća:

$x \&& y$ Ako x ima vrijednost “0” ili “**false**”, vrijednost čitavog izraza je “**false**”, pri čemu se y uopće i ne pokušava izračunati. U suprotnom se izračunava i y . Ako je njegova vrijednost “0” ili “**false**”, vrijednost čitavog izraza je “**false**”. U suprotnom, vrijednost čitavog izraza je “**true**”.

$x \mid\mid y$ Ako x ima vrijednost “**true**” ili brojnu vrijednost različitu od “0” ili, vrijednost čitavog izraza je “**true**”, pri čemu se y uopće i ne pokušava izračunati. U suprotnom se izračunava i y . Ako je njegova vrijednost “**true**” ili brojna vrijednost različita od “0”, vrijednost čitavog izraza je “**true**”. U suprotnom, vrijednost čitavog izraza je “**false**”.

Ako pažljivo razmotrimo ove interpretacije, vidjećemo da ćemo na kraju kao rezultat dobiti upravo ono što očekujemo. Međutim, karakteristično je da se vrijednost izraza “ y ” uopće ne računa ukoliko se rezultat može predvititi samo na osnovu operanda “ x ”. Ovakav princip naziva se *skraćeno izračunavanje* (engl. *short evaluation*). Programer uglavnom ne treba da brine o ovome, ali su izvjesna iznenađenja ipak moguća ukoliko izraz “ y ” sadrži *bočne efekte*. Naime, pod određenim uvjetima ovaj izraz uopće neće biti izračunat, pa ni odgovarajući bočni efekti neće biti izvršeni. Zamislimo, na primjer, da želimo da uvećamo sadržaj promjenljivih “ a ” i “ b ” za 1, a zatim da formiramo uvjet koji će testirati da li su obje promjenljive dostigle vrijednost 10. Neko bi mogao sve ovo da napiše u formi jednog izraza oblika

```
++a == 10 && ++b == 10
```

Međutim, izraz “ $++b == 10$ ” uopće se neće izračunavati ukoliko se izraz “ $++a == 10$ ” pokaže netačnim, tako da ni vrijednost promjenljive “ b ” neće biti uvećana. Naravno da pisanje ovakvih izraza treba izbjegavati. U ovom slučaju je bilo mnogo preglednije prvo uvećati promjenljive posebnim naredbama, a tek onda formirati izraz koji obavlja njihovo poređenje.

Treba obratiti pažnju da operatori “ $\&\&$ ” i “ $\mid\mid$ ” nisu istog prioriteta, nego operator “ $\&\&$ ” ima viši prioritet od operatora “ $\mid\mid$ ”. Tako, ako želimo da formiramo uvjet koji je tačan ukoliko je vrijednost promjenljive “ a ” jednaka 2 ili 3, a vrijednost promjenljive “ b ” jednaka 5, moramo pisati

```
(a == 2 \mid\mid a == 3) && b == 5
```

a ne samo

```
a == 2 \mid\mid a == 3 && b == 5
```

jer će ovakav uvjet, zbog prioriteta, biti interpretiran kao

```
a == 2 \mid\mid (a == 3 && b == 5)
```

Od logičkih izraza, samih za sebe, nema neke osobite koristi. Već smo rekli da je njihova glavna primjena u naredbama izbora i ponavljanja, koje ćemo upoznati u poglavljima koji slijede. Međutim, prije toga ćemo upoznati operator “ $? :$ ” koji omogućava da se logički izrazi veoma elegantno iskoriste. Ovaj operator ima *tri operanda*, i prema tome, predstavlja *ternarni operator* (jedini takve vrste u jeziku C++). Forma u kojoj se koristi ovaj operator ima sljedeći oblik:

```
x ? y : z
```

Ovdje su, u općem slučaju, “ x ”, “ y ” i “ z ” također izrazi. Izraz “ x ” bi trebao imati logičku vrijednost, a dopuštena je i proizvoljna numerička vrijednost (koja će biti automatski konvertirana u logičku, prema već objašnjениm pravilima). Interpretacija ovog izraza je sljedeća: ukoliko je izraz “ x ” *tačan*, tada je vrijednost

čitavog izraza jednaka vrijednosti izraza “y”, pri čemu se izraz “z” uopće ne pokušava izračunati. S druge strane, ukoliko je izraz “x” *netačan*, tada je vrijednost čitavog izraza jednaka vrijednosti izraza “z”, pri čemu se vrijednost izraza “y” uopće ne pokušava izračunati. Na primjer, naredba

```
b = (a > 0) ? a : -a;
```

dodjeljuje promjenljivoj “b” vrijednost promjenljive “a” ukoliko je ona pozitivna, a vrijednost “-a” ukoliko nije (drugim riječima, dodjeljuje promjenljivoj “b” *apsolutnu vrijednost* promjenljive “a”). Kako operator “? :” ima *veoma nizak prioritet*, zgrade oko operanada “x”, “y” i “z” gotovo nikad *nisu potrebne*, ali se preporučuju zbog čitljivosti. Tako smo prethodnu naredbu mogli napisati i kao

```
b = a > 0 ? a : -a;
```

S druge strane, također zbog veoma niskog prioriteta, zgrade oko čitavog izraza oblika “x ? y : z” praktično su uvijek potrebne kad god se on upotrijebi kao sastavni dio nekog složenijeg izraza. Na primjer, ukoliko bismo na ekran željeli *ispisati* apsolutnu vrijednost promjenljive “a” uz upotrebu operatora “? :”, morali bismo pisati

```
cout << ((a > 0) ? a : -a);
```

a ne samo

```
cout << (a > 0) ? a : -a;
```

jer bi u tom slučaju napisani izraz bio interpretiran kao

```
(cout << (a > 0)) ? a : -a;
```

Što je najgore, ovako interpretirani izraz je također sintaksno ispravan. Naime, rezultat izraza u zagradi prije znaka “?” je upravo objekat izlaznog toka “cout”, a vidjećemo kasnije da se on pod izvjesnim okolnostima može upotrijebiti kao logička vrijednost! Zapravo, jedan od najvećih problema jezika C++ je njegova velika sloboda, koja omogućava da mnoge konstrukcije, koje ne rade ono što bi na prvi pogled trebalo da rade, budu sintaksno ispravne, i prema tome, savršeno legalne! Interesantno je napomenuti da sličan izraz bez ijedne zgrade, tj. izraz

```
cout << a > 0 ? a : -a;
```

nije sintaksno ispravan, jer se on interpretira kao

```
((cout << a) > 0) ? a : -a;
```

a rezultat izraza “cout << a“ ne može se porediti sa nulom!

Izrazi “y” i “z” u opisu operatora “? :” mogu biti i *stringovi*, tako da je sljedeća konstrukcija savršeno ispravna:

```
cout << ((x >= 0) ? "Broj je nenegativan" : "Broj je negativan");
```

Ova konstrukcija ispisuje na ekran tekst “Broj je nenegativan” odnosno “Broj je negativan” u zavisnosti kakva je vrijednost promjenljive “x”. U sljedećem poglavlju vidjećemo kako se isti efekat na jasniji način može ostvariti uz pomoć naredbi grananja.

Nema nikakve prepreke da se kao izrazi “ y ” i “ z ” u opisu operatora “ $? :$ ” pojave ponovo izrazi koji sadrže operator “ $? :$ ”. Na taj način dobijaju se dosta konfuzne konstrukcije. Na primjer, sljedeća naredba

```
sgn = (x > 0) ? 1 : ((x == 0) ? 0 : -1);
```

dodjeljuje promjenljivoj “`sgn`” vrijednost “1”, “0” ili “−1”, zavisno od toga da li je vrijednost promjenljive “`x`” veća od nule, jednaka nuli ili manja od nule respektivno. Zbog prioriteta operacija, ista konstrukcija se mogla napisati bez ikakvih zagrada, tj. u obliku

```
sgn = x > 0 ? 1 : x == 0 ? 0 : -1;
```

za koji možemo reći sve osim da je jasan. Nemojte se mnogo uzbudjavati ako ne razumijete ovu konstrukciju. Nije velika šteta, s obzirom da će u sljedećem poglavljju biti pokazano kako se isti efekat može postići na mnogo razumljiviji način, korištenjem naredbi grananja.

Operator “ $? :$ ” naslijeden je iz jezika C, i korišten je jako mnogo u danima kada je jezik C nastajao, jer su tada primarni zahtjevi bili efikasnost i kratkoća programa. Danas, kada su jasnost i čitljivost programa primarni zahtjevi, prevelika upotreba ovih operatora se ne preporučuje, jer obično dovode do *veoma nečitljivih programa*. Efekat ovog operatora uvijek se može simulirati primjenom naredbi grananja, na mnogo čitljiviji način, a obično uz sasvim neznatan ili nikakav gubitak efikasnosti.

10. Naredbe izbora

Svi programi koje smo dosad pisali, imali su linijsku strukturu, tj. naredbe su se izvodile striktno jedna za drugom. Ovakva algoritamska struktura naziva se *sekvenca*, i predstavlja najprostiju algoritamsku strukturu. Pogledajmo, na primjer, kako je izgledao algoritam za računanje i prikaz obima i površine kruga:

- *Unesi vrijedost poluprečnika;*
- *Izračunaj obim po formuli $2 \cdot \pi \cdot \text{poluprečnik}$;*
- *Izračunaj površinu po formuli $\pi \cdot \text{poluprečnik}^2$;*
- *Ispiši vrijednost obima;*
- *Ispiši vrijednost površine;*

Međutim, za pisanje iole složenijih programa, pored *prostog niza (sekvence)* instrukcija, potrebno je formirati i algoritme koji sadrže *izbor* (engl. *selection*) izmedju dvije ili više mogućnosti, zavisno od vrijednosti nekog uvjeta. Ovakvu algoritamsku strukturu nazivamo *grananje* (engl. *branch*). Grnanje je jedna od *kontrolnih struktura*, koje određuju redoslijed u kojem se izvršavaju naredbe algoritma.

Prije nego što razmotrimo kako se formiraju strukture grananja u jeziku C++, moramo se prvo upoznati sa pojmom *bloka*. Blok predstavlja skupinu naredbi, koje su objedinjene u jednu cjelinu, koja započinje otvorenom, a završava zatvorenom vitičastom zagradom. Na primjer, tijelo funkcije čini jedan blok. Naredbe unutar bloka obično se pišu uvučeno, da vizuelno istaknu početak i kraj bloka. Moguće je grupirati proizvoljnu skupinu naredbi u blok, npr. nekoliko naredbi unutar neke funkcije. Na primjer, sljedeći program je potpuno legalan, mada za sada nema nikakvog razloga da vršimo grupiranje, jer bismo potpuno isti efekat imali i da nismo izvršili grupiranje:

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

int main() {
    int a, b, c;
    a = 2;
    {
        b = 3;
        c = a + b;
        cout << a << endl;
    }
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

Postoji više razloga za uvođenje blokova. Najvažniji razlog je što se, u suštini, sa aspekta okruženja u kojem se blok nalazi sve naredbe unutar nekog bloka tretiraju kao *jedna naredba*. Vidjećemo da u jeziku C++ postoji mnogo konstrukcija u kojima sintaksa jezika na nekom mjestu očekuje *tačno jednu naredbu*. Ukoliko na takvim mjestima trebamo upotrijebiti skupinu od više naredbi, takve naredbe ćemo prosto upakovati u blok, tako da će se čitava skupina ponašati kao jedna naredba. Dubina *gniježdenja* (engl. *nesting*) blokova može biti proizvoljna. Tako se, na primjer, unutar nekog bloka može pojaviti drugi blok, unutar njega treći blok, itd.

Drugi razlog za uvođenje blokova je ograničavanje područja "života" određenih promjenljivih. Naime, u jeziku C++ vrijedi konvencija da svaka promjenljiva "živi" od mjesta njene deklaracije, pa sve do

završetka bloka unutar kojeg je deklarirana (osim u slučaju tzv. *statičkih promjenljivih*, koje ćemo upoznati kasnije). Tako, na primjer, promjenljiva deklarirana unutar tijela neke funkcije (npr. "main" funkcije) "živi" do završetka te funkcije. Međutim, uvodenjem vještački kreiranih blokova, možemo skratiti vrijeme života neke promjenljive. Razmotrimo, na primjer, sljedeću sekvencu naredbi:

```
int a = 2, c;
{
    int b = 3;
    c = a + b;
    cout << a << endl;
    cout << b << endl;
}
cout << c << endl;
cout << b << endl;
```

Pokušamo li izvršiti ovu sekvencu naredbi, kompjajler će prijaviti grešku pri pokušaju ispisa promjenljive "b" izvan unutrašnjeg bloka (tj. na posljednjoj naredbi), s obzirom da promjenljiva "b" prestaje postojati po završetku unutrašnjeg bloka, unutar kojeg je deklarirana. Više o području "života" promjenljivih govorićemo u poglavljima posvećenim potprogramima i funkcijama.

Treba napomenuti da je unutar nekog unutrašnjeg bloka moguće deklarirati promjenljivu *istog imena* koja već postoji u pripadnom spoljašnjem bloku. Razmotrimo, na primjer, sljedeći program:

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

int main() {
    int a = 2;
    cout << a << endl;
    {
        int a = 4;
        cout << a << endl;
    }
    cout << a << endl;
    return 0;
}
```

U ovom programu imamo ponovljenu deklaraciju promjenljive "a", ali kompjajler neće prijaviti grešku, s obzirom da je ona deklarirana u posebnom bloku. Ovaj program će redom ispisati vrijednosti "2", "4" i "2", odakle vidimo da se unutar unutrašnjeg bloka vrši pristup promjenljivoj "a" deklariranoj u unutrašnjem bloku, odnosno ona ima *prioritet* u odnosu na istoimenu promjenljivu deklariranu u pripadnom vanjskom bloku. U ovom slučaju zapravo imamo dvije *neovisne* promjenljive sa istim imenom "a", pri čemu je unutar unutrašnjeg bloka promjenljiva "a" iz pripadnog spoljašnjeg bloka *sakrivena* (engl. *hidden*) istoimenom promjenljivom iz unutrašnjeg bloka. Također se kaže da je *vidokrug* (engl. *scope*) promjenljive "a" iz spoljašnjeg bloka privremeno *prekinut* pojmom istoimene promjenljive u unutrašnjem bloku, mada ona i dalje "živi" (što vidimo na osnovu ispisa njene vrijednosti nakon završetka unutrašnjeg bloka). Primjenom unarnog operatora ":::" možemo i unutar unutrašnjeg bloka pristupiti promjenljivoj "a" iz spoljašnjeg bloka, koja je privremeno sakrivena. Tako, ukoliko bismo naredbu ispisa u unutrašnjem bloku zamijenili naredbom

```
cout << ::a << endl;
```

program bi tri puta ispisao vrijednost "2". O vremenu života promjenljivih i njihovom vidokrugu detaljnije ćemo govoriti kasnije. Za sada je dovoljno da samo informativno znamo šta se dešava ukoliko se neka

promjenljiva deklarira unutar nekog bloka.

Nakon što smo se upoznali sa pojmom bloka, možemo preći na razmatranje kako se u jeziku C++ realiziraju strukture grananja. U bosanskom jeziku, izbor neke alternative koja se izvršava u slučaju da je neki uvjet ispunjen možemo iskazati pomoću riječi “**Ako**” (engl. “**If**”), dok se alternativa u slučaju neispunjena uvjeta izražava pomoću riječi “**Inače**” (engl. “**Else**”). Najjednostavniji slučaj izbora je kada vršimo neku akciju samo ako je uvjet tačan, a u suprotnom ne radimo ništa. Na primjer:

- **Ako** pada kiša:
 - Uzmi kišobran;
- **Ako** je danas radni dan:
 - Ustani;
 - Idi na posao;

S druge strane, možemo imati i *alternativni tok akcija* koje slijede ako uvjet *nije ispunjen*:

- **Ako** je danas radni dan:
 - Ustani;
 - Idi na posao;
- **Inače**:
 - Isključi zvono na budilniku;
 - Nastavi spavati;

U jeziku C++, izbor se izvodi upotrebom naredbe “**if**”. Prvo ćemo navesti nekoliko karakterističnih primjera:

```
if(stanje_kase < 0)
    cout << "Nema više novca";

if(starost >= 18)
    cout << "Punoljetni ste";

if(stanje_kase > 0) {
    stanje_kase -= zaduzenje;
    cout << "Trgovina obavljena";
}

if(stanje_kase < 0)
    cout << "Nema više novca";
else {
    stanje_kase -= zaduzenje;
    cout << "Trgovina obavljena";
}
```

Generalno, naredba “**if**” ima sljedeću strukturu:

if (*izraz*) *naredba*

za slučaj kad nema alternativne akcije, a

if (*izraz*) *naredba* **else** *naredba_2*

za slučaj kada želimo da zadamo i alternativni tok akcija. Smisao naredbe “**if**” je u sljedećem: prvo se *izračunava izraz u zagradi*, koji bi trebao da bude *logičkog tipa* (mada su dozvoljeni i numerički izrazi, pri čemu će njihova vrijednost biti konvertirana u logičku vrijednost prema pravilima opisanim u prethodnom poglavlju). Ukoliko je izraz *tačan*, izvršava se naredba “*naredba*” (u slučaju da treba izvršiti više naredbi u slučaju ispunjenja uvjeta, prosto ćemo upotrijebiti blok). Ukoliko izraz nije tačan, naredba “*naredba*” se ignorira, a izvršava se naredba “*naredba_2*” (također, naredba “*naredba_2*” se ignorira u slučaju da je izraz *tačan*). Naredba “*naredba_2*” također može biti blok. Sve ovo vrijedi ukoliko je predviđen alternativni tok akcija naznačen pomoću ključne riječi “**else**”. U slučaju da alternativni tok akcija nije naveden a izraz nije tačan, naredba “*naredba*” se prosto ignorira. U svakom slučaju, nakon izvršenja bilo direktno bilo alternativne akcije (tj. naredbe “*naredba*” ili naredbe “*naredba_2*”), program dalje nastavlja svojim tokom od sljedeće naredbe.

Mada se blok može principijelno sastojati od *samo jedne naredbe*, nema potrebe da formiramo blok ukoliko se direktni ili alternativni tok akcija sastoje od samo jedne naredbe. Tako je, sa aspekta izvršavanja, potpuno svejedno da li ćemo pisati

```
if(starost < 18)
    cout << "Maloljetni ste";
else
    cout << "Punoljetni ste";
```

ili

```
if(starost < 18) {
    cout << "Maloljetni ste";
}
else {
    cout << "Punoljetni ste";
}
```

Iz svih navedenih primjera možemo vidjeti da je dobar stil pisanja pisati blokove koje pripadaju naredbama “**if**” i “**else**” *uvučeno*, sa ciljem da vizuelno naglasimo direktni i alternativni tok akcija, iako takav način pisanja nije obavezan. Već smo vidjeli da razmaci i prelasci u novi red (osim u stringovima) imaju samo *estestku funkciju*, i da ni na kakav način ne utiču na izvršavanje programa. Tako su sljedeće konstrukcije sasvim valjane, ali ne i *preporučljive* (osim možda prve):

```
if(stanje_kase < 0) cout << "Nema više novca";

if(starost >= 18)
    cout << "Punoljetni ste";

if(starost < 18) cout << "Maloljetan"; else cout << "Punoljetan";

if(starost < 18)
    cout << "Maloljetni ste";
else
    cout << "Punoljetni ste";

if(stanje_kase > 0) {stanje_kase -= zaduzenje; cout << "Obavljeno!"; }
```

Zašto naglašavamo da je uvlačenje *samo stvar estetike*? Pogledajmo sljedeći primjer:

```
if(starost >= 18)
    cout << "Punoljetni ste\n";
```

```
cout << "Imate pravo glasati na izborima\n";
```

Programer je vjerovatno htio da se rečenice “Punoljetni ste” i “Imate pravo glasati na izborima” ispišu pod uvjерom da je promjenljiva “starost” veća ili jednaka od 18. Međutim, ovako kako su ove naredbe napisane, rečenica “Imate pravo glasati na izborima” biće ispisana bez obzira na vrijednost promjenljive “starost”. Naime, kako nisu upotrijebljene vitičaste zagrade (tj. kako nismo formirali blok), računar smatra da samo prva naredba ispisa pripada naredbi “**if**”, tj. kao da imamo sekvencu

```
if(starost >= 18)
    cout << "Punoljetni ste\n";
    cout << "Imate pravo glasati na izborima\n";
```

To što su obje naredbe za ispis bile *uvučene*, računaru ne znači da obje pripadaju naredbi “**if**”. Najgore je od svega što kompjajler neće prijaviti nikakvu grešku, jer *ne radimo ništa što nije dozvoljeno*. Da bi programer postigao ono što je vjerovatno htio da postigne, trebao bi *formirati blok*, tj. napisati sljedeće:

```
if(starost >= 18) {
    cout << "Punoljetni ste\n";
    cout << "Imate pravo glasati na izborima\n";
}
```

U izvjesnim “sretnim” okolnostima može se desiti da kompjajler “otkrije” ovakve logičke greške. Zamislimo da smo napisali (pogrešno):

```
if(starost >= 18)
    cout << "Punoljetni ste\n";
    cout << "Imate pravo glasati na izborima\n";
else
    cout << "Maloljetni ste\n";
    cout << "Nemate pravo glasati na izborima\n";
```

Kompajler će otkriti grešku, i to po nailasku na ključnu riječ “**else**”. Naime, u jeziku C++ ključna riječ “**else**” uvijek mora slijediti neposredno iza naredbe koja je bila vezana ključnom riječju “**if**”, što ovdje nije slučaj. Zaista, ovdje je naredba neposredno prije “**else**” naredba ispisa koja ispisuje rečenicu “Imate pravo glasati na izborima”, koja ne pripada naredbi “**if**”.

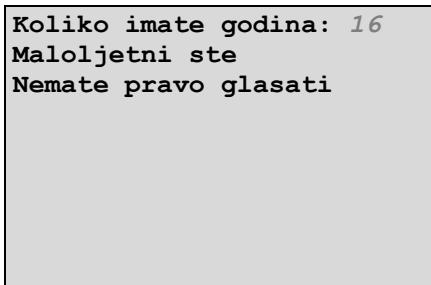
Kao ilustraciju razmotrenih koncepata, navedimo sada kompletan (i ispravan) program koji zahtijeva da unesemo svoju starost, i koji na osnovu unesene starosti daje prikladan komentar:

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

int main() {
    int starost;
    cout << "Koliko imate godina? "
    cin >> starost;
    if(starost >= 18) {
        cout << "Punoljetni ste\n";
        cout << "Imate pravo glasati\n";
    }
    else {
        cout << "Maloljetni ste\n";
        cout << "Nemate pravo glasati\n";
    }
}
```

```
    return 0;  
}
```

Sljedeća slika prikazuje šta možemo očekivati kao mogući scenario izvršavanja ovog programa:



Još jedna “ružna” greška koja se često pravi prilikom upotrebe naredbe “**if**” je *stavljanje tačka-zareza neposredno iza zagrade koja pripada naredbi “if”*. Pogledajmo sljedeći primjer:

```
if(starost >= 18);  
cout << "Punoljetni ste";
```

U ovom primjeru, rečenica “punoljetni ste” će se ispisati bez obzira na vrijednost promjenljive “starost”, jer će računar smatrati da naredbi “**if**” ne pripada *ništa*, jer je naredba zaključena prije vremena (ne zaboravimo da tačka-zarez zaključuje naredbu, tj. označava njen kraj). Preciznije, računar će smatrati da naredbi “**if**” pripada *prazna naredba* (tj. naredba koja se ne sastoji ni od čega, osim tačka-zareza koji označava njen kraj, i koja ne radi ništa). Može se postaviti pitanje zbog čega sintaksa jezika C++ uopće dozvoljava postojanje prazne naredbe. Vidjećemo kasnije da postoje razlozi kada se prazna naredba može korisno upotrijebiti (isto kao što u matematici postoje dobri razlozi da se uvede pojam praznog skupa).

Već smo rekli da bi izraz u naredbi “**if**” trebao da bude logičkog tipa, ali da je u principu dozvoljen i proizvoljan numerički (cjelobrojni ili realni) tip, zahvaljujući automatskoj konverziji numeričkih tipova u logičke, koja nastaje u slučaju da se neki numerički tip upotrijebi na mjestu gdje po prirodi stvari treba da bude logički tip. Tako, u slučaju da kao izraz u naredbi “**if**” upotrijebimo *numerički izraz*, on će se smatrati tačnim ako je *različit od nule*, a netačnim ako je *jednak nuli*. Drugim riječima, uz prepostavku da je “a” npr. cjelobrojna promjenljiva, konstrukcija

```
if(a) ...
```

potpuno je ekvivalentna konstrukciji

```
if(a != 0) ...
```

tako da možemo pisati konstrukcije poput

```
if(a) cout << "a je različito od nule";
```

kao i formirati dosta nejasne konstrukcije poput sljedeće (razmislite kako ova konstrukcija radi):

```
if(a - b) cout << "a i b su različiti";
```

Pisanje ovakvih konstrukcija bilo je jako moderno u prvim danima nastanka jezika C. Danas, a pogotovo nakon uvođenja tipa “**bool**”, pisanje ovakvih konstrukcija se smatra nepoželjnim (pogotovo u jeziku C++, za razliku od jezika C koji i dalje često propagira “hakerski” stil pisanja), s obzirom da narušava jasnoću

programa. Također, treba napomenuti da sa današnjim “inteligentnim” kompjajlerima, programi koji ne koriste ovakve “prljave trikove” ne izvršavaju se ništa manje efikasno nego programi koji ih koriste!

Kao uvjet u naredbi “**if**” često se koriste *logičke promjenljive*, s obzirom da njihov tip savršeno odgovara tipu izraza koji se očekuje u “**if**” naredbi. Tako, ukoliko imamo logičku promjenljivu “**u_dugovima**” koja čuva informaciju o tome da li je korisnikov tekući račun u dugovanju ili u primanju, sasvim normalno možemo napisati konstrukciju poput sljedeće:

```
if(u_dugovima) cout << "Račun u dugovanju\n";
else cout << "Račun u primanju\n";
```

Strukture grananja ćemo detaljnije ilustrirati na nekoliko konkretnih primjera. Prvo slijedi jedan jednostavniji primjer. Neka kompanija želi da kupi novi itison za svoje kancelarije. Dva prodavača itisona su dali svoje ponude kako slijedi:

- | | |
|-------------|---|
| Prodavač 1: | 24.50 KM po kvadratnom metru postavljenog itisona (iznos uključuje cijenu itisona i postavljanja itisona) |
| Prodavač 2: | 12.50 KM po kvadratnom metru itisona plus fiksni iznos od 400 KM (neovisno od kvadrature) |

Sljedeći program na ulazu prihvata dimenzije jedne kancelarije (pretpostavimo da je tlocrt kancelarija pravougaonog oblika), računa cijene obe ponude, ispisuje ih i preporučuje jeftinijeg prodavača:

```
#include <iostream>

using namespace std;

int main() {
    const double JedinicnaCijena1(24.50);
    const double JedinicnaCijena2(12.50);
    const double FiksnaCijena2(400);
    double duzina, sirina;
    cout << "Unesi širinu kancelarije u metrima: ";
    cin >> sirina;
    cout << "Unesi dužinu kancelarije u metrima: ";
    cin >> duzina;
    double povrsina = duzina * sirina;
    double cijena_1 = JedinicnaCijena1 * povrsina;
    double cijena_2 = JedinicnaCijena2 * povrsina + FiksnaCijena2;
    cout << "Prodavač 1 će vam naplatiti " << cijena_1 << " KM.\n"
        "Prodavač 2 će Vam naplatiti " << cijena_2 << " KM.\n"
        "Preporučujem Vam ";
    if (cijena_1 < cijena_2) cout << "prvog";
    else cout << "drugog";
    cout << " prodavača, jer je jeftiniji.\n";
    return 0;
}
```

Još jedan interesantan primjer korištenja naredbe “**if**” je program koji rješava kvadratnu jednačinu, bez obzira da li su njena rješenja realna ili kompleksna, ali ovaj put bez upotrebe tipa “**complex**”. Kompleksna rješenja se i ovdje ispisuju kao parovi (*re*, *im*), pri čemu je to ovaj put urađeno “vještacki”, tj. ručnim ispisivanjem zagrada, zareza itd. Ovaj put se realna rješenja ispisuju kao *čisto realni brojevi* a ne kao parovi:

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

```

using namespace std;

int main() {
    double a, b, c;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;
    double d = b * b - 4 * a * c;
    if(d >= 0) {
        double x1 = (-b - sqrt(d)) / (2 * a);
        double x2 = (-b + sqrt(d)) / (2 * a);
        cout << "x1 = " << x1 << "\nx2 = " << x2 << endl;
    }
    else {
        double re = -b / (2 * a);
        double im = abs(sqrt(-d)) / (2 * a));
        cout << "x1 = (" << re << "," << -im << ") \n"
            "x2 = (" << re << "," << im << ") \n";
    }
    return 0;
}

```

Ipak, ranije napisani program koji koristi tip “complex” ima tu prednost što i koeficijenti mogu biti kompleksni brojevi. Kao alternativu ovom rješenju, mogli smo koristiti tip “complex”, ali da u slučaju kada je diskriminanta veća od nule uzmememo samo realni dio rješenja, primjenom funkcije “real” (imaginarni dio je tada svakako jednak nuli). Pisanje ovakvog rješenje ostavljamo čitateljima i čitateljkama kao korisnu vježbu.

Interesantno je također primijetiti da promjenljive “x1”, “x2”, “re” i “im” imaju veoma ograničeno vrijeme života (svaka vrijedi samo unutar bloka u kojem je deklarirana). Moguće je formirati i sljedeće rješenje, u kojem se u jednom bloku promjenljive “x1” i “x2” deklariraju kao realne, a u drugom bloku kao kompleksne. Bez obzira na identična imena, radi se o *različitim promjenljivim*, s obzirom da svaka od njih živi samo unutar bloka u kojem je deklarirana:

```

#include <iostream>
#include <cmath>
#include <complex>

using namespace std;

int main() {
    double a, b, c;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;
    double d = b * b - 4 * a * c;
    if(d >= 0) {
        double x1 = (-b - sqrt(d)) / (2 * a);
        double x2 = (-b + sqrt(d)) / (2 * a);
        cout "x1 = " << x1 << "\nx2 = " << x2 << endl;
    }
}

```

```

    else {
        complex<double> dcomp = d;
        complex<double> x1 = (-b - sqrt(dcomp)) / (2 * a);
        complex<double> x2 = (-b + sqrt(dcomp)) / (2 * a);
        cout "x1 = " << x1 << "\nx2 = " << x2 << endl;
    }
    return 0;
}

```

Ovdje je bitno ukazati na jedan problem koji se javlja kod poređenja *realnih vrijednosti*, koji je možda trebalo istaći još pri uvođenju pojma logičkih izraza. Naime, zbog već opisanih problema gubitka tačnosti koji je karakterističan za realne vrijednosti, veoma je rizično dvije realne vrijednosti ispitivati na *tačnu jednakost*, jer zbog minornih grešaka u zaokruživanju uzrokovanih gubitkom tačnosti (tj. činjenicom da se realne vrijednosti pamte samo sa ograničenom tačnošću), dvije vrijednosti koje bi trebale da budu jednake mogu se *manifestirati kao različite*. Na primjer, zamislimo da želimo da napravimo program koji će ispitati da li trojka unesenih brojeva (a, b, c) predstavlja stranice pravouglog trougla, uz pretpostavku da su a i b katete, a c hipotenuza. Naravno, u načelu je potrebno samo provjeriti da li je $a^2 + b^2 = c^2$. To nas navodi da probamo napisati sljedeći programski isječak:

```

double a, b, c;
cout << "Unesite stranice trougla (najdužu unesite posljednju): ";
cin >> a >> b >> c;
if(c * c == a * a + b * b) cout << "Trougao je pravouglog\n";
else cout << "Trougao nije pravouglog\n";

```

Ukoliko testiramo ovaj programski na trojki brojeva (3, 4, 5) koje obrazuju stranice pravouglog trougla, biće zaista ispisano da ovi brojevi formiraju stranice pravouglog trougla. Međutim, za trojku brojeva (0.3, 0.4, 0.5) koja također obrazuje stranice pravouglog trougla program će ispisati da ne čini stranice pravouglog trougla, tj. dobićemo lažan odgovor! Naime, brojevi 0.3, 0.4 i 0.5 pamte se u memoriji kao brojevi u pokretnom zarezu sa mantisom u binarnom brojnom sistemu. Međutim, od pomenuta tri broja, samo mantsa broja 0.5 ima konačno mnogo decimala u binarnom brojnom sistemu, odakle slijedi da se brojevi 0.3 i 0.4 u memoriji pamte samo *približno* (doduše sa veoma velikom, ali ne i potpunom tačnošću – probate li ispisati broj 0.4 postavljajući preciznost ispisa na 30 decimala, dobićete vrijednost 0.400000000000000022204460492503 ukoliko kompjajler sa kojim radite koristi IEEE 754 standard zapisa realnih brojeva). Zbog te činjenice, rezultati izračunavanja izraza $0.3^2 + 0.4^2$ i 0.5^2 se neznatno razlikuju, zbog čega dobijamo pogrešan odgovor. Da bi stvari izgledale još čudnije, razmotrimo sljedeći programski isječak:

```

double a(0.3), b(0.4), c(0.5);
cout << c * c << " " << a * a + b * b << endl;
if(c * c == a * a + b * b) cout << "Prikazane vrijednosti su jednake\n";
else cout << "Prikazane vrijednosti su različite\n";

```

Ovaj isječak će prvo na ekranu ispisati dvije *jednake* vrijednosti (0.25), koje će nakon toga testirati operatorom “ $=$ ” i utvrditi da su one zapravo *različite*! U čemu je zapravo zvrčka? Izračunate vrijednosti ovih izraza zaista *nisu iste*, ali se razlikuju tek negdje na recimo sedamnaestoj decimali, a rezultati se nikada ne ispisuju sa tolikim brojem decimalnih mjesta. To što prikazane vrijednosti *izgledaju iste* pri ispisu na ekran, ne znači da one zaista *jesu iste* u memoriji računara!

Opisani problem može biti izvor velikih frustracija pri radu sa realnim vrijednostima. Kao pouku treba prihvati činjenicu da dvije realne vrijednosti *nikada* ne treba ispitivati na jednakost. Kako onda riješiti opisani problem? Primijetimo prvo da je (matematički) iskaz $x=y$ u suštini jednak iskazu oblika $x-y=0$. Računar zapravo tako i vrši poređenje dvije vrijednosti: oduzme ih, a zatim provjerava da li je rezultat

oduzimanja jednak nuli (što je tehnički mnogo lakše utvrditi nego direktno porediti dvije vrijednosti). Ideja je sada da iskaz oblika $x-y=0$ zamijenimo slabijim iskazom oblika $|x-y|<\epsilon$, gdje je ϵ neki mali broj (prag “tolerancije”), recimo 10^{-5} . Ova ideja iskorištena je u sljedećem programskom isječku, koji radi korektno i za trojku brojeva (0.3, 0.4, 0.5):

```
double a, b, c;
const double Eps(1e-5);
cout << "Unesite stranice trougla (najdužu unesite posljednju): ";
cin >> a >> b >> c;
if(abs(c * c - a * a - b * b) < Eps) cout << "Trougao je pravougli\n";
else cout << "Trougao nije pravougli\n";
```

Ni ovo rješenje nije savršeno: izabrana tolerancija $\epsilon=10^{-5}$ može biti prevelika ukoliko su stranice trougla veoma male. Na primjer, sigurno je da ova tolerancija nije dobra ukoliko su same stranice trougla istog reda veličine! Znatno bolje rješenje je učiniti toleranciju *proporcionalnom* učesnicima u poređenju, tj. uvjet $|x-y|<\epsilon$ zamijeniti uvjetom oblika $|x-y|<\epsilon|x|$ (drugim riječima, ϵ više nije *apsolutna*, već *relativna* tolerancija). Ovo je demonstrirano u sljedećem programskom isječku:

```
double a, b, c;
const double Eps(1e-5);
cout << "Unesite stranice trougla (najdužu unesite posljednju): ";
cin >> a >> b >> c;
if(abs(c * c - a * a + b * b) < Eps * abs(c * c))
    cout << "Trougao je pravougli\n";
else cout << "Trougao nije pravougli\n";
```

Na kraju možemo izvesti zaključak da pri poređenju dvije realne vrijednosti “ x ” i “ y ” nikada ne treba koristiti prostu konstrukciju “ $x == y$ ”, već isključivo konstrukciju poput

$$\text{abs}(x - y) < \text{Eps} * \text{abs}(x)$$

gdje je “ Eps ” neka pogodno odabrana konstanta (relativna tolerancija). Vrijednost 10^{-5} pokazuje se dobrom za većinu praktičnih primjena.

Činjenica da se numeričke vrijednosti upotrijebljene unutar uvjeta naredbe “**if**” automatski konvertiraju u logičke vrijednosti, otvara mogućnost za brojne “zloupotrebe” (tj. pisanje potpuno zbumujućih programa), ali može dovesti i do pojave fatalnih grešaka koje se teško otkrivaju. Najčešći problem nastaje uslijed zamjene operatora “ $=$ ” i “ $==$ ” koji su, kao što smo već istakli, posve različiti. Zamislimo, na primjer, da je programer htio da se neka naredba izvrši pod uvjetom da je promjenljiva “ a ” jednaka zbiru promjenljivih “ b ” i “ c ”. Ispravna naredba za obavljanje ovog zadatka glasila bi:

```
if(a == b + c) cout << "To je to";
```

Prepostavimo, međutim, da je programer greškom napisao sljedeću naredbu:

```
if(a = b + c) cout << "To je to";
```

Kompajler neće prijaviti nikakvu grešku, jer nismo napisali ništa što u jeziku C++ nije dozvoljeno (osnovna težina jezika C++ je upravo u tome što je u njemu dozvoljeno *previše* stvari). Efekat ovakve naredbe će biti da će prvo promjenljivo “ a ” biti *dodijeljena* vrijednost zbira promjenljivih “ b ” i “ c ” (dakle, vrijednost promjenljive “ a ” će se *promijeniti*), a zatim, ako je dodijeljena vrijednost *različita od nule* (sjetimo se da je *rezultat* operatora dodjele upravo *dodijeljena vrijednost*) naredba koja slijedi iza “**if**” (ispis teksta “To je to”) biće izvršena. Dakle, jedino ako su “ a ” i “ b ” bili jednaki po modulu i suprotni po znaku, tekst “To je to” neće biće isписан. Vrijednost promjenljive “ a ” biće *promijenjena* u

svakom slučaju. Čak i ako je programer željeo da postigne baš to, najtoplijе mu se preporučuje da radije napiše sljedeće naredbe:

```
a = b + c;  
if(a != 0) cout << "To je to";
```

Ovako je mnogo jasnije šta je programer htio da kaže.

Greška koja nastaje uslijed zamjene operatora “==” sa “=” toliko je česta, da neće biti na odmet da navedemo još jedan primjer. Pogledajmo slijedeću naredbu:

```
if(a = 5) cout << "Vrijednost promjenljive a je 5";
```

Sasvim je izvjesno da je programer željeo da testira da li je vrijednost promjenljive ”a” jednaka 5, ali tada je trebao pisati “==” a ne “=”. Ovako, dobijamo dvije neželjene posljedice:

- Vrijednost promjenljive ”a” će *postati* 5, kakva god da je bila prije;
- Tekst ”Vrijednost promjenljive a je 5” ispisaće se *uvijek*, neovisno od *ranije vrijednosti* promjenljive ”a”, jer je $5 \neq 0$.

Kao pravilo treba uzeti da se operator “==” izuzetno rijetko koristi unutar uvjeta naredbe “**if**”, tako da ako imate program koji koristi naredbu “**if**”, dobro provjerite da li ste slučajno koristili “=” umjesto “==”. S druge strane, operator “==” se relativno rijetko koristi izvan naredbi “**if**” i ”**while**”, o kojoj će kasnije biti riječi.

Unutar naredbe “**if**” mogu se naravno koristiti i složeniji uvjeti, formirani pomoću logičkih operatora, kao u sljedećim primjerima:

```
if(starost >= 16 && starost < 65) cout << "Možete se zaposliti\n";  
if(starost < 16 || starost >= 65) cout << "Ne možete se zaposliti\n";  
if(!(starost > 65)) cout << "Ne možete imati penziju\n";
```

Uvjeti često sadrže znakovne promjenljive, kao u sljedećoj konstrukciji:

```
char odgovor;  
cout << "Da li želite to_i_to? "  
cin >> odgovor;  
if(odgovor == 'D') cout << "Evo Vam to_i_to";
```

Primijetimo da su 'D' i 'd' dvije posve različite znakovne konstante (jedna ima vrijednost “68” a druga “100”, prema ASCII standardu), tako da ako u gornjem primjeru sa tastature unesemo malo slovo “d”, tekst ”to_i_to” neće biti ispisani, jer uvjet unutar naredbe “**if**” neće biti tačan. Najjednostavniji (mada ne i najefikasniji) način da se riješi ovaj problem je da pišemo naredbu poput

```
if(odgovor == 'D' || odgovor == 'd') cout << "Evo Vam to_i_to";
```

Alternativno, možemo iskoristiti funkciju ”**toupper**” iz biblioteke ”**cctype**” (ne zaboravimo pri tome uključiti zaglavje ove biblioteke u program):

```
if(toupper(odgovor) == 'D') cout << "Evo Vam to_i_to";
```

Na sličan način možemo vršiti razna ispitivanja sa znakovnim promjenljivim, kao u sljedećem primjeru:

```
char znak;
```

```

cout << "Unesite neki znak: ";
cin >> znak;
if(znak >= 'A' && znak <= 'Z') cout << "Unijeli ste veliko slovo\n";

```

Biblioteka “ctype” sadrži nekoliko veoma korisnih funkcija za ispitivanje prirode znaka koji im se proslijede kao argument. Sve ove funkcije testiraju da li znak pripada određenoj skupini znakova, i ukoliko ne pripada, kao rezultat daju *nulu*, a ukoliko pripada, kao rezultat daju neku vrijednost *različitu od nule* (nazovimo je *ne-nula*). Standard ne predviđa koja će vrijednost biti tačno vraćena u slučaju pripadnosti, ali to nije ni bitno, s obzirom da su ove funkcije predviđene da se koriste isključivo unutar uvjeta, tako da se svaka vrijednost različita od nule konvertira u vrijednost “**true**”. Može se postaviti pitanje zbog čega ove funkcije nisu napravljene tako da vraćaju vrijednosti “**true**” i “**false**”, ili barem “0” i “1”. Razlog leži u efikasnosti: njihova implementacija može biti mnogo efikasnija ukoliko se *ne vodi računa* o tačnoj vrijednosti koja će biti vraćena. Slijedi prikaz najvažnijih funkcija ove vrste iz biblioteke “ctype”:

<code>islower(c)</code>	Daje ne-nulu ako je <i>c</i> malo slovo
<code>isupper(c)</code>	Daje ne-nulu ako je <i>c</i> veliko slovo
<code>isalpha(c)</code>	Daje ne-nulu ako je <i>c</i> slovo
<code>isdigit(c)</code>	Daje ne-nulu ako je <i>c</i> cifra
<code>isalnum(c)</code>	Daje ne-nulu ako je <i>c</i> cifra ili slovo
<code>ispunct(c)</code>	Daje ne-nulu ako je <i>c</i> znak interpunkcije
<code>isgraph(c)</code>	Daje ne-nulu ako je <i>c</i> ispisivi znak (cifra, slovo ili znak interpunkcije)
<code>isspace(c)</code>	Daje ne-nulu ako je <i>c</i> praznina (razmak, tabulator ili oznaka kraja reda)
<code>iscntrl(c)</code>	Daje ne-nulu ako je <i>c</i> kontrolni znak (npr. oznaka kraja reda)

Stoga bismo prethodni primjer mogli napisati jednostavnije, korištenjem funkcije “`isupper`”:

```

char znak;
cout << "Unesite neki znak: ";
cin >> znak;
if(isupper(znak)) cout << "Unijeli ste veliko slovo\n";

```

Logički operatori se efikasno koriste zajedno sa logičkim promjenljivim. Na primjer, ukoliko imamo sljedeće deklaracije:

```

int ocjena, starost;
bool ispravna_ocjena, penzioner, dijete;

```

tada su sljedeće konstrukcije sasvim smislene:

```

ispravna_ocjena = (ocjena >= 5) && (ocjena <= 10);
// Zagrade nisu obavezne

dijete = starost < 18;
penzioner = (starost >= 65); // Ni ove zagrade nisu obavezne
if(penzioner || dijete) cout << "Platite pola cijene za prevoz\n";

```

Ispravno korištenje logičkih promjenljivih i logičkih operatora najbolje ilustrira sljedeći primjer. Po pravilu koje je na snazi u Velikoj Britaniji, na izborima imaju pravo glasanja samo osobe koje imaju 18 godina ili više, pod uvjetom da nisu u zatvoru, da nisu neuračunljive i da nisu “plemenite krvi” (tj. da nisu članovi kraljevske porodice ili da posjeduju neku plemićku titulu kao “vojvoda” ili “lord”). Sljedeći program postavlja korisniku nekoliko pitanja, i nakon toga mu saopštava da li ima pravo glasa po britanskom sistemu prava na glasanje:

```

#include <iostream>
#include <cctype>

using namespace std;

int main() {
    int starost;
    char odgovor;
    cout << "Molim Vas, unesite svoju starost: ";
    cin >> starost;
    cout << "Da li ste plemenite krvi? (D/N): ";
    cin >> odgovor;
    bool plemic = toupper(odgovor) == 'D';
    cout << "Da li ste u zatvoru? (D/N): ";
    cin >> odgovor;
    bool zatvorenik = toupper(odgovor) == 'D';
    cout << "Da li ste neuračunljivi? (D/N): ";
    cin >> odgovor;
    bool neuracunljiv = toupper(odgovor) == 'D';
    cout << endl;
    bool pravo_glasa = (starost >= 18)
        && !(plemic || zatvorenik || neuracunljiv);
    if(pravo_glasa) cout << "Imate pravo glasa\n";
    else cout << "Nemate pravo glasa\n";
    return 0;
}

```

Naredbe koje se nalaze unutar naredbe “**if**”, bilo u direktnom, bilo u alternativnom toku akcija (tj. iza ključne riječi “**else**”), mogu i same biti naredbe grananja (tj. mogu ponovo sadržavati ključne riječi “**if**” ili “**else**”). U tom slučaju govorimo o *ugniježdenim naredbama grananja*. Situacija kada se “ugniježđena” naredba grananja nalazi u *alternativnom toku akcija*, mnogo je jasnija, i ne stvara nikakve nedoumice. Na primjer, takva situacija se javlja u sljedećem primjeru:

```

if(starost <= 16)
    cout << "Dijete";
else
    if(starost < 65)
        cout << "Zrela osoba";
    else
        cout << "Penzioner";

```

U slučaju kada se ugniježđena naredba grananja nalazi u *direktnom toku akcija*, mogu se javiti izvjesne nedoumice po pitanju koja se ključna riječ “**else**” vezuje sa kojom ključnom riječju “**if**”. U tom slučaju vrijedi pravilo da se svaka ključna riječ “**else**” odnosi na najbližu ključnu riječ “**if**” koja do tada nije bila pridružena nekoj drugoj riječi “**else**”. Šta se pod ovim misli, najbolje se vidi iz sljedećeg šematskog prikaza:

```

if(uvjet_1)
    if(uvjet_2)
        akcija_1;
    else
        akcija_2;
else
    akcija_3;

```

Strelicama je jasno označeno koja se ključna riječ “**else**“ odnosi na koju ključnu riječ “**if**“.

Kod korištenja ugnježdenih naredbi grananja mogu nastati veliki problemi kao posljedica nepažljivosti. Pretpostavimo da je programer napisao sljedeću konstrukciju:

```
if (uvjet_1)
    if (uvjet_2)
        akcija_1;
else
    akcija_2;
```

Na osnovu kako je programer potpisao instrukcije jednu ispod druge, može se naslutiti da je on željeo sljedeće: “Ako je *uvjet_1* ispunjen, tada ako je i *uvjet_2* ispunjen, obavi akciju *akcija_1*, a ako *uvjet_1* nije ispunjen, obavi akciju *akcija_2*”. Prepostavljenu namjeru programera možemo opisati sljedećom tabelom:

<u>uvjet 1</u>	<u>uvjet 2</u>	<u>Akcija</u>
netačan	netačan	<i>akcija_2</i>
netačan	tačan	<i>akcija_2</i>
tačan	netačan	nikakva
tačan	tačan	<i>akcija_1</i>

Međutim, programer neće postići namjeravani efekat, zbog toga što je ključna riječ “**else**“ povezana sa najbližom ključnom riječju “**if**“, a ne sa onom ključnom riječju “**if**“ ispod koje je ključna riječ “**else**“ potpisana (već smo rekli da je potpisivanje samo estetski efekat, koji ne utječe na izvršavanje programa). Zbog toga će ove naredbe biti shvaćene kao “ako je *uvjet_1* ispunjen tada ako je *uvjet_2* ispunjen radi akciju *akcija_1*, a ako *uvjet_2* nije ispunjen (podrazumijevajući pri tome da je *uvjet_1* ispunjen) radi akciju *akcija_2*; ako *uvjet_1* nije ispunjen, ne dešava se ništa”. Ovaj efekat možemo ilustrirati sljedećom tabelom:

<u>uvjet 1</u>	<u>uvjet 2</u>	<u>Akcija</u>
netačan	netačan	nikakva
netačan	tačan	nikakva
tačan	netačan	<i>akcija_2</i>
tačan	tačan	<i>akcija_1</i>

Skoro da je izvjesno da ovo nije bila namjera programera, jer da jeste, on bi vjerovatno instrukcije potpisao ovako:

```
if (uvjet_1)
    if (uvjet_2)
        akcija_1;
else
    akcija_2;
```

Znamo da je potpisivanje samo estetski efekat, tako da i prethodna i ova sekvenca imaju isto dejstvo. Kako bismo onda ostvarili željenu namjeru programera? Postoji još jedno pravilo vezano za povezivanje ključnih riječi “**if**“ i “**else**“, a to je da se povezivanje vrši uvijek unutar *istog bloka* (tj. unutar iste sekvence omeđene vitičastim zagradama). Drugim riječima, nikada se neće desiti da se neka ključna riječ “**else**“ iz unutrašnjeg bloka poveže sa nekom ključnom riječju “**if**“ koja spoljašnjem bloku, ili nekom sasvim drugom bloku. Stoga, kompletno pravilo za povezivanje ključnih riječi “**if**“ i “**else**“ glasi:

- Svaka ključna riječ “**else**“ naredba odnosi se na najbližu ključnu riječ “**if**“ unutar istog bloka u kojem je i sama upotrebljena, i koja do tada nije bila povezana sa nekoj drugom ključnom rječju “**else**“ naredbom. Pri tome se povezivanje obavlja počev od najbližih parova “**if**“ – “**else**“ ka udaljenijim.

Odavde slijedi ispravan način da se ostvari programerova namjera:

```
if(uvjet_1) {
    if(uvjet_2)
        akcija_1;
}
else
    akcija_2;
```

Ovdje je ključna riječ “**else**“ povezana sa ispravnom ključnom rječju “**if**“ (a ne sa najbližom), jer najbliža ključna riječ “**if**“ ne pripada istom bloku. Zbog toga je dobra praksa kod korištenja ugnježdenih naredbi izbora uvijek vitičastim zagradama (odnosno formiranjem blokova) *eksplicitno* naznačiti šta sa čim treba biti povezano. Tako se može pisati (iako bi se i bez vitičastih zagrada postigao isti efekat):

```
if(uvjet_1) {
    if(uvjet_2)
        akcija_1;
}
else
    akcija_2;
}
```

Ugnježdene naredbe izbora ilustriraćemo opet na primjeru kvadratne jednačine, ali sada ćemo realizirati ispis kompleksnih rješenja u obliku kakav je uobičajen u matematici (npr. i , $-4i$, $2+3i$ itd). Program je nešto duži, zato se potrudite da shvatite kako radi:

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

using namespace std;

int main() {
    double a, b, c;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;
    double d = b * b - 4 * a * c;
    if(d >= 0) {
        double x1 = (-b - sqrt(d)) / (2 * a);
        double x2 = (-b + sqrt(d)) / (2 * a);
        cout "x1 = " << x1 << "\nx2 = " << x2 << endl;
    }
    else {
        double re = -b / (2 * a);
        double im = abs(sqrt(-d)) / (2 * a));
        cout "x1 = ";
        if(re != 0) cout << re;
```

```

        cout << " - ";
        if(im != 1) cout << im;
        cout << " i\nx2 = ";
        if(re != 0) cout << re << " + ";
        if(im != 1) cout << im;
        cout << " i\n";
    }
    return 0;
}

```

Dubina gniježdenja naredbi grananja može biti proizvoljno velika, pri čemu se više od dva nivoa gniježdenja obično koristi samo ukoliko želimo ostvariti višestruki izbor, kao u primjeru sljedećeg programa koji kao ulaz traži ocjenu sa ispita, i daje studentu odgovarajući komentar:

```

#include <iostream>
using namespace std;

int main() {
    int ocjena;
    cout << "Unesi svoju ocjenu sa ispita: ";
    cin >> ocjena;
    if(ocjena < 5 || ocjena > 10)
        cout << "Ocjena nije ispravna!\n";
    else
        if(ocjena >= 9)
            cout << "Odlično!\n";
        else
            if(ocjena >= 7)
                cout << "Dobro!\n";
            else
                if(ocjena >= 6) {
                    cout << "Nije tako loše\n";
                    cout << "...ali možda trebate više raditi!\n";
                }
                else
                    cout << "Više sreće sljedeći put!\n";
    return 0;
}

```

Ovakav način potpisivanja kod višestrukog izbora nije najpogodniji, jer dovodi do tzv. "stepenastog" izgleda programa (naredbe koje su duboko ugniježdene odmiču se previše udesno, naročito kod velikih dubina gniježdenja). Zbog toga se, u slučajevima višestrukog izbora (za višestruki izbor je karakteristično da iza svake ključne riječi "**else**", osim posljednje, dolazi ponovo ključna riječ "**if**") često ključna riječ "**else**" i naredna ključna riječ "**if**" pišu u istom redu:

```

#include <iostream>
using namespace std;

int main() {
    int ocjena;
    cout << "Unesi svoju ocjenu sa ispita: ";
    cin >> ocjena;
    if(ocjena < 5 || ocjena > 10) cout << "Ocjena nije ispravna!\n";
    else if(ocjena >= 9) cout << "Odlično!\n";
    else if(ocjena >= 7) cout << "Dobro!\n";
    else if(ocjena >=6) {

```

```

        cout << "Nije tako loše\n";
        cout << "...ali možda trebate više raditi!\n";
    }
else cout << "Više sreće sljedeći put!\n";
return 0;
}

```

Na ovaj način ključna riječ “**else**“ i naredna ključna riječ “**if**“ tvore karakterističnu tzv. “**else if**“ strukturu.

Radi bolje ilustracije višestrukog izbora, navedimo i sljedeći primjer. Željezničke kompanije nekih evropskih zemalja naplaćuju karte na sljedeći način:

Djeca (ispod 16 godina)	pola cijene
Penzioneri (60 godina i stariji)	besplatno
Odrasli (od 16 do 60 godina)	puna cijena

Sljedeći program na ulazu prihvata punu cijenu karte, kao i godine starosti putnika koji želi da putuje, a na izlazu daje cijenu karte koju putnik treba da plati.

```

#include <iostream>
using namespace std;

int main() {
    cout << "Unesi punu cijenu vozne karte u EUR: ";
    double cijena, starost;
    cin >> cijena;
    cout << "Koliko imate godina? ";
    cin >> starost;
    if(starost >= 60) cout << "Možete putovati besplatno. ";
    else if(starost < 16)
        cout << "Plaćate pola cijene, tj. " << cijena / 2 << " EUR.\n";
    else cout << "Plaćate punu cijenu, tj. " << cijena << " EUR.\n";
    return 0;
}

```

U prethodnom poglavlju smo istakli da se ternarni operator “`? :`” može uvijek izbjegći korištenjem naredbe “**if**”. Razmotrimo, na primjer sljedeće primjere iz prethodnog poglavlja u kojima je korišten ternarni operator “`? :`”:

```

b = (a > 0) ? a : -a;
cout << ((x >= 0) ? "Broj je nenegativan" : "Broj je negativan");
sgn = (x >= 0) ? 1 : ((x == 0) ? 0 : -1);

```

Svi razmotreni primjeri mogu se napisati uz pomoć naredbi grananja na sljedeći način. Rezultirajuće naredbe su nešto duže, ali su u svakom slučaju neuporedivo razumljivije:

```

if(a > 0) b = a;
else b = -a;

if(x >= 0) cout << "Broj je nenegativan";
else cout << "Broj je negativan";

if(x >= 0) sgn = 1;

```

```

else if(x == 0) sgn = 0;
else sgn = -1;

```

Interesantno je da se kao uvjet unutar naredbe “**if**” može upotrijebiti *objekat ulaznog toka* “`cin`”. U tom slučaju se smatra da je objekat ulaznog toka “tačan” ukoliko je tok u *ispravnom stanju*, a “netačan” ukoliko je tok u *neispravnom stanju*. Ulazni tok može dospjeti u neispravno stanje na primjer u slučaju da se očekuje unos broja sa tastature, a umjesto broja korisnik unese znakove koji ne predstavljaju broj. Ovo je ilustrirano sljedećim primjerom:

```

int broj;
cout << "Unesite cijeli broj: ";
cin >> broj;
if(cin) cout << "Unijeli ste broj " << broj << endl;
else cout << "Varate, niste uopće unijeli broj!\n";

```

Takoder, nad objektom ulaznog toka može se primijeniti i operator negacije “`!`”. Tako je izraz “`!cin`” tačan ukoliko je tok u neispravnom stanju, a netačan u suprotnom. Stoga smo prethodni primjer mogli napisati i ovako:

```

int broj;
cout << "Unesite cijeli broj: ";
cin >> broj;
if(!cin) cout << "Varate, niste uopće unijeli broj!\n";
else cout << "Unijeli ste broj " << broj << endl;

```

O testiranju ulaznog toka na ispravnost govorićemo detaljnije kada upoznamo naredbe ponavljanja, koje će omogućiti da u slučaju neispravnog unosa zahtijevamo od korisnika da *ponovi unos*.

U jeziku C++ postoji još jedan način za realizaciju višestrukog izbora, koji se zasniva na naredbi “**switch**”. Ova naredba omogućava izbor izvršavanja *jedne od nekoliko mogućih alternativa*, pri čemu je izbor zasnovan na vrijednosti jednog izraza koji se naziva *izborni izraz* (engl. *case expression*). U najopćenitijem slučaju, naredba “**switch**” ima sljedeći oblik:

```

switch(izraz) {
    case konstanta_1:
        naredba_11
        naredba_12
        ...
    case konstanta_2:
        naredba_21
        naredba_22
        ...
        ...
    case konstanta_N:
        naredba_N1
        naredba_N2
        ...
    default:
        naredba_d1
        naredba_d2
        ...
}

```

Smisao ove naredbe je u sljedećem: izraz u zagradi iza ključne riječi “**switch**” se izračunava, i ako je njegova vrijednost jednakoj od konstanti navedenih iza labele (oznake) “**case**” unutar pripadnog bloka “**switch**”, izvršavanje programa se nastavlja od tog mesta. Ako vrijednost izraza nije jednakoj *ni jednoj* od konstanti navedenih iza ključne riječi “**case**”, izvršavanje se nastavlja od mjesta unutar bloka označenog labelom “**default**”, ako takva oznaka postoji. U suprotnom, čitav pripadni blok “**switch**” naredbe se preskače. Sve konstante navedene iza “**case**” unutar istog bloka moraju se međusobno razlikovati. Te konstante mogu biti *prave konstante* (uključujući i obične brojeve), kao i *konstantni izrazi*, ali ne i *neprave konstante*. Razmak između ključne riječi “**case**” i konstante je obavezan.

Izborni izraz može biti bilo kojeg *rednog* (engl. *ordinal*) tipa. To su svi tipovi između kojih je moguće uspostaviti *diskretni poredak*, zasnovan na pojmovima “*sljedeći*” i “*prethodni*” kao npr. tipovi “**int**”, “**char**” i “**bool**” (smatra se da je “**false**” < “**true**”), kao i svi tzv. *pobrojani* (engl. *enumerated*) tipovi koje ćemo upoznati kasnije, ali ne i realni tipovi poput “**double**” koji su prividno *kontinualni*). Na primjer, ako pretpostavimo da je “stupanj” promjenljiva tipa “**char**”, možemo napisati sljedeću “**switch**” naredbu, koja u zavisnosti od ocjene (stupnja) koju je kandidat dobio na nekom testu (u rasponu od 'A' do 'E') ispisuje koliko je poena kandidat imao na testu:

```
switch(stupanj) {
    case 'A':
        cout << "75% ili više\n";
        break;
    case 'B':
        cout << "65% - 74%\n";
        break;
    case 'C':
        cout << "55% - 64%\n";
        break;
    case 'D':
        cout << "40% - 54%\n";
        break;
    case 'E':
        cout << "manje od 40%\n";
}
```

Ako izborni izraz (u ovom slučaju to je samo promjenljiva “stupanj”) ima vrijednost npr. 'C', tada će se izvršavanje programa nastaviti od mjesta gdje piše “**case 'C'**”, tako da će na ekranu biti ispisani tekst “55% – 64%”. U ovom primjeru se također koristi ključna riječ “**break**” koja napušta blok “**switch**” naredbe, i pomoću koje sprečavamo da se izvršavaju naredbe navedene iza ostalih “**case**” oznaka. Da smo ispustili ključne riječi “**break**”, tj. da smo napisali sljedeću konstrukciju:

```
switch(stupanj) {
    case 'A':
        cout << "75% ili više\n";
    case 'B':
        cout << "65% - 74%\n";
    case 'C':
        cout << "55% - 64%\n";
    case 'D':
        cout << "40% - 54%\n";
    case 'E':
        cout << "manje od 40%\n";
}
```

tada bi se u slučaju da je promjenljiva “stupanj” imala vrijednost 'C', na ekranu ispisalo:

55% - 64%
40% - 54%
manje od 40%

odnosno, ne ono što smo (vjerovatno) željeli.

Nekada želimo da izvršimo istu akciju za više različitih vrijednosti izbornog izraza. To možemo postići navođenjem više “**case**” oznaka jednu iza druge. Na primjer, ako je “ocjena” promjenljiva tipa “**int**”, tada uz pretpostavku da su moguće ocjene od 1 do 10, a da su samo ocjene od 6 do 10 prolazne, možemo pisati:

```
switch(ocjena) {  
    case 10:  
        cout << "Odlično!\n";  
        break;  
    case 9:  
        cout << "Dobro!\n";  
        break;  
    case 8:  
    case 7:  
        cout << "Prosječno!\n";  
        break;  
    case 6:  
        cout << "Slabo!\n";  
        break;  
    case 5:  
    case 4:  
    case 3:  
    case 2:  
    case 1:  
        cout << "Veoma slabo!\n";  
}
```

Nekada se sve višestruke “**case**” oznake pišu jedna iza druge, što je samo stvar stila:

```
switch(ocjena) {  
    case 10:  
        cout << "Odlično!\n";  
        break;  
    case 9:  
        cout << "Dobro!\n ";  
        break;  
    case 8: case 7:  
        cout << "Prosječno!\n ";  
        break;  
    case 6:  
        cout << "Slabo!\n ";  
        break;  
    case 5: case 4: case 3: case 2: case 1:  
        cout << "Veoma slabo!\n ";  
}
```

U ovom primjeru smo mogli upotrebiti i oznaku “**default**”:

```
switch(ocjena) {  
    case 10:
```

```

        cout << "Odlično!\n ";
        break;
case 9:
    cout << "Dobro!\n ";
    break;
case 8: case 7:
    cout << "Prosječno!\n ";
    break;
case 6:
    cout << "Slabo!\n ";
    break;
default:
    cout << "Veoma slabo!\n ";
}

```

Često se, radi uštede prostora, za slučaj kada se iza “**case**” oznake nalazi samo jedna naredba i ključna riječ “**break**”, sva tri elementa (oznaka, naredba i ključna riječ “**break**”) pišu u istom redu:

```

switch(ocjena) {
    case 10: cout << "Odlično!\n "; break;
    case 9: cout << "Dobro!\n "; break;
    case 8: case 7: cout << "Prosječno!\n "; break;
    case 6: cout << "Slabo!\n "; break;
    default: cout << "Veoma slabo!\n ";
}

```

Naravno, i ovdje se samo radi o pitanju stila.

Slijedi još jedan ilustrativan primjer. Naredni program simulira jednostavan kalkulator, koji omogućava korisniku da unese broj, operator “+”, “-”, “*” ili “/”, i drugi broj (npr. “2+3”). Program tada računa i prikazuje rezultat tražene operacije nad traženim operandima:

```

#include <iostream>
using namespace std;

int main() {
    double prvi, drugi;
    char operacija;
    cin << prvi << operacija << drugi;
    switch(operacija) {
        case '+': cout << prvi + drugi << endl; break;
        case '-': cout << prvi - drugi << endl; break;
        case '*': cout << prvi * drugi << endl; break;
        case '/': cout << prvi / drugi << endl; break;
        default: cout << "Nedozvoljena operacija!\n ";
    }
    return 0;
}

```

Da bismo shvatili kako radi ovaj program, moramo se podsjetiti da se izdvajanje broja pomoću operatora “**>>**” iz ulaznog toka prekida na prvom znaku koji sigurno ne pripada broju. Tako, ako zadamo naredbu poput

```
cin >> prvi;
```

a sa tastature unesemo nešto poput “321*443”, tada će u promjenljivu “prvi” biti smješten broj “321”, jer se izdvajanje pomoću operatora “>>” prekida na znaku “*” koji nije sastavni dio broja. Druga činjenica bitna za razumijevanje je da se izdvajanje sljedećeg elementa nastavlja na mjestu gdje se završilo izdvajanje prethodnog elementa. Zbog toga će nakon naredbe

```
cin >> prvi >> operacija >> drugi;
```

za slučaj da je unos sa tastature bio također “321*443”, u promjenljivu “prvi” biti smješten broj “321”, u promjenljivu “operacija” znak ‘+’ (zapravo njegova šifra ako hoćemo da budemo posve precizni), a u promjenljivu “drugi” broj “443”. Uz ovo objašnjenje ostatak programa je dovoljno jasan.

U principu, naredba “**switch**” se uvijek može simulirati (nekad lakše, a nekad teže) pomoću višestrukih “**if**” – “**else**” struktura. Tako je, na primjer, sljedeći program, koji ne koristi “**switch**” naredbu, funkcionalno potpuno ekvivalentan prethodnom programu:

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

int main() {
    double prvi, drugi;
    char operacija;
    cin << prvi << operacija << drugi;
    if(operacija == '+') cout << prvi + drugi << endl;
    else if(operacija == '-') cout << prvi - drugi << endl;
    else if(operacija == '*') cout << prvi * drugi << endl;
    else if(operacija == '/') cout << prvi / drugi << endl;
    else cout << "Nedozvoljena operacija!\n";
    return 0;
}
```

U ovom primjeru smo čak dobili kraći program bez upotrebe naredbe “**switch**”. Generalno je teško postaviti pravilo kada je bolje koristiti “**switch**” a kada višestruke “**if**” – “**else**” strukture. To bitno zavisi od strukture problema koji se rješava. Zapravo, osnovni razlog zbog kojeg je naredba “**switch**” uopće uvedena je *efikasnost*. Generalno se pomoću naredbe “**switch**” postiže veća efikasnost nego pomoću ugniježdenih “**if**” – “**else**” struktura. Naime, kod ugniježdenih “**if**” – “**else**” struktura, svaki od uvjeta se mora posebno izračunati i ispitati sve dok se ne najde na odgovarajući uvjet koji je ispunjen (u slučaju da niti jedan uvjet nije ispunjen, prethodno će se izračunati i ispitati *svi uvjeti*). S druge strane, pri upotrebi “**switch**” naredbe, izborni izraz se izračunava *samo jednom*, nakon čega se prosti vrši skok na odgovarajuću “**case**” labelu.

11. Naredbe ponavljanja

U svim dosada napisanim programima, napisane naredbe izvršavale su se *najviše jedanput*. Stoga nam u takvim situacijama programi i nisu bili osobito potrebni – sve probleme koje smo do sada rješavali lako bismo riješili uz pomoć običnog džepnog kalkulatora. Pravi smisao pisanja programa dolazi do izražaja tek kada treba iskazati algoritme koji zahtijevaju mehaničko *ponavljanje* (engl. *repetition*) određene skupine akcija. Na primjer, zamislimo da je potrebno izračunati sumu

$$S = \sum_{i=1}^{1000000} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{1000000}$$

Ovu sumu bismo *principijelno* također mogli izračunati uz pomoć džepnog kalkulatora, samo problem nastaje što u ovoj sumi imamo *milion sabiraka* (napomenimo da se ova suma ne može iskazati u nekom kompaktnom obliku, kao npr. suma aritmetičkog ili geometrijskog reda, već je zaista potrebno ručno izračunati i sabrati sve sabirke). Ukoliko bismo svake sekunde pritiskali po jedan taster na kalkulatoru, potrebno je preko pet mjeseci svakodnevnog osmočasovnog rada da bismo došli do rezultata (uz pretpostavku da nigdje ne pogriješimo, što je naravno praktično nemoguće). Međutim, uskoro ćemo vidjeti da je ovu sumu moguće izračunati za najviše nekoliko sekundi (ovisno od brzine računara) korištenjem sljedeće sekvence naredbi (čiji će smisao uskoro biti razjašnjen):

```
double S(0);
for(int i = 1; i <= 1000000; i++) S += 1. / i;
```

Ono što je bitno uočiti je da je pri računanju navedene sume stalno potrebno ponavljanje praktično *identičnih* akcija (računanje recipročne vrijednosti, i dodavanje izračunate vrijednosti na sumu prethodno sabranih članova sume). Ovakvi algoritmi realiziraju se upravljačkom strukturon koja se naziva *petlja* (engl. *loop*) ili *iteracija* (engl. *iteration*), koja predstavlja upravljačku strukturu koja omogućava *ponavljanje neke akcije* (ili skupine akcija).

Najgeneralniji oblik petlje nastaje u slučajevima kada ne znamo unaprijed koliko puta se neka akcija treba ponoviti. Za te slučajeve je karakteristična pojava specijalne fraze “**Sve dok**” (engl. “*While*”) u verbalnom prikazu algoritma koji ih opisuje. Na primjer, ovo je algoritam koji princ iz bajke koristi da pronađe Pepeljugu:

- *Uzmi cipelu;*
- **Sve dok** Pepeljuga nije nađena:
 - *Potraži prvu sljedeću djevojku;*
 - *Probaj može li obući cipelu;*
 - *Ako cipela odgovara:*
 - *Pepeljuga je nađena;*
- *Oženi se;*

Princ ne zna unaprijed koliko će djevojaka morati da ispita prije nego sto nađe pravu. Kada pri izvršavanju ovog algoritma nađemo na specijalnu frazu “**Sve dok**”, ispituje se tačnost uvjeta koji slijedi. Ukoliko je *uvjet ispunjen*, naredbe unutar petlje (prikazane uvučeno) se izvršavaju. Nakon što se sve naredbe unutar petlje izvrše jedanput, izvođenje petlje započinje ispočetka. Preciznije, uvjet se ispituje ponovo, i ukoliko je još uvijek ispunjen, naredbe unutar petlje se izvršavaju ponovo. Postupak se dalje ponavlja sve dok navedeni uvjet ne prestane biti tačan. Nakon toga, tijelo petlje se preskače, i izvršavanje algoritma se nastavlja od prve naredbe iza tijela petlje.

Analogna upravljačka struktura koja vrši ponavljanje skupine naredbi *sve dok je neki uvjet ispunjen* u jeziku C++ realizira se pomoću naredbe “**while**”. Ona se koristi u sljedećem obliku:

```
while (izraz) naredba
```

Primijetimo da je struktura naredbe “**while**” praktično identična strukturi naredbe “**if**”. Ipak, njihov smisao se *bitno razlikuje*. Slično kao naredbe “**if**”, i kod naredbe “**while**” izraz u zagradi bi trebao da predstavlja uvjet (odnosno trebao bi biti logičkog tipa), a prihvata se i izraz proizvoljnog numeričkog tipa (koji se tom prilikom konvertira u logički, po ranije opisanim pravilima). Ukoliko uvjet nije tačan, naredba navedena iza zagrada se preskače (i izvođenje programa se nastavlja od sljedeće naredbe), a u protivnom se izvršava. Po ovome se naredba “**while**” ponaša tačno kao i naredba “**if**”. Međutim, nakon što se naredba navedena iza zagrada izvrši, izraz (uvjet) se *ponovo izračunava*, i ukoliko je on i dalje tačan, navedena naredba se *ponovo izvršava*. Postupak se ponavlja *sve dok je uvjet naveden u zagradi tačan*. Tek kada uvjet *postane netačan*, program se nastavlja izvršavati od sljedeće naredbe. Dakle, kod naredbe “**if**”, pripadna naredba koja slijedi iza zagrada se izvršava *ako je uvjet tačan*, a kod naredbe “**while**”, pripadna naredba koja slijedi iza zagrada (za koju kažemo da čini *tijelo petlje*) se neprestano izvršava *sve dok je uvjet tačan*. Obratimo pažnju da naredba “**if**” *ne tvori petlju*!

Ukoliko želimo da ponavljamo veću skupinu naredbi, odnosno ukoliko želimo da se tijelo petlje sastoji od više naredbi, prosto ćemo naredbe koje čine tijelo *objediniti u blok*, s obzirom da smo rekli da se sa aspekta orkuženja u kojem se nalazi blok, čitav blok tretira kao *jedna naredba*. Kao ilustraciju naredbe “**while**”, napišimo sekvencu naredbi koja će ispisati sve prirodne brojeve od 1 do 100 (svaki u novom redu):

```
int i(1);
while(i <= 100) {
    cout << i << endl;
    i++;
}
```

U ovom slučaju, tijelo petlje čini blok koji se sastoji od dvije naredbe. Radikalniji C++ programer bi možda izbjegao potrebu za formiranjem bloka na sljedeći način:

```
int i(1);
while(i <= 100) cout << i++ << endl;
```

Kako se kod naredbe “**while**” tijelo petlje izvršava sve dok je navedeni uvjet ispunjen, slijedi da bi petlja imala smisla, makar negdje unutar njenog tijela trebala bi se nalaziti neka naredba koja će *promijeniti vrijednost barem neke od promjenljivih koje se pojavljuju unutar navedenog uvjeta*. U suprotnom, ukoliko je uvjet pri ulasku u petlju bio tačan, takav će sigurno i ostati, pa se petlja *nikad neće završiti*, odnosno tijelo petlje će se izvršavati *unedogled* (odnosno, dok na neki način nasilno ne prekinemo izvršavanje programa). Ovo je naročito česta greška koja se javlja u programima koji sadrže petlje, pogotovo kod početnika.

U nešto ekstremnijim slučajevima, moguće je formirati uvjete koji sadrže bočne efekte, i koji *mijenjaju vrijednosti promjenljivih koje se u njima nalaze*, tako da petlja može imati smisla čak i ukoliko se promjenljive nigdje ne mijenjaju unutar samog tijela petlje. Ovo je demonstrirano u sljedećem primjeru, koji radi isto što i prethodna dva primjera, a koji bi možda napisali ekstremistički nastrojeni C++ programeri:

```
int i(0);
```

```
while(++i <= 100) cout << i << endl;
```

Ovakvo pisanje naravno nije dobra programerska praksa. Ipak, uvjeti koji sadrže bočne efekte u nekim slučajevima se mogu efektno upotrijebiti ukoliko dobro pazimo šta radimo i zašto to radimo.

Sljedeći primjer prikazuje program koji omogućava korisniku da unosi nenegativne cijele brojeve i pamti ukupnu sumu unesenih brojeva. Kada se unese negativan broj, program ispisuje izračunatu sumu (ne računajući i taj negativan broj) i završava sa radom. Ideja programa se zasniva u uvođenju neke promjenljive (nazovimo je npr. "suma") koja će pamtitи *trenutnu sumu* (odnosno sumu *prethodno unesenih brojeva*), a čija će početna vrijednost biti nula. Svaki put kada unesemo novi broj, njega samo *nadodajemo* na trenutnu vrijednost sume. Postupak se ponavlja sve dok su uneseni brojevi nenegativni:

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

int main() {
    int broj;
    cout << "Unesite brojeve koji će se sabirati "
        "- negativan broj označava prekid.\n\n"
        "Unesite broj: ";
    cin >> broj;
    int suma(0);
    while(broj >= 0) {
        suma += broj;
        cout << "Unesite broj: ";
        cin >> broj;
    }
    cout << "\nSuma unesenih brojeva je: " << suma << endl;
    return 0;
}
```

Primijetimo da smo u ovom primjeru istu naredbu za unos morali pisati *dva puta*, jedanput *ispred petlje*, a drugi put *unutar petlje*. Razlog je u činjenici da se uvjet kod naredbe "**while**" ispituje na početku petlje, tako da odmah na početku moramo znati kakva je vrijednost promjenljive "broj" da bismo znali da li petlja treba uopće da se izvrši ili ne. Taj problem možemo riješiti sljedećim, funkcionalno ekvivalentnim, ali logički elegantnijim programom:

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

int main() {
    cout << "Unesite brojeve koji će se sabirati "
        "- negativan broj označava prekid.\n\n";
    int suma(0), broj(0);
    while(broj >= 0) {
        cout << "Unesite broj: ";
        cin >> broj;
        if(broj >= 0) suma += broj;
    }
    cout << "\nSuma unesenih brojeva je: " << suma << endl;
    return 0;
}
```

U ovom primjeru smo, na prvi pogled nepotrebno, inicijalizirali i promjenljivu "broj". Ova

inicijalizacija je neophodna iz sljedećeg razloga: ako promjenljivu "broj" ne inicijaliziramo, njena početna vrijednost će biti *slučajna* (nepredvidljiva). Kako se promjenljiva "broj" pojavljuje u uvjetu naredbe "**while**" prije nego što se njena vrijednost prvi put očita sa tastature, u slučaju da se dogodi da je slučajna početna vrijednost ove promjenljive *negativna*, petlja se neće izvršiti *niti jedanput*, jer će uvjet *odmah na početku biti netačan*. Ako se ovo dogodi, kao posljedicu ćemo imati potpuno neispravan rad programa: program nas neće pitati ni za jedan broj, i ispisaće kao krajnji rezultat nulu! Da stvar bude još gora, zavisno od slučaja, program će nekada raditi, a nekada neće. Podsjetimo se da smo još ranije naglasili da je nepredvidljiv rad programa (odnosno program koji nekada radi a nekada ne radi) tipičan simptom korištenja sadržaja promjenljivih kojima prethodno nije dodijeljena vrijednost!

U prethodnom programu koristili smo i naredbu "**if**" čiji je smisao jasan: ona sprečava da kada unesemo negativan broj on bude uračunat u sumu (pogledajte postavku zadatka). Naredbu "**if**" možemo izbjegći jednim elegantnim trikom: kako nula sabrana sa bilo kojim brojem daje taj isti broj, i kako su obje promjenljive "suma" i "broj" inicijalizirane na nulu, radiće i sljedeći program u kojem je, pomalo čudno, zamijenjen redoslijed sabiranja i unosa sa tastature:

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

int main() {
    cout << "Unesite brojeve koji će se sabirati ";
    "- negativan broj označava prekid.\n\n";
    int suma(0), broj(0);
    while(broj >= 0) {
        suma += broj;
        cout << "Unesite broj: ";
        cin >> broj;
    }
    cout << "\nSuma unesenih brojeva je: " << suma << endl;
    return 0;
}
```

Iskoristimo sada naredbu "**while**" za rješavanje problema računanja sume recipročnih vrijednosti svih prirodnih brojeva od 1 do n , pri čemu je n broj koji se unosi sa tastature (problem postavljen na početku ovog poglavlja je specijalan slučaj ovog problema za $n=1000000$). Rješenje ovog problema dato je sljedećim programom koji je dovoljno jasan sam po sebi (kasnije ćemo vidjeti da se isti problem može brže i elegantnije riješiti pomoću naredbe "**for**"):

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

int main() {
    int n;
    cout << "Unesite n: "
    cin >> n;
    double suma(0);
    int i(1);
    while(i <= n) {
        suma += 1. / i;
        i++;
    }
    cout << "Suma recipročnih vrijednosti brojeva od 1 do " << n
    << " iznosi " << suma << endl;
    return 0;
}
```

```
}
```

Obratimo pažnju na tačku unutar broja u izrazu “1. / i”. Ukoliko bismo izostavili ovu tačku, kao krajnji rezultat bismo dobili nulu, jer bi operator “/” bio shvaćen kao operator cjelobrojnog dijeljenja, s obzirom da je “i” cjelobrojna promjenljiva (tako da bi oba operanda bila cjelobrojnog tipa). Stoga još jednom ukazujemo na potrebu posebnog opreza pri korištenju operatora “/”!

Slično kao i kod naredbe “**if**”, česta greška prilikom korištenja naredbe “**while**” je *nenamjerno stavljanje tačka-zareza iza naredbe “while”*. Najgore je što u tom slučaju kompjajler neće prijaviti grešku, nego će program raditi, ali *neispravno*, jer će računar smatrati da se tijelo petlje sastoji od *prazne naredbe*, odnosno da je tijelo *petlje prazno* (petlja u kojoj se ne izvršava ništa). Naredbe koje smo željeli da budu u petlji u takvom slučaju će se zapravo nazaliti *izvan nje!* Vjerovatno će se program “zaglaviti” u mrtvoj (beskonačnoj) petlji, jer ako je uvjet bio ispunjen na početku, takav će ostati i zauvijek (osim ako možda ne sadrži bočne efekte), jer su se naredbe koje mijenjaju vrijednost promjenljivih koje učestvuju u uvjetu najvjerovaljnije nalazile unutar nekog bloka, koji zbog prijevremenog tačka-zareza nije shvaćen kao pripadni blok “**while**” petlje. Dakle, sljedeći niz naredbi će “zaglaviti” program:

```
int i(1);
while(i <= 10);           // Ovaj ";" je pogrešan
{
    cout << i << endl;
    i++;
}
```

Jedan od načina da se ovakvi problemi izbjegnu je navikavanje na (neobaveznu) praksu da se otvorena vitičasta zagrada koja započinje blok koji čini tijelo petlje piše odmah iza zatvorene zagrade u kojoj se nalazi uvjet petlje (slična strategija vrijedi za tijelo “**if**” naredbe).

Može se postaviti pitanje zbog čega kompjajler ne otkriva i ne prijavljuje ovakve vrste grešaka? Odgovor je u tome što, formalno gledajući, ovo nisu *jezičke (sintaksne)* nego *logičke* greške: petlje “bez tijela” (tj. sa praznim tijelom) *nisu zabranjene* u jeziku C++ i nekada čak imaju i smisla! Kada upoznamo neke naprednije konstrukcije jezika C++, vidjećemo da se neki problemi prirodno rješavaju uz pomoć petlji koje uopće nemaju tijela. Na ovom mjestu ćemo dati samo jednu vještački formirani ilustraciju koja ukazuje na mogućnost smislenih petlji bez tijela. Razmotrimo sljedeći matematski problem: Naći najmanji prirodan broj n takav da je vrijednost izraza $5n + 11$ veća od 30000. Ostavimo po strani činjenicu da je matematsko rješenje ovog problema krajnje jednostavno (svaki učenik osnovne škole će bez imalo muke pronaći da je rješenje 5998), i potražimo rješenje “grubom silom”, tj. ispitivanjem svih brojeva po redu dok ne nađemo broj koji ispunjava zadani uvjet. To bismo mogli ostvariti sljedećom sekvencom naredbi:

```
int n(1);
while(5 * n + 11 <= 30000) n++;
cout << "Traženi broj je " << n << endl;
```

U ovom slučaju tijelo petlje se sastoji samo od naredbe za povećanje promjenljive “n”. Radikalniji C++ programer bi mogao povećanje ove promjenljive realizirati kao bočni efekat unutar samog uvjeta “**while**” naredbe, čime bi u potpunosti eliminirao potrebu za tijelom petlje:

```
int n(0);
while(5 * ++n + 11 <= 30000);
cout << "Traženi broj je " << n << endl;
```

Ovakav stil pisanja se *ne preporučuje*, naročito ne početnicima, ali ako ipak volite egzotike, razmislite zašto smo ovdje inicijalizirali promjenljivu “n” na nulu a ne na jedinicu, i zašto program ne bi radio da

smo umjesto “`++n`” upotrebili “`n++`”. Kasnije ćemo vidjeti smislenije i korisnije primjere petlji bez tijela, a ovdje je samo cilj da uvidimo da takve petlje *u načelu* mogu imati smisla (naravno, da bi petlja bez tijela imala ikakav smisao, uvjet definitivno mora sadržavati neki bočni efekat).

Razmotrimo sada kako bismo mogli ostvariti *bolju kontrolu nad ulaznim tokom*. Već smo ranije rekli da u slučaju da korisnik sa tastature unese neočekivani podatak (npr. tekst kada se očekuje broj), ulazni tok dospijeva u neispravno stanje, i svaka dalja upotreba ulaznog toka biće ignorirana, sve dok tok ne vratimo u ispravno stanje (pozivom funkcije “`cin.clear`”). U narednom primjeru ćemo iskoristiti “`while`” petlju sa ciljem ponavljanja unosa sve dok unos ne bude ispravan:

```
int broj;
cout << "Unesite broj: ";
cin >> broj;
while(!cin) {
    cout << "Nemojte se šaliti, unesite broj!\n";
    cin.clear();
    cin.ignore(10000, '\n');
    cin >> broj;
}
cout << "U redu, unijeli ste broj " << broj << endl;
```

Rad ovog isječka je prilično jasan. Ukoliko je nakon zahtijevanog unosa broja zaista unesen broj, ulazni tok će biti u ispravnom stanju, uvjet “`!cin`” neće biti tačan, i tijelo “`while`” petlje neće se uopće ni izvršiti. Međutim, ukoliko tok dospije u neispravno stanje, unutar tijela petlje ispisujemo poruku upozorenja, vraćamo tok u ispravno stanje, uklanjamo iz ulaznog toka znakove koji su doveli do problema (pozivom funkcije “`cin.ignore`”), i zahtijevamo novi unos. Nakon toga, uvjet petlje se ponovo provjerava, i postupak se ponavlja sve dok unos ne bude ispravan (tj. sve dok uvjet “`!cin`” ne postane netačan).

U prethodnom primjeru, naredba za unos broja sa tastature ponovljena je unutar petlje. Može li se ovo dupliranje izbjegići? Mada na prvi pogled izgleda da je ovo dupliranje nemoguće (unos je potreban *prije* testiranja uvjeta petlje, dakle praktično *izvan* petlje, a potrebno ga ponoviti u slučaju neispravnog unosa, dakle *unutar* petlje), odgovor je ipak potvrđan, uz upotrebu jednog prilično “prljavog” trika, koji vrijedi objasniti, s obzirom da se često susreće u C++ programima. Ideja je da se sam unos sa tastature ostvari kao *bočni efekat uvjeta petlje!* Naime, konstrukcija “`cin >> broj`” sama po sebi predstavlja izraz (sa bočnim efektom), koji pored činjenice da očitava vrijednost promjenljive “`broj`” iz ulaznog toka, također kao rezultat vraća sam objekat ulaznog toka “`cin`”, na koji je dalje moguće primijeniti operator “`!`” sa ciljem testiranja ispravnosti toka. Stoga je savršeno ispravno formirati izraz poput “`!(cin >> broj)`” koji će očitati vrijednost promjenljive “`broj`” iz ulaznog toka a zatim testirati stanje toka. Rezultat ovog izraza biće “`true`” ili “`false`”, u zavisnosti od stanja toka, odnosno on predstavlja sasvim ispravan uvjet, koji se može iskoristiti za kontrolu “`while`” petlje! Kada uzmemo sve ovo u obzir, nije teško shvatiti kako radi sljedeći programski isječak:

```
int broj;
cout << "Unesite broj: ";
while(!(cin >> broj)) {
    cout << "Nemojte se šaliti, unesite broj!\n";
    cin.clear();
    cin.ignore(10000, '\n');
}
cout << "U redu, unijeli ste broj " << broj << endl;
```

Demonstriraćemo još jedan interesantan primjer gdje je korisno imati bočni efekat unutar uvjeta

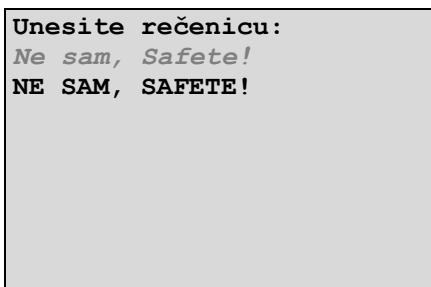
petlje. Pretpostavimo da želimo napisati program koji će tražiti od korisnika unos neke rečenice, a koji će zatim ponoviti istu rečenicu pretvarajući svako malo slovo u veliko. Kako ne znamo unaprijed koliko će rečenica imati znakova, znakove ćemo čitati u petlji, koja će se izvršavati sve dok se ne dostigne oznaka za kraj reda, tj. kontrolni znak '\n'. Znakove ćemo čitati pozivom funkcije "cin.get", s obzirom da operator izdvajanja ">>" ignorira razmake i oznaku kraja reda, kao što smo već ranije napomenuli. Stoga bi traženi program mogao izgledati poput sljedećeg:

```
#include <iostream>
#include <cctype>

using namespace std;

int main() {
    cout << "Unesite rečenicu: ";
    char znak = cin.get();
    while(znak != '\n') {
        cout << (char) toupper(znak);
        znak = cin.get();
    }
    cout << endl;
    return 0;
}
```

Mogući scenario izvršavanja ovog programa prikazan je na sljedećoj slici:



Može se primijetiti da i u ovom primjeru imamo dupliranje: funkciju "cin.get" pozivamo kako unutar tijela petlje, tako i prije ulaska u petlju, iz sličnih razloga kao u prethodnom primjeru. Dupliranje možemo izbjegići na sličan način kao u prethodnom primjeru, tako što očitavanje znaka pozivom funkcije "cin.get" obavimo kao bočni efekat unutar uvjeta petlje. Na taj način dobijamo sljedeći optimizirani program:

```
#include <iostream>
#include <cctype>

using namespace std;

int main() {
    cout << "Unesite rečenicu: ";
    char znak;
    while((znak = cin.get()) != '\n') cout << toupper(znak);
    cout << endl;
    return 0;
}
```

U ovom slučaju, očitani znak se *dodjeljuje* promjenljivoj "znak", nakon čega se testira da li je upravo dodijeljena vrijednost različita od oznake kraja reda. Ovo je interesantan primjer smislene upotrebe

operatora dodjele “=” (a ne operatora poređenja “==”) unutar uvjeta “**while**” petlje (sličan primjer upotrebe mogao bi se konstruisati i za naredbu grananja “**if**”). Trikovi poput ovih ubrajaju se u “prljave trikove”, ali se opisane konstrukcije toliko često koriste u postojećim C++ programima da ih nije loše poznavati. Naravno, ovakve trikove treba koristiti samo u slučajevima kada je korist od njihove upotrebe zaista očigledna, a ne samo sa ciljem “pametovanja”.

Uvjet kod “**while**” petlje ispituje se na *sautom početku petlje*, tako da se tijelo petlje neće izvršiti niti jedanput ukoliko je pripadni uvjet već na početku bio netačan. Međutim, često je potrebno imati takvu strukturu ponavljanja u kojoj neku akciju (ili skupinu akcija) želimo da preduzmemo *barem jedanput*, i da želimo vršiti ponavljanje sve dok je neki uvjet ispunjen, pri čemu se ispunjenje uvjeta treba provjeravati tek na *kraju petlje* (a ne na početku), jer je uvjet takav da ovisi o vrijednosti nekih promjenljivih čije vrijednosti *tek treba da se formiraju* unutar tijela petlje. Ovo se naprimjer dešava kada tražimo neki unos od korisnika, i ponavljamo zahtjev sve dok unos ne bude ispravan. Na primjer, ovdje je dat algoritam koji prikazuje na ekranu izbor opcija (meni), i traži unos od korisnika sve dok korisnik ne da korektan odgovor:

- **Radi sljedeće:**
 - Ispiši spisak opcija (“E - Editor”, “S - Snimanje”, “B - Brisanje”, “K - Kraj”);
 - Učitaj znak sa tastature
- Ponavljam gore navedene akcije *sve dok je znak različit od 'E', 'S', 'B' i 'K'*;

U prikazanom opisu algoritma, prvo se izvršavaju sve naredbe koje slijede iza fraze “**Radi sljedeće**” (engl. “**Do**”), a nakon toga se ispituje uvjet naveden iza fraze “**sve dok je**” (engl. “**while**”). Ako je on tačan (tj. ako je korisnik unijeo pogrešan znak), naredbe unutar petlje (pričuvane uvučeno) se ponavljaju, uvjet se testira ponovo, itd. sve dok je uvjet ispunjen (u našem slučaju, sve dok korisnik ne unese ispravan znak).

U jeziku C++, ovakav tip ponavljanja realizira se pomoću strukture “**do** – ”**while**”, koja se koristi u sljedećem obliku:

```
do naredba while(neki_uvjet) ;
```

Smisao ove strukture potpuno je analogan običnoj ”**while**” naredbi, samo što se uvjet provjerava tek na kraju tijela petlje, odnosno nakon što se naredba napisana između ”**do**” i ”**while**” izvrši. Naravno, ukoliko želimo formirati tijelo petlje koje se sastoji od više naredbi, sve naredbe koje čine tijelo petlje objedinićemo u blok. Tako bi se ranije prikazan algoritam u jeziku C++ mogao realizirati ovako:

```
char opcija;
do {
    cout << "E - Editor\nS - Snimanje\nB - Brisanje\nK - Kraj\n\n"
    "Unesite opciju: ";
    cin >> opcija;
} while(opcija != 'E' && Opcija != 'S' && opcija != 'B' && opcija != 'K');
```

Struktura ”**do** – ”**while**” često se može (ali ne uvijek) zamijeniti običnom ”**while**” naredbom tako što će se neka promjenljiva od koje zavisi uvjet petlje vještački inicijalizirati na neku vrijednost koja će garantirati da će petlja započeti, odnosno da će uvjet na početku biti ispunjen. Na primjer, prethodni algoritam mogao se realizirati i pomoći obične ”**while**” petlje, na sljedeći način, u kojem je promjenljiva ”**opcija**” inicijalizirana na neki neispravni znak (npr. ’Z’) čime se garantira da će se naredbe unutar petlje sigurno izvršiti barem jedanput:

```
char opcija('Z');
while(opcija != 'E' && Opcija != 'S' && opcija != 'B' && opcija != 'K') {
    cout << "E - Editor\nS - Snimanje\nB - Brisanje\nK - Kraj\n\n"
```

```

    "Unesite opciju: ";
    cin >> opcija;
}

```

Međutim, ukoliko je priroda problema takva da se logično nameće ispitivanje uvjeta *na kraju* a ne *na početku* petlje, prirodnije je koristiti "do" – "while" strukturu. U posljednje vrijeme mogu se čuti prigovori da "do" – "while" struktura ima tendenciju da dopušta duži period postojanja promjenljivih bez dodijeljene vrijednosti (tj. neinicijaliziranih promjenljivih) nego obična "while" petlja, i da je zbog toga treba izbjegavati. Mada ova primjedba nije posve bez osnova, ipak djeluje previše paranoično iskoristiti je kao razlog za "protjerivanje" strukture "do" – "while" iz programa.

Česti su slučajevi kada je gotovo svejedno da li ćemo koristiti "do" – "while" strukturu ili običnu "while" petlju. Na primjer, program za računanje sume unesenih brojeva do prvog negativnog unesenog broja koji smo pisali nešto ranije može se lako napisati i pomoću strukture "do" – "while":

```

#include <iostream>
using namespace std;

int main() {
    cout << "Unesite brojeve koji će se sabirati "
        "- negativan broj označava prekid.\n\n";
    int suma(0), broj(0);
    do {
        suma += broj;
        cout << "Unesite broj: ";
        cin >> broj;
    } while(broj >= 0);
    cout << "\nSuma unesenih brojeva je: " << suma << endl;
    return 0;
}

```

S druge strane, postoje brojne situacije u kojima je provjera uvjeta na kraju petlje koju vrši struktura "do" – "while" prirodnija od provjere uvjeta na početku koja se dešava u običnoj "while" petlji. Na primjer, neka je potrebno napraviti program koji određuje broj cifara i sumu cifara unesenog broja. Ideja se sastoji u činjenici da je ostatak dijeljenja bilo kojeg broja sa 10 jednak njegovoj posljednjoj cifri, dok cjelobrojni količnik sa 10 sadrži sve cifre osim posljednje. Dakle, ako uzastopno dijelimo broj sa 10, i pri tome na neku promjenljivu čija je početna vrijednost nula stalno nadodajemo ostatak tog dijeljenja, i ako postupak ponavljamo sve dok količnik ne bude nula, dobicemo željenu sumu cifara. Broj cifara ćemo dobiti tako što ćemo prosto brojati koliko je cifara izdvojeno u opisanom postupku:

```

#include <iostream>
using namespace std;

int main() {
    int broj;
    cout << "Unesite broj: ";
    cin >> broj;
    int suma(0), broj_cifara(0), broj_1(broj);
    do {
        int cifra = broj % 10;      // Izdvajena cifra
        suma += cifra;
        broj_cifara++;
        broj /= 10;
    } while(broj != 0);
    cout << "Broj " << broj_1 << " ima " << broj_cifara << " cifara\n"
}

```

```

    "Suma njegovih cifara je " << suma << endl;
    return 0;
}

```

Obratimo pažnju na pomoćnu promjenljivu “`broj_1`” koju smo inicijalizirali na vrijednost promjenljive “`broj`”. Ona nam je potrebna da bismo na kraju mogli ispisati originalnu vrijednost unesenog broja, s obzirom da je izvorna vrijednost promjenljive “`broj`” uništena tokom izvršavanja petlje (njena konačna vrijednost je nula). Također je interesantno razmotriti promjenljivu “`cifra`” koja je deklarirana *unutar tijela petlje*. Kao takva, ona “živi” samo dok traje izvršavanje tijela petlje, nakon čega ona prestaje postojati. Ovo je dozvoljeno s obzirom da nam ova promjenljiva samo i treba unutar tijela petlje (smatra se veoma dobrom praksom uvijek tako deklarirati promjenljive da im vrijeme života bude ograničeno samo na onaj period kada su zaista potrebne, jer se na taj način efikasnije troše memorijski resursi, i smanjuje se opasnost od grešaka). Strogo uvezši, promjenljiva “`cifra`” nam nije bila ni neophodna, jer smo skupinu naredbi

```

int cifra = broj % 10;      // Izdvojena cifra
summa += cifra;

```

mogli prosto zamijeniti jednom naredbom

```
summa += broj % 10;
```

Također je principijelno moguće umjesto “`while(broj != 0)`” pisati samo “`while(broj)`”, zbog automatske konverzije numeričkih u logičke vrijednosti, o čemu smo detaljno govorili u prethodnom poglavlju. Ipak, ovo se ne preporučuje, zbog smanjenja čitljivosti programa. Krajnje radikalni programer bi mogao izbjegći i naredbu “`broj /= 10;`” tako što bi dijeljenje promjenljive “`broj`” sa 10 izveo kao bočni efekat uvjeta petlje pisanjem konstrukcije poput “`while((broj /= 10) != 0)`” ili, još kraće, samo “`while(broj /= 10)`”. Za ovolikim pretjerivanjem zaista nema nikakve potrebe (da i ne govorimo koliko se ovako narušava jasnoća programa), ali ipak nije loše da razmotrite kako ovo radi. Primjetimo još da kod strukture “`do`” – “`while`” tačka-zarez iza zatvorene zagrade koja slijedi nakon ključne riječi “`while`” nije vjerovatna greška (kao kod obične ”`while`” petlje) već *obavezni element* koji se ne smije izostaviti, s obzirom da se radi o *kraju naredbe*.

Razmotrimo pažljivije zbog čega nam je ovdje bila neophodna “`do`” – “`while`” petlja, a ne obična ”`while`” petlja. Strogo uvezši, da nam je bilo neophodno samo računanje sume cifara, a ne i brojanje cifara, dovoljna bi bila obična ”`while`” petlja. Naime, razmotrimo sljedeći program, koji računa sumu cifara unesenog broja, a u kojem se koristi obična ”`while`” petlja:

```

#include <iostream>
using namespace std;

int main() {
    int broj;
    cout << "Unesite broj: ";
    cin >> broj;
    int suma(0), broj_1(broj);
    while(broj != 0) {
        suma += broj % 10;
        broj /= 10;
    }
    cout << "Suma cifara broja " << broj_1 << " je " << suma << endl;
    return 0;
}

```

Ako pažljivo analiziramo ovaj program vidjećemo da će se tijelo petlje u svakom slučaju izvršiti makar jedanput (preciznije, onoliko puta koliko broj ima cifara), *osim u slučaju kada je unesen broj nula*. U tom slučaju tijelo petlje neće se izvršiti nijednom. Međutim, program će i u tom slučaju davati ispravan rezultat, jer je promjenljiva "suma" inicijalizirana na nulu, i takva će i ostati, a suma cifara broja 0 je također nula. S druge strane, ukoliko bismo u isti program dodali brojač koji utvrđuje broj cifara brojanjem prolazaka kroz petlju, dobili bismo besmislen rezultat da broj nula ima nula cifara! Nama je zapravo potrebno da se petlja u svakom slučaju izvrši barem jedanput, što smo upravo postigli "do" – "while" petljom (stoga se kaže da je "do" – "while" petlja *bezvjetna*, jer se njeno tijelo bezvjetno izvršava barem jedanput, dok je obična "while" petlja *uvjetna*, jer se njeno tijelo ne mora izvršiti). U ovom slučaju, ne možemo vještačkom inicijalizacijom neke promjenljive izazvati siguran ulazak u petlju, jer u ovom slučaju ulazak odnosno neulazak u petlju ovisi od vrijednosti koju korisnik unese sa tastature. Naravno, kako jedini problem nastaje pri unosu nule, možemo ovaj specijalni slučaj tretirati posebno (pomoću "if" naredbe), kao u sljedećem programu:

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

int main() {
    int broj;
    cout << "Unesite broj: ";
    cin >> broj;
    int suma(0), broj_cifara(0), broj_1(broj);
    if(broj == 0) broj_cifara = 1;
    while(broj != 0) {
        suma += broj % 10;
        broj_cifara++;
        broj /= 10;
    }
    cout << "Broj " << broj_1 << " ima " << broj_cifara << " cifara\n"
        "Suma njegovih cifara je " << suma << endl;
    return 0;
}
```

Naravno, rješenje sa "do" – "while" petljom je elegantnije s obzirom da ne moramo voditi računa ni o kakvima specijalnim slučajevima. Inače se od više različitih rješenja nekog problema obično može smatrati elegantnijim ono rješenje kod kojeg se javlja manja potreba za tretiranjem raznih specijalnih slučajeva.

Naredni primjer također demonstrira slučaj u kojem je "do" – "while" petlja pogodnija od obične "while" petlje. Neka je potrebno izračunati najveći zajednički djelilac (NZD) dva cijela broja a i b . Najefikasniji metod za rješavanje ovog problema je poznati Euklidov algoritam koji se sastoji u sljedećem: prvi broj se dijeli sa drugim, a zatim se drugi broj dijeli sa ostatkom tog dijeljenja, i postupak se ponavlja sve dok se ne dobije ostatak jednak nuli. Posljednji djelilac je tada traženi NZD. Na primjer, postupak traženja NZD za brojeve 2725 i 115 izgleda ovako:

$$\begin{array}{r} 2725 : 115 = 23 \\ \quad 425 \\ \quad \quad 80 \\ \quad \quad \quad 35 \\ \quad \quad \quad \quad 10 \\ \quad \quad \quad \quad \quad 5 \\ \quad \quad \quad \quad \quad \quad 0 \end{array} \qquad \begin{array}{r} 115 : 80 = 1 \\ \quad 35 \\ \quad \quad 10 \\ \quad \quad \quad 5 \\ \quad \quad \quad \quad 0 \end{array} \qquad \begin{array}{r} 80 : 35 = 2 \\ \quad 10 \\ \quad \quad 5 \\ \quad \quad \quad 0 \end{array} \qquad \begin{array}{r} 35 : 10 = 3 \\ \quad 5 \\ \quad \quad 0 \end{array} \qquad \begin{array}{r} 10 : 5 = 2 \\ \quad 0 \end{array}$$

Dakle, $\text{NZD}(2725, 115) = 5$. Slijedi program koji po ovom algoritmu računa NZD (samo glavni dio):

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

int main() {
```

```

int a, b, ostatak;
cout << "Unesite dva broja: ";
cin >> a >> b;
do {
    ostatak = a % b; a = b; b = ostatak;
} while(ostatak != 0);
cout << "NZD(" << a << "," << b << ") = " << a << endl;
return 0;
}

```

U prikazanom programu upotrijebljena je “**do**” – “**while**” petlja, s obzirom da njen uvjet zavisi od vrijednosti promjenljive “**ostatak**”, koja se prvi put računa tek unutar tijela petlje.

Razmotreni program ilustrira veoma čestu tehniku programiranja koju možemo nazvati *presipanje promjenljivih iz jedne u drugu unutar petlje*. Naime, možemo primijetiti da se sadržaji pojedinih promjenljivih unutar petlje stalno “presipaju” iz jedne u drugu, da bi u svakom novom prolasku kroz petlju imale drugačije vrijednosti nego što su imale u prethodnom prolasku kroz petlju (preciznije, u svakom trenutku promjenljive “a” i “b” sadrže *tekući djeljenik* i *tekući djelilac*, a ne nužno vrijednosti kakve su bile unesene na početku). Primijetimo da iako iz algoritma slijedi da bi rezultat na kraju trebao da bude u promjenljivoj “b”, on se na kraju nalazi u promjenljivoj “a”, jer je presipanje (dodjeljivanje)

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

izvršeno bez obzira da li je vrijednost promjenljive “**ostatak**” bila nula ili nije. Također primijetimo da je redoslijed u kojem se izvršava dodjeljivanje *veoma bitan*: da smo u istom programu napisali

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

program ne bi davao tačan rezultat (razmislite zašto).

Iako su Vam ovakve napomene možda već postale dosadne, još jedanput ćemo upozoriti da *ne zloupotrebljavate* operatore jezika C++ (što je omiljena zabava “hakerski” nastrojenih početnika). Na primjer, neko bi mogao petlju u prethodnom programu napisati na sljedeći (principijelno ispravan) način u kojem je zloupotrebljen operator dodjele “=” unutar “**while**” naredbe:

```

do {
    ostatak = a % b; a = b;
} while (b = ostatak);

```

Jasno je da je prikazano rješenje veoma ružno. Naime, letimičnim pogledom na program svako bi vjerovatno pomislio da se u njemu *porede* vrijednosti promjenljivih “b” i “**ostatak**”, a zapravo se promjenljiva “**ostatak**” dodjeljuje promjenljivoj “b” i rezultat dodjele poredi sa nulom.

U sljedećem primjeru kombinira se “**do**” – “**while**” struktura sa “**if**” – “**else**” strukturom. Priloženi program omogućava korisniku da obavlja niz operacija na bankovnom računu. Korisnik treba da ukuca jedno slovo, na primjer slovo 'U' (unos) za stavljanje novca na račun, slovo 'P' (podizanje) za uzimanje novca sa računa, ili slovo 'K' (kraj) za prekid rada. Ako se traži stavljanje ili uzimanje novca sa računa, program učitava željeni iznos, i zatim prikazuje novo stanje na računu. Nakon izvršene jedne operacije unosa ili uzimanja novca, korisniku je omogućen izbor nove operacije, s tim da se program završava kada korisnik ukuca slovo 'K'. Početno stanje na računu je nula. Programu je svejedno da li korisnik unosi velika ili mala slova. U slučaju da korisnik unese nepostojeću komandu, ispisuje se upozorenje, nakon čega korisnik može zadati novu komandu.

```

#include <iostream>
using namespace std;

int main() {
    double stanje(0);
    char komanda;
    do {
        cout << "Unesite komandu (U - unos, P - podizanje, K - kraj): ";
        cin >> komanda;
        if(komanda == 'U' || komanda == 'u') {
            double iznos;
            cout << "Unesite iznos koji ulažete: ";
            cin >> iznos;
            stanje += iznos;
            cout << "Novo stanje na računu je: " << stanje << endl;
        }
        else if(komanda == 'P' || komanda == 'p') {
            double iznos;
            cout << "Unesite iznos koji podižete: ";
            cin >> iznos;
            if(stanje < iznos)
                cout << "Nažalost, nemate dovoljno novca na računu!\n";
            else {
                stanje -= iznos;
                cout << "Novo stanje na računu je: " << stanje << endl;
            }
        }
        else if(komanda != 'K' && komanda != 'k')
            cout << "Neispravna komanda!\n";
    } while(komanda != 'K' && komanda != 'k');
    return 0;
}

```

U ovom primjeru interesantno je razmotriti vrijeme života pojedinih promjenljivih korištenih u programu. Prvo obratimo pažnju na promjenljivu "iznos". Izgleda kao da je ova promjenljiva deklarirana *dva puta* u programu. Međutim, u suštini se radi o *dvije različite promjenljive istog imena* od kojih svaka postoji samo unutar bloka u kojem je deklarirana. U principu smo mogli deklarirati i samo jednu promjenljivu "iznos" koja bi vrijedila u oba bloka (na primjer, deklaracijom na početku tijela "do" – "while" petlje), ali ovdje smo namjerno željeli demonstrirati i ovakvu mogućnost. Dalje, promjenljiva "komanda" deklarirana je *ispred* "do" – "while" petlje. Ovu promjenljivu *nismo smjeli deklarirati unutar tijela petlje* (mada se tek tamo prvi put koristi), jer u tom slučaju ona *ne bi bila* dostupna *unutar uvjeta petlje* (s obzirom da bi joj dostupnost bila ograničena samo na tijelo petlje). Ipak, najinteresantnije je razmotriti promjenljivu "stanje". Na prvi pogled, ona bi se mogla deklarirati unutar tijela "do" – "while" petlje, jer se ona samo tamo i koristi. Međutim, ukoliko to učinimo, program neće raditi ispravno! Naime, ova promjenljiva je inicijalizirana na vrijednost nula. Ukoliko neku promjenljivu *deklariramo i inicijaliziramo* unutar tijela petlje, ta inicijalizacija će se *iznova vršiti* pri svakom prolasku kroz petlju (odnosno, svaki put kada se pri izvršavanju programa naiđe na navedenu inicijalizaciju). U našem slučaju, vrijednost promjenljive "stanje" bi se pri svakom prolasku kroz petlju iznova vraćala na nulu, umjesto da čuva tekuću vrijednost stanja na računu. Iz izloženog primjera možemo zaključiti da je potreban izvjestan oprez pri deklariranju promjenljivih unutar tijela petlje. Naime, unutar tijela petlje ima smisla deklarirati samo promjenljive čisto lokalnog značaja (npr. privremene promjenljive, čiji značaj prestaje nakon što se upotrijebi, poput promjenljive "iznos" iz prikazanog primjera).

Najčešći algoritmi koji koriste ponavljanje su algoritmi u kojima se javlja potreba za ponavljanjem neke akcije ili skupine akcija *tačno određeni broj puta*. Na primjer, zamislimo da želimo da unesemo ukupnu količinu padavina u toku svakog mjeseca u godini, i da izračunamo prosječnu količinu padavina u toku jednog mjeseca. Algoritam za rješavanje ovog problema može izgledati ovako:

- Postavi ukupnu količinu padavina na 0;
- Za sve mjesecce od 1 do ukupnog broja mjeseci (12):
 - Unesi količinu padavina u toku mjeseca;
 - Dodaj uneseni broj na ukupnu količinu;
- Podijeli ukupnu količinu sa brojem mjeseci (12);
- Prikaži rezultat;

Specijalna riječ “**Za**” (engl. “**for**”) upotrebljena u opisu algoritma ukazuje da se akcije koje slijede (pričekane uvučeno) trebaju ponoviti tačno određeni broj puta, preciznije za sve mjesecce redom od 1 do 12. Pričekani algoritam nije teško realizirati pomoću naredbe “**while**”:

```
#include <iostream>
using namespace std;
int main() {
    const int BrojMjeseci(12); // Ukupan broj mjeseci
    double ukupno(0); // Ukupna godišnja kolicina padavina
    int mjesec; // Redni broj mjeseca (1 - 12)

    mjesec = 1;
    while(mjesec <= BrojMjeseci) {
        double padavine;
        cout << "Unesite količinu padavina u toku "
            << mjesec << ". mjeseca: ";
        cin >> padavine;
        ukupno += padavine;
        mjesec++;
    }
    double prosjek = ukupno / BrojMjeseci;
    cout << "Prosječna količina padavina je: " << prosjek << endl;
    return 0;
}
```

Razumije se da je dodjela “mjesec = 1” mogla biti izvršena odmah pri deklaraciji promjenljive “mjesec”, putem njene inicijalizacije. Međutim, ovdje smo namjerno razdvojili deklaraciju promjenljive i dodjelu vrijednosti da bismo jasnije uočili neke karakteristike koje su zajedničke za sve algoritme ovog tipa.

Mada je ova realizacija sasvim korektna, i ne može joj se ništa osporiti sa aspekta efikasnosti, u jeziku C++ algoritmi poput ovog elegantnije se izvode pomoću naredbe “**for**”. Naime, u svim algoritmima ovog tipa može se uočiti pojava karakteristične konstrukcije poput sljedeće:

```
inicijalizacija_brojača;
while(neki_uvjet) {
    skupina_naredbi;
    izmjena_brojača;
}
```

U našem primjeru, frazu “*inicijalizacija_brojača*” predstavlja je izraz dodjeljivanja “`mjesec = 1`”, frazu “*neki_uvjet*” predstavlja je uvjet “`mjesec <= 12`”, dok je frazi “*izmjena_brojača*” odgovarao izraz “`mjesec++`”. Ovakva konstrukcija dovoljno se često javlja da je jezik C++ uveo posebnu naredbu “**for**”. Ona se koristi u sljedećem obliku:

```
for (inicijalizacija_brojača; neki_uvjet; izmjena_brojača) naredba;
```

U suštini, značenje prikazane naredbe “**for**” identično je značenje maločas navedene konstrukcije, samo što se u ovom slučaju tijelo petlje sastoji samo od jedne naredbe (označene frazom “*naredba*”). Međutim, i u ovom slučaju moguće je upotrijebiti čitavu skupinu naredbi, ukoliko ih prethodno objedinimo u blok. Tako, korištenjem naredbe “**for**”, program za računanje prosječne količine padavina možemo napisati ovako:

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

int main() {
    const int BrojMjeseci(12); // Ukupan broj mjeseci
    double ukupno(0); // Ukupna godišnja kolicina padavina
    int mjesec; // Redni broj mjeseca (1 - 12)

    for(mjesec = 1; mjesec <= BrojMjeseci; mjesec++) {
        double padavine;
        cout << "Unesite količinu padavina u toku "
            << mjesec << ". mjeseca: ";
        cin >> padavine;
        ukupno += padavine;
    }
    double prosjek = ukupno / BrojMjeseci;
    cout << "Prosječna količina padavina je: " << prosjek << endl;
    return 0;
}
```

Naredba “**for**” u jeziku C++ je veoma općenita, mnogo opštija nego u drugim programskim jezicima. Ipak, ova naredba se najčešće koristi kada se neki algoritam treba izvršiti za više vrijednosti neke promjenljive (brojača) koja redom uzima vrijednosti od neke početne do neke krajnje vrijednosti. U takvim specijalnim slučajevima naredba “**for**” dobija sljedeći karakterističan oblik:

```
for (brojač = početna_vrijednost; brojač <= krajnja_vrijednost; brojač++) naredba;
```

Tako će sljedeća sekvenca naredbi ispisati sve brojeve redom od 1 do 10 razmagnute jednim razmakom:

```
int i;
for(i = 1; i <= 10; i++) cout << i << " ";
```

Općenitost naredbe “**for**” u jeziku C++ ogleda se u činjenici da njeni argumenti (koji se međusobno razdvajaju *tačka-zarezom*, za razliku od argumenata funkcija koji se razdvajaju *zarezom*) mogu biti *potpuno proizvoljni izrazi*. Generalno je naredba oblika

```
for (izraz_1; izraz_2; izraz_3) naredba;
```

potpuno ekvivalentna konstrukciji

```

izraz_1;
while(izraz_2) {
    naredba;
    izraz_3;
}

```

odakle, između ostalog, slijedi da izrazi “*izraz_1*” i “*izraz_3*” moraju sadržavati neki bočni efekat da bi čitava konstrukcija imala smisla. Riječima iskazano, smisao naredbe “**for**” je slijedeći:

- *Inicijaliziraj* promjenljive pomoću izraza “*izraz_1*”;
- Izvršavaj tijelo petlje sve dok je izraz “*izraz_2*” *tačan* (ili *različit od nule* ukoliko se ne radi o logičkom izrazu);
- Svaki put kada se tijelo petlje izvrši *izmjeni vrijednost* promjenljivih (drugim riječima, izvrši njihovu *reinicijalizaciju*) pomoću izraza “*izraz_3*”.

Ova fleksibilnost naredbe “**for**” često je jako korisna (naravno, ne treba je zloupotrebjavati). Tako, na primjer, sljedeća sekvenca ispisuje stepene broja 2 koji nisu veći od 70000 (ovdje koristimo tip “**long int**” da bismo izbjegli eventualne probleme sa prekoračenjem na kompjuterima kod kojih promjenljive tipa “**int**” zauzimaju 2 bajta):

```

long int i;
for(i = 1; i <= 70000; i *= 2) cout << i << " ";

```

Bilo koji od tri argumenta “**for**” naredbe može da se i izostavi (mada najčešće nema osobitog razloga da to radimo). Pri tome se tačka-zarezi *ne izostavljaju* (razmislite zašto). Tako je, na primjer,

```
for(; izraz_2;)
```

isto što i

```
while(izraz_2)
```

Ukoliko se izostavi središnji izraz koji predstavlja *uvjet petlje* (izraz “*izraz_2*”), smatra se da je uvjet *stalno ispunjen*, tako da će se petlja izvršavati unedogled, osim ukoliko se nasilno prekine (jedan od načina za nasilno prekidanje petlje upoznaćemo uskoro).

Ovdje nije na odmet ukazati na jednu čestu grešku, na koju smo doduše ukazivali i ranije, ali koja se naročito često javlja pri upotrebi naredbe “**for**”. Razmislite šta će ispisati sljedeća sekvenca naredbi:

```

int i;
for(i = 1; i <= 5; i++);
    cout << i;

```

Ukoliko je Vaš odgovor “12345”, pogriješili ste. Tačan odgovor je “6”. Naime, zbog tačka-zareza napisanog neposredno iza zatvorene zagrade, tijelo ove petlje je *prazno*, a naredba za ispis promjenljive “*i*”, koja se vjerovatno trebala nalaziti unutar tijela petlje, zapravo se nalazi izvan tijela petlje, i izvršava se tek kada se petlja završi (odnosno kada promjenljiva “*i*” dostigne vrijednost 6). Ova sekvenca naredbi zapravo je ekvivalentna sljedećoj sekvenci:

```

int i = 1;
while(i <= 5) i++;
    cout << i;

```

Naravno (i nažalost), prevodilac ne prijavljuje grešku u ovakvim slučajevima, jer su petlje bez tijela u jeziku C++ dozvoljene i često sasvim logične. Na primjer, sljedeći niz naredbi (u kojem se javlja *petlja bez tijela*) pronalazi najmanji prirodni broj n za koji je $n^3 + n + 1 > 30000$:

```
int n;
for(n = 1; n * n * n + n + 1 <= 30000; n++);
cout << "Traženi broj je " << n << endl;
```

Mada deklaracije nisu izrazi, sintaksa jezika C++ (za razliku od jezika C) dopušta da se kao prvi argument naredbe “**for**” upotrijebi i proizvoljna *deklaracija* (koja tipično sadrži i inicijalizaciju). Tako, ispis brojeva od 1 do 10 možemo realizirati i sljedećom konstrukcijom:

```
for(int i = 1; i <= 10; i++) cout << i << " ";
```

Međutim, po konvenciji, promjenljive deklarirane unutar prvog parametra naredbe “**for**” imaju strogo lokalni karakter, odnosno imaju vrijeme života samo do završetka petlje. Preciznije, naredba poput

```
for(deklaracija; izraz_2; izraz_3) naredba;
```

zapravo je ekvivalentna konstrukcija

```
{
    deklaracija;
    while(izraz_2) {
        naredba;
        izraz_3;
    }
}
```

Dodatni par vitičastih zagrada ograničava vrijeme života deklariranih promjenljivih. Danas se preporučuje da se promjenljive koje imaju lokalni značaj samo kao upravljačke promjenljive “**for**” petlje, odnosno koje nisu potrebne nakon završetka petlje, uvijek deklariraju unutar prvog parametra naredbe “**for**”. Na taj način se smanjuje mogućnost nekih grešaka. Na primjer, ukoliko napišemo

```
for(int i = 1; i <= 5; i++);
    cout << i;
```

sa nehotice napisanim tačka-zarezom iza zatvorene zgrade, kompjajler će prijaviti grešku pri pokušaju ispisa promjenljive “*i*” koja prestaje da živi nakon završetka petlje, čije tijelo je zapravo prazno (osim u slučaju da smo ranije imali deklariranu promjenljivu koja se također zove “*i*”, koja je privremeno skrivena istoimenom lokalnom promjenljivom za vrijeme trajanja petlje, a koja postaje ponovo vidljiva nakon što lokalna promjenljiva “*i*” prestane da živi). U skladu sa ovakvom preporukom, program za računanje prosječne količine padavina trebalo bi pisati ovako:

```
#include <iostream>
using namespace std;
int main() {
    const int BrojMjeseci(12);
    double ukupno(0);
    for(int mjesec = 1; mjesec <= BrojMjeseci; mjesec++) {
```

```

double padavine;
cout << "Unesite količinu padavina u toku "
    << mjesec << ". mjeseca: ";
cin >> padavine;
ukupno += padavine;
}
double prosjek = ukupno / BrojMjeseci;
cout << "Prosječna količina padavina je: " << prosjek << endl;
return 0;
}

```

Petlje tipa “**for**” u jeziku C++ zaista su veoma fleksibilne, ali njihovu fleksibilnost ne treba zloupotrebljavati. Čak ni najradikalniji C++ programer vjerovatno neće napisati ovaku (sasvim legalnu) naredbu za ispis brojeva redom od 1 do 10:

```
for(int i = 1; i <= 10; cout << i++ << " ");
```

Kako je naredba “**for**” specijalni slučaj “**while**” naredbe, ako uvjet “**for**” petlje odmah na početku nije ispunjen, petlja se neće izvršiti ni jedanput. Ispostavlja se da je to veoma često upravo ono što nam je potrebno. Međutim, treba pripaziti na ovakve greške. Ako treba ispisati brojeve od 10 do 1 *unazad*, ova naredba *neće raditi ispravno* (preciznije, ona će ispisivati sve brojeve od 10 do najvećeg broja koji može stati u promjenljivu tipa “**int**”, nakon čega će, zbog prekoračenja, promjenljiva “*i*” postati negativna, i petlja će se prekinuti):

```
for(int i = 10; i >= 1; i++) cout << i << " ";
```

Ispravno rješenje je, naravno,

```
for(int i = 10; i >= 1; i--) cout << i << " ";
```

ili u duhu malo radikalnijeg C++ programera (razmislite kako ovo radi):

```
for(int i = 10; i; i--) cout << i << " ";
```

Česta je situacija u kojoj *potpuno istu akciju ili grupu akcija* (bez ikakve izmjene) treba ponoviti više puta. Naredba “**for**” je jako pogodna za tu svrhu. Za brojanje koliko puta je akcija izvršena obavezno treba uvesti neku brojačku promjenljivu, bez obzira što ona uopće neće biti upotrijebljena unutar akcije (ili grupe akcija) koja čini tijelo petlje. Na primjer, sljedeća naredba ispisuje riječi pjesme popularne među engleskim nogometnim navijačima: ona se sastoji od samo jednog stiha “Here we go” koji se ponavlja mnogo puta (u ovom primjeru 20 puta):

```
for(int brojac = 1; brojac <= 20; brojac++) cout << "Here we go\n";
```

Lako je zaključiti da je naredba “**for**” najpogodnija u slučajevima kada se tačno zna unaprijed *za koje vrijednosti neke promjenljive* treba da se izvrši tražena akcija ili skupina akcija. Na primjer, u programu koji računa sumu recipročnih vrijednosti brojeva od 1 do *n* koji smo priložili kao demonstraciju naredbe “**while**”, daleko je prirodnije upotrebiti naredbu “**for**”:

```
#include <iostream>
using namespace std;
int main() {
    int n;
    cout << "Unesite n: "
```

```

    cin >> n;
    double suma(0);
    for(int i = 1; i <= n; i++) suma += 1. / i;
    cout << "Suma recipročnih vrijednosti brojeva od 1 do " << n
        << " iznosi " << suma << endl;
    return 0;
}

```

Slijede još neki primjeri koji ilustriraju primjenu naredbe “**for**”. Naredni program na izlazu daje tabelu konverzije temperature izražene u stepenima Celzijusa u stepene Farenhajta, i to u intervalu od 0 do 100 stepeni Celzijusovih, sa korakom od po 10 stepeni (obratite pažnju kako je ovaj korak realiziran u naredbi “**for**”).

```

#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    cout << "Celzijusi: Farenhajti:\n"
        "-----\n";
    for(double celzijusi = 0; celzijusi <= 100; celzijusi += 10) {
        double farenhajti = 9 * celzijusi / 5 + 32;
        cout << setw(10) << celzijusi << setw(13) << farenhajti << endl;
    }
    return 0;
}

```

U ovom programu smo iskoristili formulu za konverziju stepena Celzijusa u stepene Farenhajta koja glasi:

$$Farenhajt = \frac{9}{5} * Celzijus + 32$$

Usput, razmislite zašto program ne bi radio da smo umjesto naredbe

```
double farenhajti = 9 * celzijusi / 5 + 32;
```

napisali naredbu

```
double farenhajti = 9 / 5 * celzijusi + 32;
```

(uputa: čuvajte se cjelobrojnog dijeljenja!)

Naredni program za uneseni broj n računa njegov faktorijel $n! = 1 \cdot 2 \cdot \dots \cdot n$. Promjenljive su definirane kao promjenljive tipa “**long int**”, sa ciljem da se poveća opseg vrijednosti za koje se dobija tačan rezultat (vodite računa da je faktorijel funkcija koja raste veoma brzo):

```

#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Unesite broj: ";
    cin >> n;
    long int proizvod(1);
    for(int i = 1; i <= n; i++) proizvod *= i;
    cout << n << "!" << " = " << proizvod << endl;
}

```

```

    return 0;
}

```

Vjerovatno ste već primijetili da se izrazi tipa

$$S = \sum_{i=1}^n f(i)$$

najefikasnije izračunavaju pomoću konstrukcije tipa

```

S = 0;
for(int i = 1; i <= n; i++) S += f(i);

```

dok je za računanje izraza tipa

$$P = \prod_{i=1}^n f(i)$$

najefikasnija sljedeća konstrukcija:

```

P = 1;
for(int i = 1; i <= n; i++) P *= f(i);

```

Na ovom mjestu ćemo se ukratko upoznati sa operatorom *zarez*, koji je uglavnom uveden radi primjena u “**for**” naredbi. Izraz oblika

izraz_1, izraz_2, ... izraz_n

interpretira se tako što se svi izrazi “*izraz_1*”, “*izraz_2*” itd. sve do izraza “*izraz_n*” izračunaju (redom slijeva nadesno), ali se kao rezultat čitavog izraza uzima samo rezultat izraza “*izraz_n*”. Čemu uopće služi ovakav izraz? Suština je u tome da se na taj način skupina neovisnih izraza može objediniti tako da se tretira kao *jedan izraz*, i da se prema tome može upotrijebiti na mjestima gdje sintaksa jezika C++ očekuje tačno jedan izraz. U tom smislu, operator zarez objedinjuje više izraza u jednu cjelinu slično kao što blok objedinjuje više naredbi u jednu cjelinu. Naravno, da bi čitava konstrukcija imala smisla, svi izrazi razdvojeni zarezom, osim eventualno posljednjeg, trebaju sadržavati bočne efekte (s obzirom da se njihovi rezultati ignoriraju).

Operator zarez može se ponekad koristiti kao zamjena za blok. Tako se, na primjer, umjesto

```

if(d >= 0) {
    x1 = (-b - sqrt(d)) / (2 * a);
    x2 = (-b + sqrt(d)) / (2 * a);
}

```

može pisati samo

```
if(d >= 0) x1 = (-b - sqrt(d)) / (2 * a), x2 = (-b + sqrt(d)) / (2 * a);
```

Međutim, za razliku od bloka, operator zarez objedinjuje izraze u novi izraz (dok blok objedinjuje naredbe u novu naredbu), tako da se čitava konstrukcija može iskoristiti na mjestu gdje sintaksa jezika C++ zahtijeva isključivo *izraze* (npr. u argumentima naredbe “**for**”). Tako je sljedeća naredba, u kojoj se koristi operator zarez, sasvim korektna i smislena:

```

for(int i = 1, j = 10; i <= j; i++, j--)
    cout << i << " * " << j << " = " << i * j << "     ";

```

Ova naredba će ispisati sljedeće:

```

1 * 10 = 10   2 * 9 = 18   3 * 8 = 24   4 * 7 = 32   5 * 6 = 30

```

Da smo htjeli isti efekat postići bez upotrebe zarez operatora, morali bismo pisati nešto poput sljedeće konstrukcije:

```

{
    int j = 10;
    for(int i = 1; i <= j; i++) {
        cout << i << " * " << j << " = " << i * j << "     ";
        j--;
    }
}

```

Treba voditi računa da operator zarez ne može uvijek zamijeniti blok, jer on može objediniti *samo izraze*, dok blok objedinjuje naredbe, koje su općenitije od izraza (na primjer, deklaracije nisu izrazi). Operator zarez treba koristiti sa oprezom. U neku ruku, od ovog operatara možda ima više štete nego koristi, jer smo već ranije vidjeli da su zahvaljujući njihovom postojanju sintaksno ispravne neke konstrukcije koje ne rade ono što bi se na prvi pogled moglo pomisliti.

Sljedeći program je proširenje programa za pomoć korisniku za izbor pogodnijeg prodavača itisona za svoje kancelarije, koji smo ranije priložili kao ilustraciju naredbe “**if**”. U ovom programu se ne unose dimenzije samo jedne kancelarije, nego dimenzije za nekoliko kancelarija, tako da program može da preporuči jeftiniju varijantu za ukupnu površinu kancelarija. Program na početku pita korisnika u koliko kancelarija treba postaviti itison.

```

#include <iostream>
using namespace std;

int main() {
    const double JedinicnaCijena1(24.50);
    const double JedinicnaCijena2(12.50);
    const double FiksnaCijena_2(400);
    int broj_kancelarija;
    cout << "Koliko ima kancelarija? ";
    cin >> broj_kancelarija;
    double cijena_1(0), cijena_2(0);
    for(int i = 1; i <= broj_kancelarija; i++) {
        double duzina, sirina;
        cout << "Unesite širinu " << i << ". kancelarije u metrima: ";
        cin >> sirina;
        cout << "Unesite dužinu " << i << ". kancelarije u metrima: ";
        cin >> duzina;
        double povrsina = duzina * sirina;
        cijena_1 += JedinicnaCijena1 * povrsina;
        cijena_2 += JedinicnaCijena2 * povrsina + FiksnaCijena2;
    }
    cout << "Prodavač 1 će Vam naplatiti " << cijena_1 << " KM.\n"
        "Prodavač 2 će Vam naplatiti " << cijena_2 << " KM.\n"
        "Preporucujem Vam ";
    if(cijena_1 < cijena_2) cout << "prvog";
    else cout << "drugog";
}

```

```

    cout << " proizvođača, jer je jeftiniji.\n";
    return 0;
}

```

Prilikom demonstriranja određivanja najvećeg zajedničkog djelioca za dva broja pomoću Euklidovog algoritma, korištena je programerska tehnika koju smo nazvali *presipanje promjenljivih iz jedne u drugu unutar petlje*. Kako je ovo veoma često korištena tehnika, kojom se jako efikasno rješavaju izvjesni tipovi problema, ali čija primjena nije uvijek tako očigledna, ovdje ćemo istu tehniku demonstrirati na još jednom primjeru. Neka je potrebno napraviti program koji ispisuje niz Fibonačijevih brojeva, koji su definirani na sljedeći način: prva dva Fibonačijeva broja F_0 i F_1 su jedinice, a svaki sljedeći Fibonačijev broj je zbir dva prethodna Fibonačijeva broja (tj. $F_0=F_1=1$, $F_k=F_{k-1}+F_{k-2}$ za $k>1$). Prvih nekoliko Fibonačijevih brojeva su, prema tome, 1, 1, 2, 3, 5, 8, 13, 21 itd. Program koji slijedi ispisuje prvih n Fibonačijevih brojeva, pri čemu se n unosi sa tastature. Iako je kratak, program je *na prvi pogled nešto teži za razumijevanje*, ali je neophodno uložiti trud za njegovo razumijevanje, jer se na sličan način mogu rješavati mnogi srodnici problemi (ovdje promjenljive "p", "q" i "r" u svakom trenutku čuvaju respektivno *tekuci i dva sljedeća* Fibonačijeva broja):

```

#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Koliko želite Fibonačijevih brojeva? ";
    cin >> n;
    long int p(1), q(1); // Pomoćne promjenljive
    for(int i = 1; i <= n; i++) {
        long int r = p + q; // Još jedna pomoćna promjenljiva
        cout << p << " ";
        p = q; q = r;
    }
    return 0;
}

```

Sljedeći primjer je također veoma važan, jer ilustrira još neke važne tehnike programiranja. Neka je potrebno odrediti najmanji i najveći broj od skupine brojeva koji se unose sa tastature. Osnovna ideja programa je da na početku *prvi uneseni broj* proglašimo za *trenutni minimum* i za *trenutni maksimum*. Pri unosu svakog novog broja, za slučaj da je unesen broj manji od trenutnog minimuma, trenutni minimum postaje jednak unesenom broju. Slično postupamo i sa trenutnim maksimumom. Očigledno će nakon unosa svih brojeva trenutni minimum i trenutni maksimum biti jednak najmanjem i najvećem od unesenih brojeva (odnosno totalnom minimumu i maksimumu):

```

#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Koliko ima brojeva? ";
    cin >> n;
    double min, max;
    for(int i = 1; i <= n; i++) {
        double broj;

```

```

cout << "Unesite " << i << ". broj: ";
cin >> broj;
if(i == 1) { // Postupak za prvi uneseni broj
    min = broj; max = broj;
}
else { // Postupak za ostale brojeve
    if(broj > max) max = broj;
    if(broj < min) min = broj;
}
cout << "Najmanji uneseni broj je " << min
    << ", a najveći uneseni broj je " << max << endl;
return 0;
}

```

Ovaj program se može znatno pojednostaviti ako znamo opseg u kojem će se nalaziti uneseni brojevi. Tako, ako npr. znamo da će svi uneseni brojevi ležati u opsegu od -10000 do $+10000$, možemo na početku inicijalizirati trenutni minimum na $+10000$ a trenutni maksimum na -10000 i na isti način tretirati prvi uneseni broj i sve ostale brojeve. Naime, prvi uneseni broj će svakako biti *veći od tako inicijaliziranog trenutnog maksimuma i manji od tako inicijaliziranog trenutnog maksimuma*, tako da će i trenutni minimum i trenutni maksimum dobiti vrijednost prvog unesenog broja. Za ostale brojeve postupak je identičan kao u prethodnom programu:

```

#include <iostream>
using namespace std;
int main() {
    int n;
    cout << "Koliko ima brojeva? ";
    cin >> n;
    double min(10000), max(-10000);
    for(int i = 1; i <= n; i++) {
        double broj;
        cout << "Unesite " << i << ". broj: ";
        cin >> broj;
        if(broj > max) max = broj;
        if(broj < min) min = broj;
    }
    cout << "Najmanji uneseni broj je " << min
        << ", a najveći uneseni broj je " << max << endl;
    return 0;
}

```

Nema nikakvih razloga da se unutar tijela jedne petlje ne mogu nalaziti druge petlje. To je ilustrirano u sljedećem programu, u kojem se unutar jedne “**do**” – “**while**” petlje (vanske) nalaze druge dvije “**do**” – “**while**” petlje (unutrašnje), kao i jedna “**for**” petlja. U priloženom programu korisnik treba da unese neki broj u intervalu od 2 do 10. Ako korisnik unese pogrešan broj, ispisuje se upozorenje i korisnik treba da ponovo unese broj. Nakon unosa ispravnog broja, program na izlazu daje tablicu množenja za unijeti broj. Na primjer, ako korisnik unese broj 3, na izlazu program daje tablicu množenja za broj 3, od 3×1 do 3×10 . Nakon unosa jednog broja, program treba da pita korisnika da li želi da unese drugi broj. Ako je odgovor ‘D’, postupak se ponavlja, a ako je odgovor ‘N’ program završava sa radom. Ako odgovor nije niti ‘D’ niti ‘N’, ispisuje se upozorenje, i od korisnika se zahtijeva da dâ smislen odgovor, sve dok ne unese ‘D’ ili ‘N’. Podržana su i velika i mala slova u odgovorima.

```

#include <iostream>
#include <cctype>

using namespace std;

int main() {
    char odgovor;
    do {
        int broj;
        do {
            cout << "Unesite broj između 2 i 10: ";
            cin >> broj;
            if(broj < 2 || broj > 10) cout << "Neispravan broj!\n";
        } while (broj < 2 || broj > 10);
        for(int i = 1; i <= 10; i++)
            cout << broj << " x " << i << " = " << broj * i << endl;
        cout << "Želite li unijeti novi broj? ";
        bool dobar_odgovor;
        do {
            cin >> odgovor;
            cin.ignore(10000, '\n')
            odgovor = toupper(odgovor);
            dobar_odgovor = odgovor == 'D' || odgovor == 'N';
            if(!dobar_odgovor) cout << "Odgovorite sa 'D' ili 'N'!\n";
        } while (!dobar_odgovor);
    } while (odgovor == 'D');
    return 0;
}

```

Obratite pažnju na to da je u program uvedena *logička promjenljiva* “`dobar_odgovor`”, kao i na način kako je ona iskorištena. Također, obratite pažnju i na mjesto gdje su deklarirane pojedine promjenljive. Kao što je već rečeno, dobro je truditi se da njihovo vrijeme života bude što je god moguće kraće, ali trebamo paziti da promjenljiva ne prestane postojati na mjestu gdje je njeno postojanje još uvijek potrebno. Na primjer, mada na prvi pogled izgleda da je promjenljiva “`odgovor`” potrebna samo unutar tijela posljednje unutrašnje “`do`”–“`while`” petlje, ona je zapravo potrebna i unutar uvjeta spoljašnje “`do`”–“`while`” petlje, tako da je moramo deklarirati prije početka spoljašnje “`do`”–“`while`” petlje!

Česti su slučajevi u kojima se kao tijelo “`for`” petlje javlja nova “`for`” petlja. Ovo se naročito često javlja kod obrade podataka organiziranih u formi *tablica*. Na primjer, sljedeći program, koji koristi dvije “`for`” petlje, jednu unutar druge, ispisuje na ekran tablicu množenja za sve brojeve od 1 do 10:

```

#include <iostream>
using namespace std;

int main() {
    for(int i = 1; i <= 10; i++) {
        for(int j = 1; j <= 10; j++) {
            cout.width(5);
            cout << i * j;
        }
        cout << endl;
    }
    return 0;
}

```

Kada pokrenemo ovaj program, ispis će izgledati ovako:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Ovu tablicu možemo još malo “uljepšati” sljedećim programom:

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

using namespace std;

int main() {
    for(int i = 1; i <= 10; i++) {
        for(int j = 1; j <= 10; j++) cout << "-----";
        cout << "+\n";
        for(int j = 1; j <= 10; j++) cout << " | " << setw(4) << i * j;
        cout << "| \n";
    }
    for(int j = 1; j <= 10; j++) cout << "-----";
    cout << "+\n";
    return 0;
}
```

Ovaj program proizveće sljedeći ispis:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Primijetimo da u ovom programu zapravo imamo tri neovisne promjenljive nazvane “*j*” od kojih svaka živi samo unutar svoje pripadne petlje. Ovo smo naravno mogli izbjegći deklariranjem jedinstvene promjenljive “*j*” *izvan* petlji, ali se smatra da je ovakvo rješenje bolje, jer brojačka promjenljiva svake od

navedenih petlji zapravo i ne treba da postoji nakon njenog završetka. Također, obratite pažnju na sekvencu naredbi:

```
for(int j = 1; j <= 10; j++) cout << "-----";
cout << "+\n";
```

Ona se mogla zamijeniti sljedećom (rogobatnom) naredbom za ispis:

```
cout << "+-----+\n";
```

Postoje slučajevi kada je potrebno nasilno prekinuti petlju prije ispunjenja uvjeta za kraj petlje. Za tu svrhu možemo koristiti naredbu “**break**” koja bezuvjetno napušta tijelo petlje unutar kojeg je upotrebljena (ova naredbu smo također koristili u kombinaciji sa naredbom “**switch**”). Upotrebu ove naredbe prvo ćemo ilustrirati ponovo na primjeru sabiranja unesenih brojeva sve dok se ne unese negativan broj (poučnije je isti zadatak rješiti na nekoliko načina nego nekoliko zadataka na isti način):

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

int main() {
    cout << "Unesite brojeve koji će se sabirati "
        << "- negativan broj označava prekid.\n\n";
    int suma(0);
    while(true) {
        int broj;
        cout << "Unesite broj: ";
        cin >> broj;
        if(broj < 0) break;
        suma += broj;
    }
    cout << "\nSuma unesenih brojeva je: " << suma << endl;
    return 0;
}
```

U ovom primjeru upotrijebili smo konstrukciju “**while (true)**” koja očigledno predstavlja petlju bez (prirodnog) izlaza, jer je njen “uvjet” uvijek “tačan” (inače, za realizaciju petlji bez izlaza umjesto konstrukcije “**while (true)**” često se koriste i kraće konstrukcije “**while (1)**”, ili “**for (;;)**”). Kada se unese negativan broj, naredbom “**break**” nasilno prekidamo petlju. Ovo rješenje od svih predloženih rješenja ovog problema vjerovatno najdirektnije prati tok misli koji bi čovjek primjenjivao pri rješavanju istog problema. Primijetimo da je u ovom slučaju sasvim moguće da promjenljiva “**broj**” egzistira samo unutar tijela petlje, s obzirom da se uvjet prekida petlje testira također unutar tijela petlje.

Veoma često je nasilno prekidanje petlje sasvim opravdano. Na primjer, zamislimo da želimo da napišemo program koji će ispitati da li je broj unesen sa tastature prost ili složen. Znamo da je broj n prost ako je djeljiv samo sa 1 i n , tj. ako nije djeljiv ni sa jednim prirodnim brojem od 2 do $n-1$. “Najgrublji” način da to provjerimo predstavlja sljedeći program (metod “grube sile”):

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

int main() {
    long int n;
```

```

cout << "Unesite broj: ";
cin >> n;
bool prost(true);
for(long int i = 2; i < n; i++)
    if(n % i == 0) prost = false;
if(prost) cout << "Broj je prost\n";
else cout << "Broj je složen\n";
return 0;
}

```

U ovom programu smo *a priori* prepostavili da je broj prost tako što smo logičku promjenljivu “prost” inicijalizirali na vrijednost “**true**”, nakon čega u petlji ispitujemo da li je broj djeljiv sa svim brojevima od 2 do $n-1$. Ukoliko ustanovimo djeljivost, ovu promjenljivu postavljamo na vrijednost “**false**”, čime signaliziramo da broj nije prost. Ukoliko se cijela petlja izvršila a broj nije bio djeljiv ni sa jednim ispitivanim brojem, tijelo naredbe “**if**” neće se izvršiti niti jedanput, tako da će promjenljiva “prost” ostati onakva kakva je bila na početku (tj. tačna), iz čega zaključujemo da je broj *zaista prost*.

Priloženi program je sve samo ne efikasan. Zamislimo da treba ispitati da li je broj 35172969 prost. On to nije (jer je djeljiv sa 3), ali ovako napisan program u petlji ispituje djeljivost sa svim brojevima od 2 do 35172968, čime se tijelo petlje izvršava 35171967 puta. Daleko je prirodnije prekinuti izvršavanje petlje čim se ustanovi djeljivost. Ovim dolazimo do sljedećeg programa, u kojem je upotrebljena naredba “**break**”:

```

#include <iostream>
using namespace std;

int main() {
    long int n;
    cout << "Unesite broj: ";
    cin >> n;
    bool prost(true);
    for(long int i = 2; i < n; i++) {
        if(n % i == 0) {
            prost = false;
            break;
        }
    }
    if(prost) cout << "Broj je prost\n";
    else cout << "Broj je složen\n";
    return 0;
}

```

Za ispitivani broj 35172969 sa ovakvim programom petlja će se izvršiti samo dva puta, jer se petlja prekida čim ustanovimo djeljivost sa 3. Međutim, za broj 37665427 (koji *jeste* prost) petlja se i dalje mora izvršiti 37665425 puta. Ovaj program može se dalje drastično ubrzati uz malu pomoć elementarne matematike. Lako je provjeriti da ako broj n ima djelitelj koji je veći od \sqrt{n} , on takođe mora imati i djelitelj koji je manji od \sqrt{n} , iz čega slijedi da je dovoljno da gornja granica petlje bude \sqrt{n} a ne $n-1$ (razmislite *koliko se puta* izvrši petlja u ovom programu ako je n jednak 2 ili 3):

```

#include <iostream>
#include <cmath>

using namespace std;

int main() {
    long int n;
    cout << "Unesite broj: ";

```

```

    cin >> n;
    bool prost(true);
    for(long int i = 2; i <= sqrt(double(n)); i++)
        if(n % i == 0) {
            prost = false;
            break;
        }
    if(prost) cout << "Broj je prost\n";
    else cout << "Broj je složen\n";
    return 0;
}

```

Koliko smo ubrzanje postigli nije teško izračunati. Pri ispitivanju prostog broja 37665427 petlja će se sa ovakvim programom izvršiti 6136 puta, što predstavlja ubrzanje od preko 6138 puta! Na PC računaru sa Pentium I procesorom na 133 MHz i Turbo C++ kompjlerom, ispitivanje ovog prostog broja sa prvim programom traje 1 minutu i 55 sekundi, dok je pomoću posljednjeg programa trajanje ispitivanja reda 2 stotinke sekunde! To i dalje nije sve što možemo uraditi po pitanju efikasnosti programa. Primijetimo da se prilikom ispitivanja uvjeta “**for**” petlje korijen iz jednog te istog broja (n) neprestano izračunava svaki put kada se provjerava uvjet petlje. Računanje korijena nije naročito brza operacija, tako da je znatno efikasnije taj korijen izračunati *izvan petlje* i pridružiti ga nekoj promjenljivoj, kao u sljedećem programu:

```

#include <iostream>
#include <cmath>

using namespace std;

int main() {
    long int n;
    cout << "Unesite broj: ";
    cin >> n;
    bool prost(true);
    int korijen = sqrt(double(n));
    for(long int i = 2; i <= korijen; i++)
        if(n % i == 0) {
            prost = false;
            break;
        }
    if(prost) cout << "Broj je prost\n";
    else cout << "Broj je složen\n";
    return 0;
}

```

Ovim smo postigli ubrzanje od još nekoliko puta, zavisno od toga koliko brzo konkretan računar izračunava korijen. Doduše, postoje inteligentni kompjajleri koji će ukoliko primijete da se unutar petlje stalno izračunava vrijednost jednog te istog izraza, automatski prebaciti njegovo izračunavanje ispred petlje, odnosno pri prevođenju će automatski obaviti transformaciju koju smo mi u prethodnom programu eksplisitno izvršili. Kod takvih kompjajlera, prethodni primjer neće pokazati nikakvo ubrzanje u odnosu na program naveden prije njega, međutim prilikom programiranja nije se dobro previše uzdati u inteligenciju kompjajlera.

Sljedeći korak ka ubrzanju (dvostrukom) je uočavanje činjenice da posebno možemo ispitati djeljivost sa 2, a ukoliko broj nije djeljiv sa 2, dovoljno je ispitivati djeljivost samo sa *neparnim brojevima* (od 3 nadalje), jer ukoliko je broj djeljiv sa nekim *parnim brojem*, sigurno je mora biti djeljiv i sa 2, tako da sigurno nije prost (oprez: broj 2 *jeste* prost broj iako je djeljiv sa 2). Ova osobina iskorištena je u

sljedećem programu. Interesantno je analizirati kako radi ovaj program za različite ulazne podatke. Pri tome su naročito interesantni ulazni podaci 2, 3, 5 i 7, jer iako u sva četiri slučaja tok programa vodi do naredbe “**for**”, petlja neće biti izvršena niti jedanput!

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

using namespace std;

int main() {
    long int n;
    cout << "Unesite broj: ";
    cin >> n;
    bool prost(true);
    if(n != 2 && n % 2 == 0) prost = false;
    else {
        int korijen = sqrt(n);
        for(long int i = 3; i <= korijen; i += 2)
            if(n % i == 0) {
                prost = false;
                break;
            }
    }
    if(prost) cout << "Broj je prost\n";
    else cout << "Broj je složen\n";
    return 0;
}
```

Dobiveni program je, za 8-cifrene ulazne podatke, preko 30000 puta brži od programa od kojeg smo krenuli (i to je uglavnom najviše što možemo postići bez ulaženja u višu matematičku teoriju brojeva). Zbog toga, prije nego što prihvativmo rješenje “grubom silom” uvijek treba razmisliti može li se problem riješiti bolje i efikasnije. Zapamtitte da je cilj programiranja ne samo da napravimo program koji daje tačno rješenje, nego se to tačno rješenje pronađe u razumnom vremenu! Lako je navesti primjere problema koji se mogu riješiti u veoma kratkom vremenu, a čije bi rješavanje “grubom silom” trajalo više hiljada godina i na najbržim postojećim računarima! Inače, interesantno je napomenuti da je problem ispitivanja da li su pojedini veoma veliki brojevi (50 cifara i više) prosti ili ne izuzetno značajan u praksi (naročito u kriptografiji), i da je problem *efikasnog* ispitivanja veoma velikih brojeva na prostost još uvijek otvoren.

Izlaganje o naredbi “**break**” završićemo konstatacijom da je ovu naredbu uvijek *moguće izbjegći* pisanjem složenijih uvjeta za izlazak iz petlje koji u sebi uključuju i logičke promjenljive (ponekad je takve promjenljive potrebno dodatno uvesti u program da bi se izbjegla “**break**” naredba). Ipak, korištenje naredbe “**break**” obično dovodi do jasnijih programa. Kao primjer, sljedeći program je funkcionalno potpuno ekvivalentan (i podjednako efikasan) kao i prethodni program, ali *ne koristi* naredbu “**break**”:

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

using namespace std;

int main() {
    long int n;
    cout << "Unesite broj: ";
    cin >> n;
    bool prost(true);
    if(n != 2 && n % 2 == 0) prost = false;
```

```
    else {
        int korijen = sqrt(n);
        for(long int i = 3; i <= korijen && prost; i += 2)
            if(n % i == 0) prost = false;
    }
    if(prost) cout << "Broj je prost\n";
    else cout << "Broj je složen\n";
    return 0;
}
```

Trik koji je ovdje korišten zasnovan je na činjenici da je uvjet petlje tako modificiran primjenom logičkog operatora “`&&`” tako da postane netačan čim promjenljiva “`prost`” poprimi vrijednost “`false`”, odnosno čim se otkrije da broj nije prost.

12. Nizovi i vektori

Dosada uvedeni tipovi podataka poput “**int**”, “**double**”, “**char**”, “**bool**” itd. omogućavaju predstavljanje *samo pojedinačnih vrijednosti*, u smislu da *jedna promjenljiva* u jednom trenutku može pamtiti *samo jednu vrijednost*. Na primjer, ukoliko su date sljedeće deklaracije:

```
int brojac;
double temperatura;
char slovo;
```

njima odgovara sljedeća slika u memoriji računara:

brojac	temperatura	slovo
--------	-------------	-------

S druge strane, pri rješavanju problema iz stvarnog svijeta često se javlja potreba za predstavljanjem *skupine vrijednosti* koje su sve *istog tipa*, na primjer:

- Prosječne količine padavina za svaki mjesec;
- Ocjene svih učenika u razredu;
- Stanje prodaje za svaki dan u sedmici, itd.

U načelu, za predstavljanje skupine vrijednosti principijelno je moguće deklarirati skupinu neovisnih promjenljivih. Na primjer, za predstavljanje prosječne količine padavina za svaki mjesec, principijelno je moguće deklarirati 12 različitih promjenljivih “*padavine_1*”, “*padavine_2*” itd. do “*padavine_12*”. Međutim, na opisani način nemoguće je obrađivati podatke iz ovih 12 nezavisnih promjenljivih i na kakav sistematičan način. Na primjer, nije moguće ispisati njihov sadržaj petljom poput

```
for(int i = 1; i <= 12; i++) cout << padavine_i << endl;
```

jer bi ovakva petlja, umjesto ispisa redom promjenljivih “*padavine_1*”, “*padavine_2*” itd. zapravo 12 puta ispisala vrijednost jedne te iste promjenljive nazvane “*padavine_i*”, pod uvjetom da takva postoji (u protivnom bi bila prijavljena greška).

Da bi se omogućila sistematična obrada skupina vrijednosti srodne prirode, jezik C++ omogućava razne načine za čuvanje skupina vrijednosti unutar *jedne promjenljive*. Najelementarniji način za predstavljanje skupina vrijednosti istog tipa je upotreba tzv. *nizova* (engl. *arrays*), koje ćemo razmotriti u ovom poglavlju. Druge načine za predstavljanje skupina vrijednosti istog tipa, koji su u osnovi najčešće također izvedeni iz nizova, razmotrićemo kasnije. Isto vrijedi i za predstavljanje skupina međusobno povezanih vrijednosti koje *nisu istog tipa*. U našoj literaturi se, pored termina *niz*, susreće i termin *polje*, mada takva terminologija može dovesti do zabune, jer se poljima također nazivaju i dijelovi složenih struktura podataka nazvanih *slogovi*, koje ćemo upoznati kasnije.

Niz možemo kreirati pomoću deklaracije poput:

```
double padavine[12];
```

Ova deklaracija promjenljive kreiraće promjenljivu “*padavine*” koja predstavlja niz koji može čuvati

najviše 12 vrijednosti, pri čemu je svaka od tih vrijednosti tipa “**double**”. Ovome odgovara sljedeća memorijска слика:



Pojedinačni elementi niza su određeni svojim *indeksom*. Indeks se piše u *uglastim zagradama, nakon navođenja imena niza*. Pri tome se svaki element niza sam za sebe ponaša poput individualnih promjenljivih (preciznije, elementi niza predstavljaju *l-vrijednosti*). Drugim riječima, bilo koja operacija koja se može vršiti nad običnim promjenljivim odgovarajućeg tipa, može se vršiti i nad *invididualnim elementima niza* istog tipa. Na primjer,

```
padavine[4] = 20.3;  
cout << padavine[7];
```

Kao i u slučaju običnih promjenljivih, elementi niza *nemaju nikakvu precizno određenu vrijednost* sve dok im se eksplicitno ne dodijeli vrijednost, bilo pomoću operatora dodjele “=”, bilo pomoću unosa sa tastature. Alternativno, postoji mogućnost da izvrši zadavanje početnih vrijednosti elemenata niza (inicijalizacija) uporedo sa deklaracijom, o čemu ćemo govoriti nešto kasnije u ovom poglavlju.

Prilikom *deklaracije nizovnih promjenljivih*, unutar uglastih zagrada mora se nalaziti nešto čija je vrijednost *konstantna i poznata prije početka izvršavanja programa*, poput *broja, prave konstante* (deklarirane sa oznakom “**const**” i inicijalizirane *brojem ili konstantnim izrazom*) ili *konstantnog izraza*. Tako, ako je “n” promjenljiva, deklaracija poput

```
double padavine[n];
```

nije dozvoljena, dok je ista deklaracija dozvoljena u slučaju da je “n” *prava konstanta*. Treba napomenuti da izvjesni kompjajleri za C++ (na primjer, kompjajleri iz GNU porodice kompjajlera) dopuštaju da se u deklaraciji nizovnih promjenljivih unutar uglaste zgrade upotrijebe proizvoljni numerički izrazi. Međutim, treba napomenuti da se u tom slučaju radi o *nestandardnom proširenju jezika (ekstenziji)* implementiranom od strane autora kompjajlera koje *nije predviđeno standardom jezika C++*. Stoga se na takva proširenja *ne treba oslanjati* ukoliko želimo da program koji razvijamo bude neovisan od upotrijebljenog kompjajlera. Standard jezika C++ predviđa druge načine da se postigne isti efekat (pomoću tzv. *dinamičke alokacije memorije* ili pomoću *standardnih kontejnerskih klasa*). O ovim načinima ćemo govoriti kasnije.

S druge strane, kada *pristupamo elementima niza*, indeks u zagradi može biti proizvoljan *aritmetički izraz cjelobrojnog tipa*, ili *tipa koji se može promovirati u cjelobrojni tip* (npr. biće prihvaćeni i izrazi znakovnog ili logičkog tipa, koji će biti automatski konvertirani u cjelobrojni tip, ali ne i izrazi realnog tipa). Tako je, na primjer, uz pretpostavku da je “n” cjelobrojna promjenljiva, sljedeći izraz potpuno legalan:

```
padavine[2 * n - 3] = 13.7;
```

Veoma je važno uočiti da broj u uglastoj zagradi *nema isto značenje* prilikom *deklaracije nizovne promjenljive* i prilikom *pristupa njenim elementima*. Naime, taj broj prilikom deklaracije *označava*

maksimalan broj elemenata niza, dok prilikom pristupa elementima niza označava *indeks elementa niza kojem želimo pristupiti*.

U jeziku C++ nisu direktno podržane operacije koje bi se izvršavale nad nizovnim promjenljivim *kao cjelinom* (bar ne sa standardnim nizovnim tipovima), nego samo nad *individualnim elementima niza* (iako u standardnim bibliotekama jezika C++ postoje *funkcije* koje rade sa nizovima kao cjelinama). Na primjer, ukoliko želimo ispisati sve elemente niza, postaviti sve elemente niza na neku vrijednost (npr. nulu), ili unijeti sve elemente niza sa tastature, ne možemo pisati naredbe poput sljedećih:

```
cin >> padavine;  
padavine = 0;  
cout << padavine;
```

Isto tako, ukoliko su npr. “niz_1” i “niz_2” dvije nizovne promjenljive sa istim brojem elemenata (npr. 100) i istim tipom elemenata (npr. “**int**”), nije moguće iskopirati sve elemente niza “niz_2” u odgovarajuće elemente niza “niz_1” jednom naredbom poput

```
niz1 = niz2;
```

Umjesto toga, željene operacije potrebno je obaviti *element po element*, najčešće uz pomoć “**for**” petlje. Na primjer, sve elemente niza “padavine” možemo postaviti na nulu pomoću naredbe

```
for(int mjesec = 0; mjesec < 12; mjesec++) padavine[mjesec] = 0;
```

dok sve elemente niza “niz_2” možemo iskopirati u odgovarajuće elemente niza “niz_1” pomoću sljedeće naredbe:

```
for(int i = 0; i < 100; i++) niz_1[i] = niz_2[i];
```

Kasnije ćemo vidjeti da u standardnoj biblioteci “*algorithm*” postoje funkcije “*fill*” i “*copy*” koja obavljuju upravo opisane zadatke, samo na malo drugačiji način. Primijetimo da smo u oba slučaja unutar uvjeta “**for**” petlje upotrijebili operator “<” a ne “<=”. To je zbog činjenice da su posljednji indeksi navedenih nizova respektivno 11 i 99, a ne 12 i 100, s obzirom da numeracija indeksa započinje od *nule*, a ne od *jedinice*. Razumije se da smo mogli upotrijebiti i operator “<=” pišući naredbe poput

```
for(int mjesec = 0; mjesec <= 11; mjesec++) padavine[mjesec] = 0;
```

odnosno

```
for(int i = 0; i <= 99; i++) niz_1[i] = niz_2[i];
```

Međutim, upotrebljom operatora “<” jasnije se ističe veza između broja elemenata niza i njihovih indeksa (indeks uvijek mora biti *strogo manji* od broja elemenata niza).

Treba napomenuti da činjenica da se sa nizovima ne može baratati *kao cjelinom* nego samo sa njegovim *individualnim elementima* ne znači da se ime nizovne promjenljive ne može upotrijebiti samo za sebe, bez navođenja indeksa. Međutim, takve konstrukcije ne rade ono što bi se od njih moglo očekivati na prvi pogled. Na primjer, sljedeća naredba, u kojoj se pojavljuje ime nizovne promjenljive “padavine”

```
cout << padavine;
```

sasvim je legalna u jeziku C++, ali ne radi ono što bi se moglo pomisliti (odnosno ne ispisuje sve elemente niza “padavine” na ekran). Umjesto toga, kao rezultat izvršavanja ove naredbe na ekranu će biti isписан

neki heksadekadni broj (koji predstavlja adresu lokacije u memoriji od koje započinje smještanje elemenata ovog niza)! Naime, kasnije ćemo vidjeti da se ime svake nizovne promjenljive upotrijebljeno samo za sebe (tj. bez navođenja indeksa) automatski konvertira u tzv. *pokazivač na prvi element niza*, a pokazivači se na ekran ispisuju u vidu adresa lokacija u memoriji na koje pokazuju (u heksadekadnom brojnom sistemu). O pokazivačima ćemo detaljno govoriti u kasnijim poglavljima, a na ovom mjestu je samo potrebno da zapamtimo da upotreba imena nizovnih promjenljivih bez navođenja indeksa neće dati očekivane rezultate.

Sljedeći primjer, koji predstavlja program koji traži unos pet cijena i ispisuje ih kao spisak sa naslovom, ilustrira rad sa nizovima u kombinaciji sa naredbom “**for**”, kao i neke probleme uzrokovane činjenicom da indeksiranje nizova započinje od nule a ne od jedinice:

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

using namespace std;

int main() {
    double cijene[5];
    cout << "Unesi pet cijena:\n";
    for(int i = 0; i < 5; i++) cin >> cijene[i];
    cout << endl <<
        "Spisak cijena\n"
        "-----\n"
        "Stavka      Cijena\n";
    for(int i = 0; i < 5; i++)
        cout << setw(2) << i + 1 << "." << setw(13) << cijene[i] << endl;
    return 0;
}
```

Zbog činjenice da se indeksi niza koji ima N elemenata kreću od 0 do N-1, a ne od 1 do N, u ovom primjeru unesene cijene su smještene u elemente niza sa indeksima 0, 1, 2, 3 i 4, odnosno prva cijena smještena je u “nulti” element niza, itd. Ovakva numeracija, koja je sasvim logična sa aspekta organizacije podataka u računarskoj memoriji, veoma je neuobičajena sa aspekta čovjeka, koji je navikao da vrši indeksiranje počevši od 1 nadalje. Zbog toga je u prethodnom programu, prilikom ispisa cijena, u koloni “stavka” indeks “vještački” uvećan za 1, tako da će brojevi ispisani u ovoj koloni biti 1, 2, 3, 4 i 5, kao što je običajeno. Generalno se prilikom svakog ispisa indeksa niza na ekran, on obično uvećava za 1 za bi se ispis prilagodio obliku na koji je čovjek navikao.

Kao alternativno rješenje, neki programeri deklariraju niz koji sadrži jedan element više nego što je potrebno, pri čemu element sa indeksom 0 *nikada ne koriste* (tj. ostavljaju ga *neiskorištenim*), kao u sljedećem programu, koji je funkcionalno ekvivalentan prethodnom programu:

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

using namespace std;

int main() {
    double cijene[6];
    cout << "Unesi pet cijena:\n";
    for(int i = 1; i <= 5; i++) cin >> cijene[i];
    cout << endl <<
        "Spisak cijena\n"
        "-----\n"
        "Stavka      Cijena\n";
    for(int i = 1; i <= 5; i++)
```

```

        cout << setw(2) << i << "." << setw(13) << cijene[i] << endl;
    return 0;
}

```

Međutim, postoji više razloga zbog čega ovakvo rješenje nije osobito preporučljivo. Pored činjenice da na taj način nepotrebno trošimo memoriju za jedan element niza više, neke naprednije tehnike programiranja koje se oslanjaju na upotrebu tzv. *pokazivača* i *pokazivačke aritmetike* očekuju da su elementi niza zaista smješteni počev od elementa sa indeksom 0. U suštini, ako je indeksacija koja započinje od nule svojstvo jezika C++, za dobro ovladavanje ovim jezikom bolje se *nativići* na ovu činjenicu i *prilagoditi joj se*, nego *bježati od nje*.

Sličnu situaciju imamo i u sljedećem programu, koji učitava prosječnu temperaturu za svaki od 12 mjeseci, i prikazuje unesene temperature na ekranu u obliku tabele. U ovom programu, kada nas program pita "Unesi temperaturu u toku 1. mjeseca:" unesena vrijednost se zapravo smješta u element čiji je indeks 0, a ne 1, jer je indeks isписан na ekran (tj. promjenljiva "mjesec") vještački isписан za 1 veći nego što zaista jeste:

```

#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    double temperature[12];
    for(int mjesec = 0; mjesec < 12; mjesec++) {
        cout << "Unesi prosječnu temperaturu u toku "
            << mjesec + 1 << ". mjeseca: ";
        cin >> temperature[mjesec];
    }
    cout << "\nMjesec      Temperatura\n";
    for(int mjesec = 0; mjesec < 12; i++)
        cout << setw(6) << mjesec + 1 << setw(14)
            << temperature[mjesec] << endl;
    return 0;
}

```

Bitno je napomenuti da nizove treba koristiti samo u slučajevima u kojima je *zaista neophodno* pamćenje čitave skupine vrijednosti. Tako, ukoliko je potrebno sabrati 100 brojeva koji se unose sa tastature, nizovi nisu potrebni, s obzirom da nije potrebno pamtitи sve unesene vrijednosti, već samo upravo uneseni broj i sumu do tada unesenih brojeva. Sličnu situaciju imamo i prilikom traženja najveće ili najmanje vrijednosti između skupine brojeva unesenih sa tastature, što smo već ranije demonstrirali. Međutim, ukoliko je potrebno *prebrojati* koliko se puta u skupini brojeva unesenih sa tastature javlja najveća vrijednost, nizovi su potrebni. Naime, koja je najveća vrijednost možemo znati tek nakon što unesemo *posljednji broj*, nakon čega trebamo prebrojati koliko puta se među unesenim brojevima pojavila ta vrijednost. Međutim, to ne možemo uraditi ukoliko nismo prethodno *zapamtili sve unesene brojeve*. Ovo će detaljnije biti ilustrirano u jednom od programa prikazanih kasnije u ovom poglavljju.

Generalno, nizove (ili neku srodnu strukturu podataka, koje će biti kasnije razmotrene) treba koristiti kad god je potrebno izvršiti neku obradu podataka za koju je potrebno u jednom trenutku poznavati *čitavu skupinu* podataka. Na primjer, nemoguće je ispisati skupinu unesenih brojeva nakon završetka njihovog unosa ukoliko njihove vrijednosti nismo prethodno pohranili u memoriju. Dobru ilustraciju pruža i sljedeći program, koji zahtijeva unos broja polaznika koji su prisustvovali nekom petodnevnom kursu za svaki dan pojedinačno, a zatim prikazuje *linijski dijagram* (engl. *line chart*) prisustva, sastavljen od zvjezdica:

```

#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    int broj_polaznika[5];
    for(int dan = 0; dan < 5; dan++) {
        cout << "Unesi broj polaznika u toku " << dan + 1 << "dana: ";
        cin >> broj_polaznika[dan];
    }
    cout << "Dijagram prisustva\n"
        "-----\n"
        "Dan\n";
    for(int dan = 0; dan < 5; dan++)
        cout << setw(2) << dan + 1 << " " << setfill('*')
            << setw(broj_polaznika[dan]) << "" << endl;
    return 0;
}

```

Ako su, na primjer, uneseni podaci bili 16, 13, 11, 18 i 15, nakon njihovog unosa dobćemo prikaz kao na sljedećoj slici:

Dijagram prisustva	

Dan	
1	*****
2	*****
3	*****
4	*****
5	*****

U ovom primjeru za ispis zvjezdica na interesantan način je iskorišten manipulator “`setfill`”. Alternativno, zvjezdice smo mogli ispisati upotrebom dodatne “`for`” petlje. U tom slučaju, posljednja petlja u prethodnom programu mogla bi izgledati npr. ovako:

```

for(int dan = 0; dan < 5; dan++)
    cout << setw(2) << dan + 1 << " ";
    for(zvjezdica = 1; zvjezdica <= broj_polaznika[dan]; zvjezdica++)
        cout << "*";
    cout << endl;
}

```

U praksi često ne znamo unaprijed koliko je elemenata potrebno smjestiti u niz. Nažalost, jedini način da ovaj problem riješimo uz upotrebu *klasičnih nizova* je da prilikom deklaracije niza zadamo *najveći očekivani broj elemenata niza*, pri čemu nas niko ne primorava da zaista moramo iskoristiti sve elemente. Neka, na primjer, određen broj studenata (ali ne više od 50) polaze neki ispit. Za svaki rad dobija se od 0 do 100 poena. Sljedeći program prvo pita korisnika koliko je kandidata pristupilo ispitu, zatim učitava broj poena za svakog kandidata posebno, i konačno, prikazuje na ekranu rezultate ispita (položio/pao) za svakog kandidata, uz pretpostavku da je za prolaz potrebno barem 55 poena. U program je ugrađena kontrola da li je broj kandidata zaista u dozvoljenom opsegu (od 1 do 50). S druge strane, radi jednostavnosti, nije ugrađena provjera ispravnosti broja unesenih poena za svakog kandidata:

```

#include <iostream>
#include <iomanip>

```

```

using namespace std;

int main() {
    const int MaxBroj(50), PragZaProlaz(55);
    int broj_kandidata;
    do {
        cout << "Koliko kandidata je pristupilo ispitu? ";
        cin >> broj_kandidata;
        if(broj_kandidata < 1 || broj_kandidata > MaxBroj)
            cout << "Broj mora biti od 1 do " << MaxBroj << "!\n";
    } while(broj_kandidata < 1 || broj_kandidata > MaxBroj);
    int poeni[MaxBroj];
    for(int i = 0; i < broj_kandidata; i++) {
        cout << "Unesi broj poena za " << i + 1 << "kandidata: ";
        cin >> poeni[i];
    }
    cout << endl << "Kandidat Status\n";
    for(int i = 0; i < broj_kandidata; i++) {
        cout << setw(7) << i + 1 << ".";
        if(poeni[i] >= PragZaProlaz) cout << "POLOŽIO\n";
        else cout << "PAO\n";
    }
    return 0;
}

```

Očigledan nedostatak rješenja u kojem se deklarira najveći očekivani broj elemenata niza je činjenica da se tako nepotrebno rezervira memorija za elemente koji se ne koriste. U slučajevima kada ne možemo ni grubo pretpostaviti koliki bi maksimalan broj elemenata bio potreban, jedino rješenje (za sada) je da rezerviramo veoma veliki broj elemenata niza, što je zaista veliko rasipanje memorije. Na primjer, ukoliko deklariramo niz od 1000000 cijelih brojeva, zauzimamo (vjerovatno bespotrebno) skoro 4 megabajta memorije, uz pretpostavku da jedan element niza (jedan cijeli broj) zauzima 4 bajta!

Možemo primijetiti da je u prethodnom programu veoma lako izvršiti prilagođavanje za bilo koji maksimalno dozvoljeni broj studenata, s obzirom da se sve promjene svode samo na promjenu vrijednosti konstante "MaxBroj". Ovakvu praksu bi trebalo primjenjivati u svim programima.

Neki programeri prakticiraju upotrebu različitih trikova kojima je moguće na kasnijim mjestima u programu odrediti kapacitet niza (odnosno maksimalan broj elemenata koje niz može primiti) zadani pri deklaraciji. Jedan od najčešće korištenih trikova prikazan je u sljedećem programskom isječku:

```

int niz[100];
...
cout << "Broj elemenata je " << sizeof niz / sizeof niz[0];

```

Kako radi ovaj isječak? Podsjetimo se da operator "**sizeof**" daje kao rezultat broj bajta koje zauzima tip, promjenljiva ili izraz naveden kao njegov argument. Tako, "**sizeof niz**" daje broj bajtova u memoriji koje zauzima *čitav niz*, što je jednak broju elemenata niza pomnoženom sa brojem bajtova koje zauzima *jedan element niza*. Međutim, tu veličinu lako možemo saznati pomoću izraza "**sizeof niz[0]**", tako da dijeljenjem ove dvije vrijednosti dobijamo broj elemenata niza. U konkretnom primjeru, umjesto "**sizeof niz[0]**" mogli smo upotrijebiti "**sizeof(int)**", ali prikazano rješenje je *neovisno od tipa elemenata niza*. Mnogi C++ programeri često koriste opisani trik. Na primjer, sljedeća petlja ispisuje *sve* elemente niza "niz", bez potrebe da apriori znamo broj njegovih elemenata:

```
for(int i = 0; i < sizeof niz / sizeof niz[0]; i++)
    cout << niz[i] << endl;
```

Međutim, postoji dosta razloga zbog kojih ovakvi trikovi nisu preporučljivi. Prvo, ovaj trik ne radi ispravno za nizove koji predstavljaju *formalne parametre funkcija*, o čemu ćemo govoriti kasnije, kao i za *dinamički alocirane nizove* (o kojima ćemo također govoriti kasnije). Također, ovaj trik ne daje ispravne rezultate ukoliko se primijeni na neke kontejnerske klase slične nizovima. Zbog svega rečenog, nepažljiva primjena ovog trika može dovesti do ozbiljnih zabuna. Stoga se preporučuje da se broj deklariranih elemenata niza ne određuje programski, već da se za njega rezervira *posebna konstanta*, kao što je urađeno u prethodnom programu.

Veoma je važno naglasiti da C++ ne provjerava da li se indeks niza prilikom pristupa elementima niza nalazi u dozvoljenim granicama. Ovo je urađeno zbog efikasnosti, jer bi provjera ispravnosti indeksa pri svakom pristupu nekom elementu niza osjetno usporila rad sa nizovima. Nažalost, to znači ukoliko greškom upotrijebimo indeks koji je veći ili jednak broju elemenata niza, ili čak negativan indeks, neće biti prijavljena nikakva greška, nego program neće raditi ispravno, često uz veoma neugodne posljedice. Razmotrimo, na primjer, sljedeću sekvencu naredbi:

```
int niz[5];
for(int i = 0; i < 30000; i++) niz[i] = 1000;
```

U ovom primjeru, deklaracijom “`int niz[5]`” u memoriji je rezerviran prostor za samo 5 elemenata niza (sa indeksima 0, 1, 2, 3 i 4). S druge strane, u “`for`” petlji pokušavamo smjestiti broj 1000 u sve elemente niza sa indeksima od 0 do 29999. Pri tome, kako memorijski prostor za elemente sa indeksima od 5 do 29999 *nije rezerviran*, program će broj 1000 smještati u dijelove memorije koje uopće ne pripadaju nizu “`niz`”. Praktično je nemoguće predvidjeti šta se u tim dijelovima memorije zaista nalazi. Možda se tamo nalazi sadržaj nekih drugih promjenljivih, tako da će ove naredbe poremetiti sadržaj promjenljivih koje se u njima nalaze. Možda je upravo u tim dijelovima memorije smješten i sam program koji se izvršava, koji će nakon izvršenja navedenih naredbi biti potpuno “izbombardiran”. Dalje, postoji mogućnost da se u tim dijelovima memorije nalaze neki podaci koji su neophodni za sam rad operativnog sistema i samog računara. U svakom slučaju, krajnji efekat će biti potpuno nepredvidljiv.

Kod starijih operativnih sistema (poput MS DOS-a) greške ovakvog tipa često su dovode do pada operativnog sistema ili potpune blokade računara, nakon čega je neophodno njegovo resetiranje (uz gubitak programa koji smo radili ukoliko ga prethodno nismo snimili). Noviji operativni sistemi (npr. operativni sistemi iz MS Windows serije) posjeduju zaštitu od “bombardiranja” dijelova memorije koji ne pripadaju samom programu koji se izvršava (nego npr. operativnom sistemu ili nekom drugom programu). Ukoliko se primijete takve akcije, operativni sistem će automatski prekinuti izvršavanje programa koji se “oteo kontroli”, uz ispis odgovarajuće poruke (poput “This program performed illegal operation and will be shut down” ili neke slične). Međutim, i dalje program ima mogućnost da “izbombarduje” sam sebe (ili svoje vlastite promjenljive), i takve akcije mogu dugo ostati neprimijećene, odnosno mogu voditi ka programima koji počinju da rade neispravno tek nakon dužeg vremena. Na ovakve greške treba dobro paziti, jer se obično teško otkrivaju.

Prethodno izlaganje ne treba da uplaši čitatelje i čitateljke, i da ih odvrati od upotrebe nizova. Naprotiv, njihova upotreba je, u mnogim situacijama, naprosto neizbjegljiva. Cilj prethodnog izlaganja je bio upozoravanje na činjenicu da je pri upotrebi nizova potreban nešto veći oprez nego pri upotrebi do sada uvedenih elemenata jezika C++. Naime, nizovi su prvi “zločesti” elementi jezika C++ sa kojima se susrećemo (engl. “*arrays are evil*”). Srećom, jezik C++ omogućava samostalno definiranje novih tipova podataka koji se u potpunosti ponašaju u skladu sa korisnikovim željama, tako da postoji razvijeni izvjesni korisnički definirani tipovi podataka, nazovimo ih npr. *sigurni nizovi* (engl. *safe arrays*), koji se

ponašaju srođno klasičnim nizovima, ali kod kojih se vrši kontrola ispravnosti indeksa pri pristupu njihovim elementima (u slučaju neispravnosti indeksa, automatski dolazi do prekida rada programa uz prijavu greške). Ovakvi tipovi podataka nalaze se u bibliotekama koje *nisu dio standardne biblioteke* koja dolazi uz jezik C++ nego se nabavljaju posebno, tako da o njima nećemo govoriti, niti ćemo ih koristiti. U kasnijim poglavljima ćemo naučiti kako možemo sami razviti ovakve tipove podataka.

Sasvim je moguće definirati i nizove znakova. Sljedeći primjer traži od korisnika da unese neku *petoslovnu riječ*, a zatim ispisuje unesenu riječ u obrnutom poretku slova:

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

int main() {
    const int BrojSlova(5);
    char rijec[BrojSlova];
    cout << "Unesi riječ od " << BrojSlova << " slova: ";
    for(int i = 0; i < BrojSlova; i++) cin >> rijec[i];
    cout << "Riječ izgovorena naopako glasi: ";
    for(int i = BrojSlova - 1; i >= 0; i--) cout << rijec[i];
    return 0;
}
```

Ukoliko ste shvatili kako tačno rade sve upotrebljene naredbe (a pogotovo naredba za unos znakova sa ulaznog toka), lako ćete zaključiti šta će se desiti ukoliko korisnik unese riječ koja ima više ili manje od 5 slova. Naime, ukoliko se unese više od 5 slova, višak će biti *ignoriran*, a ako se unese manje od 5 slova, nakon pritiska na ENTER program će ponovo tražiti unos sa tastature, sve dok se ne prikupi tačno 5 slova. Također, zbog prirode operadora “>>”, jasno je da će eventualno uneseni razmaci biti ignorirani (što je, u slučaju potrebe, moguće izbjegći upotrebo funkcije “*cin.get*”).

Prethodni program je lako prepraviti da radi sa dužim ili kraćim rijećima prostom promjenom vrijednosti konstante “*BrojSlova*”. Međutim, nije preteško napraviti izmjene koje će omogućiti prepravku ovog programa tako da radi sa rijećima (ili općenitije, rečenicama) *proizvoljne dužine*, odnosno čija dužina nije poznata unaprijed. Sve što je potrebno uraditi je čitati znakove redom i smještati ih u niz sve dok se ne dostigne oznaka kraja reda (za tu svrhu moramo koristiti funkciju “*cin.get*”). Istovremeno je potrebno brojati pročitane znakove, da bismo znali koliko je zaista znakova pročitano (u programu koji slijedi za tu svrhu je iskorištena promjenljiva “*broj_znakova*”, koju povećavamo za 1 prilikom smještanja svakog novog znaka u niz). S obzirom da ne znamo unaprijed koliko će znakova biti pročitano, za njihovo smještanje ćemo rezervirati prostor koji je znatno veći od očekivanog broja znakova koji će biti pročitani (1000 u navedenom primjeru, što je određeno vrijednošću konstante “*MaxBrojZnakova*”). Program koji slijedi ilustrira upravo opisanu tehniku:

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

int main() {
    const int MaxBrojZnakova(1000);
    char znak, recenica[MaxBrojZnakova];
    cout << "Unesi rečenicu: ";
    int broj_znakova(0);
    while((znak = cin.get()) != '\n') recenica[broj_znakova++] = znak;
    cout << "Rečenica izgovorena naopako glasi: ";
    for(int i = broj_znakova - 1; i >= 0; i--) cout << recenica[i];
    return 0;
}
```

```
}
```

U ovom programu smo očigledno pretpostavili da unesena rečenica neće biti duža od 1000 znakova. Međutim, ukoliko korisnik ipak unese više od 1000 znakova, doći će do ozbiljnih problema, jer će se prekoračiti dozvoljeni opseg indeksa u nizu “recenica” što, kao što smo već istakli, može dovesti do kraha programa. Ovaj problem se lako može riješiti modifikacijom uvjeta u “**while**“ petlji tako da se ona prekine ukoliko promjenljiva “broj_znakova“ koja broji unesene znakove dostigne vrijednost 1000 (odnosno, vrijednost konstante “MaxBrojZnakova”). Ova modifikacija izvedena je u sljedećem programu:

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

int main() {
    const int MaxBrojZnakova(1000);
    char znak, recenica[MaxBrojZnakova];
    cout << "Unesi rečenicu: ";
    int broj_znakova(0);
    while((znak = cin.get()) != '\n' && broj_znakova < MaxBrojZnakova)
        recenica[broj_znakova++] = znak;
    cout << "Rečenica izgovorena naopako glasi: ";
    for(int i = broj_znakova - 1; i >= 0; i--) cout << recenica[i];
    return 0;
}
```

O radu sa nizovima znakova više će biti govora u poglavljju koje govori o *stringovima* kao posebnoj vrsti nizova znakova.

Često je potrebno ispitati da li *svi elementi nekog niza posjeduju neko svojstvo*. Na primjer, u jednom od ranije navedenih primjera imali smo niz nazvan “temperature”. Ukoliko želimo utvrditi da li su sve temperature pozitivne, možemo koristiti sljedeću konstrukciju:

```
bool svi_su_pozitivni(true);
for(int mjesec = 0; mjesec < 12; mjesec++)
    if(temperature[mjesec] <= 0) svi_su_pozitivni = false;
```

Princip rada ove konstrukcije je sličan principu koji smo koristili kod ispitivanja da li je broj prost ili ne. Drugim riječima, *a priori* pretpostavljamo da svi elementi jesu pozitivni tako što logičku promjenljivu “svi_su_pozitivni” inicijaliziramo na vrijednost “**true**”, a zatim u petlji tražimo eventualan kontraprimjer za postavljenu pretpostavku. Ukoliko pri tome najđemo na kontraprimjer (tj. na negativan element ili nulu), ovu promjenljivu postavljamo na vrijednost “**false**”, čime signaliziramo da pretpostavka nije bila tačna (tj. da svi elementi nisu pozitivni). Ukoliko se cijela petlja izvršila a ni jedan element nije bio negativan (ili nula), tijelo naredbe “**if**” neće se izvršiti niti jedanput, tako da će vrijednost promjenljive “svi_su_pozitivni” ostati onakva kakva je bila na početku (tj. “**true**”), iz čega zaključujemo da je pretpostavka bila tačna, tj. da su zaista *svi elementi pozitivni*.

Veoma je važno uočiti da prethodni problem nije moguće problem riješiti postavljanjem logičke promjenljive “svi_su_pozitivni” na vrijednost “**false**”, a zatim njenim postavljanjem na “**true**” onog trenutka kada najđemo na pozitivnu vrijednost. Drugim riječima, sljedeća sekvenca naredbi *neće raditi ispravno*:

```
bool svi_su_pozitivni(false);
for(int mjesec = 0; mjesec < 12; mjesec++)
```

```
if(temperature[mjesec] > 0) svi_su_pozitivni = true;
```

Zaista, promjenljiva “svi_su_pozitivni” biće postavljena na “**true**” pri *prvom nailasku* na eventualnu pozitivnu vrijednost, i takva će ostati do kraja, bez obzira kakve su ostale vrijednosti. Drugim riječima, prikazana sekvenca naredbi zapravo ispituje da li je *makar jedan element* niza pozitivan. Stoga bi promjenljivoj “svi_su_pozitivni” trebalo promijeniti ime npr. u “makar_jedan_pozitivan” da bi njeno ime zaista odražavalo smisao prethodne sekvence:

```
bool makar_jedan_pozitivan(false);
for(int mjesec = 0; mjesec < 12; mjesec++)
    if(temperature[mjesec] > 0) makar_jedan_pozitivan = true;
```

Generalno, kada god treba ispitati da li *makar jedan element* niza posjeduje određeno svojstvo, potrebno je poći od suprotne pretpostavke (odnosno pretpostaviti da pretpostavka *nije tačna*), i tragati za eventualnim *primjerom* posjedovanja tog svojstva. Razlog za ovakav tretman leži u prostoj činjenici da je postojanje makar jednog elementa niza koji posjeduje određeno svojstvo jednostavno dokazati prostim nalaženjem primjera takvog elementa. S druge strane, tvrdnju da svi elementi niza posjeduju neko svojstvo nije moguće *dokazati* navođenjem primjera elementa koji posjeduje navedeno svojstvo, ali ju je moguće *opovrgnuti* nalaženjem makar jednog kontraprimjera, odnosno elementa koji *ne posjeduje* navedeno svojstvo.

Oba navedena primjera moguće je učiniti efikasnijim “nasilnim” prekidom petlje kada se uoči kontraprimjer, odnosno primjer za prepostavljeni tvrđenje. Na primjer, testiranje da li je makar jedan element niza “temperature” pozitivan efikasnije se može obaviti sljedećom sekvencom:

```
bool makar_jedan_pozitivan(false);
for(int mjesec = 0; mjesec < 12; mjesec++)
    if(temperature[mjesec] <= 0) {
        makar_jedan_pozitivan = true;
        break;
    }
```

Alternativno, naredba “**break**” se može izbjegići jednostavnom modifikacijom uvjeta petlje:

```
bool makar_jedan_pozitivan(false);
for(int mjesec = 0; mjesec < 12 && ! makar_jedan_pozitivan; mjesec++)
    if(temperature[mjesec] > 0) makar_jedan_pozitivan = true;
```

Uz ovaku modifikaciju čak je moguće potpuno izbjegići naredbu “**if**” i pisati konstrukciju poput sljedeće:

```
bool makar_jedan_pozitivan(false);
for(int mjesec = 0; mjesec < 12 && !makar_jedan_pozitivan; mjesec++)
    makar_jedan_pozitivan = temperature[mjesec] > 0;
```

Razmislite kako radi ova konstrukcija, kao i zbog čega slična konstrukcija ne radi ispravno bez izvršene modifikacije uvjeta petlje. Međutim, u svakom slučaju, rješenje sa naredbom “**break**” je vjerovatno neznatno efikasnije, jer se petlje čiji je uvjet jednostavniji brže izvršavaju (s obzirom da se uvjet mora testirati pri svakom prolasku kroz petlju).

Utvrđivanje karakterističnih osobina elemenata niza ilustriraćemo na još nekoliko primjera. Neka je potrebno ispitati da li su svi elementi niza “niz” (koji ima “broj_elemenata” elemenata) međusobno jednak. Nije teško utvrditi da je za tu svrhu dovoljno ispitivati samo elemente sa *susjednim indeksima*, i ustanoviti da tražena osobina nije zadovoljena ukoliko najđemo na makar jedan par susjednih elemenata koji nisu jednak (zbog tranzitivnosti relacije jednakosti, jednakost svih susjednih elemenata povlači i

jednakost svih elemenata). Stoga traženi problem rješava sljedeći programski isječak:

```
bool svi_su_istи(true);
for(int i = 0; i < broj_elemenata - 1; i++)
    if(niz[i] != niz[i+1]) {
        svi_su_исти = false;
        break;
}
```

Neka je sada potrebno ispitati da li su svi elementi istog niza međusobno različiti. Ova pretpostavka nije prosto negacija prethodne pretpostavke, iako bi se to moglo brzopleti zaključiti (naime, negacija pretpostavke "svi elementi su međusobno jednaki" glasi "svi elementi nisu međusobno jednaki", što zapravo znači "neki od elemenata su međusobno različiti" a ne "svi elementi su međusobno različiti"). Ovaj problem je teži od prethodnog, jer za utvrđivanje da li su svi elementi niza različiti nije dovoljno ispitivati samo parove susjednih elemenata, već je svaki element potrebno uporediti sa *svim elementima koji slijede nakon njega* (činjenica da se svi parovi susjednih elemenata međusobno razlikuju nije dovoljna da zaključimo da su svi elementi različiti, s obzirom da se čak i tada mogu pojaviti međusobno jednaki elementi koji nisu međusobno susjedni). Stoga nije dovoljna jedna "for" petlja, već je potrebno koristiti dvije ugniježdene "for" petlje. Zbog simetričnosti relacije jednakosti, nije neophodno upoređivati svaki element sa svim ostalim, već samo sa elementima koji slijede nakon njega, jer je njihova eventualna jednakost sa elementima koji mu prethode već utvrđena prilikom njihovog poređenja sa elementima koji im slijede. Ovo dovodi do sljedećeg rješenja postavljenog problema:

```
bool svi_su_razliciti(true);
for(int i = 0; i < broj_elemenata - 1; i++)
    for(int j = i + 1; i < broj_elemenata; j++)
        if(niz[i] == niz[j]) {
            svi_su_razliciti = false;
            break;
}
```

Sljedeći primjer ilustrira mnoge od do sada opisanih tehnika. Neka meteorološka služba svakog dana registrira najvišu i najnižu dnevnu temperaturu. Prikazani program učitava najvišu i najnižu temperaturu za svaki dan jednog mjeseca (u trajanju od 30 dana), i na izlazu daje sljedeće podatke:

- Maksimalnu temperaturu zabilježenu tokom tog mjeseca;
- Minimalnu temperaturu zabilježenu tokom tog mjeseca;
- Broj dana u toku mjeseca u kojima je zabilježena maksimalna temperatura;
- Prepostavku o godišnjem dobu na osnovu sljedećih pravila: prepostavlja se da je u pitanju zima ako je makar jedanput maksimalna dnevna temperatura bila ispod nule, a prepostavlja se da je u pitanju ljeto ako minimalne dnevne temperature niti jedanput nisu bile ispod 15 stepeni.

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

int main() {
    const int BrojDana(30);
    double max_temperature[BrojDana], min_temperature[BrojDana];
    for(int dan = 0; dan < BrojDana; dan++)
        do {
            cout << "Unesite minimalnu temperaturu u toku "
                << dan + 1 << "dana: ";
            cin >> min_temperature[dan];
```

```

cout << "Unesite maksimalnu temperaturu u toku "
    << dan + 1 << "dana: ";
cin >> max_temperature[dan];
if(min_temperature[dan] > max_temperature[dan]) {
    cout << "Neispravan unos!\n"
        "MAX mora biti veći od MIN!\n";
} while (min_temperature[dan] > max_temperature[dan]);
double max_temperatura(max_temperature[0]);
double min_temperatura(min_temperature[0]);
for(int dan = 1; dan < BrojDana; dan++) {
    if(max_temperature[dan] > max_temperatura)
        max_temperatura = max_temperature[dan];
    if(min_temperature[dan] < min_temperatura)
        min_temperatura = min_temperature[dan];
}
cout << "Maksimalna temperatura zabilježena u toku mjeseca je "
    << max_temperatura << ".\n"
    << "Minimalna temperatura zabilježena u toku mjeseca je "
    << min_temperatura << ".\n";
int br_dana_sa_max_temp(0);
for(int dan = 0; dan < BrojDana; dan++)
    if(max_temperature[dan] == max_temperatura)
        br_dana_sa_max_temp++;
cout << "Maksimalna temperatura zabilježena je "
    << br_dana_sa_max_temp << " puta u toku mjeseca.\n";
bool da_li_je_ljeto(true), da_li_je_zima(false);
for(int dan = 0; dan < BrojDana; dan++)
    if(max_temperature[dan] < 0) {
        da_li_je_zima = true;
        break;
    }
for(int dan = 0; dan < BrojDana; dan++)
    if(min_temperature[dan] < 15) {
        da_li_je_ljeto = false;
        break;
    }
if(da_li_je_ljeto && da_li_je_zima)
    cout << "Temperaturne oscilacije ovog mjeseca su veoma čudne.";
else if(da_li_je_ljeto)
    cout << "Jako je toplo, vjerovatno je ljeto.";
else if(da_li_je_zima)
    cout << "Vjerovatno je zima, temperature su bile ispod nule.";
return 0;
}

```

U ovom programu je naročito bitno razlikovati promjenljive nazvane "max_temperature" i "min_temperature" od promjenljivih sličnih imena "max_temperatura" i "min_temperatura" (napomenimo inače da davanje isuviše sličnih imena različitim promjenljivim nije osobito dobra programerska praksa). Promjenljive "max_temperature" i "min_temperature" predstavljaju nizove od 30 elemenata koje čuvaju maksimalne odnosno minimalne temperature u toku svakog od 30 dana, dok su promjenljive "max_temperatura" i "min_temperature" obične (skalarne) promjenljive koja čuvaju maksimalnu odnosno minimalnu temperaturu u toku *čitavog mjeseca*. Interesantno je pogledati kako je realizirano određivanje dana u kojima je postignuta maksimalna temperatura. Primijetimo da se konstrukcija

```
for(int dan = 0; dan < BrojDana; dan++)
```

```
if(max_temperature[dan] == max_temperatura)
    br_dana_sa_max_temp++;
```

mogla pisati i kraće, bez “**if**” naredbe, na sljedeći način, zahvaljujući automatskoj konverziji logičkih vrijednosti u cijelobrojnu vrijednost 0 ili 1 unutar aritmetičkih izraza:

```
for(int dan = 0; dan < BrojDana; dan++)
    br_dana_sa_max_temp += max_temperature[dan] == max_temperatura;
```

Prilikom testiranja prethodnog programa, možemo privremeno smanjiti vrijednost konstante “`BrojDana`” sa 30 na neku manju vrijednost, da izbjegnemo potrebu za unosom 60 podataka (po dvije temperature za svaki od 30 dana).

Kao što nije moguće kopirati jednu nizovnu promjenljivu u drugu pomoću operatora “`=`”, tako nije moguće ni ispitati da li su dvije nizovne promjenljive jednakе odnosno različite (tj. da li se sastoje od istih elemenata na istim pozicijama ili ne) pomoću operatora “`==`” odnosno “`!=`”. Međutim, neko bi mogao doći u zabludu da su ovakva poređenja moguća, s obzirom da će kompjuter prihvati sintaksno konstrukcije poput sljedeće (uz pretpostavku da su “`niz_1`” i “`niz_2`” dvije nizovne promjenljive istog tipa):

```
if(niz_1 == niz_2) cout << "Nizovi su isti!";
else cout << "Nizovi su različiti!";
```

Razlog zbog kojeg će ova konstrukcija biti prihvaćena je već spomenuta automatska konverzija imena nizovne promjenljive u pokazivač na prvi element niza, tako da će gore navedena konstrukcija uporediti pokazivače na prve elemente nizova “`niz_1`” i “`niz_2`”, a ne same nizove (preciznije, biće upoređene *adrese u memoriji* na kojima se ovi nizovi nalaze), što svakako neće dovesti do očekivanog rezultata. Ukoliko zaista želimo poređiti dva niza, poređenje moramo izvršiti *element po element*, koristeći sličnu strategiju kao pri ispitivanju da li svi elementi niza posjeduju ili ne posjeduju neko svojstvo. Na primjer, možemo izvesti konstrukciju poput sljedeće (ovdje je “`broj_elemenata`” broj elemenata ovih nizova, za koji prepostavljamo da je isti za oba niza):

```
bool isti_su(true);
for(int i = 0; i < broj_elemenata; i++)
    if(niz_1[i] != niz_2[i]) {
        isti_su = false; break;
    }
if(isti_su) cout << "Nizovi su isti!";
else cout << "Nizovi su različiti!";
```

Alternativno, možemo postupiti i ovako:

```
bool razliciti_su(false);
for(int i = 0; i < broj_elemenata; i++)
    if(niz_1[i] != niz_2[i]) {
        razliciti_su = true; break;
    }
if(razliciti_su) cout << "Nizovi su različiti!";
else cout << "Nizovi su isti!";
```

U ovim primjerima iskorištena je činjenica da se pretpostavka o jednakosti može lako oboriti navođenjem jednog kontraprimjera, a da se pretpostavka o različitosti može lako potvrditi navođenjem jednog primjera (obrnuto ne vrijedi, odnosno pretpostavka o jednakosti ne može se potvrditi navođenjem jednog primjera, niti se pretpostavka o različitosti može oboriti navođenjem jednog kontraprimjera).

Moguće je svim elementima niza odmah prilikom deklaracije (ali *samo tada*) dodijeliti *početne vrijednosti* navođenjem spiska vrijednosti unutar vitičastih zagrada (strogo rečeno, ovdje se ne radi o *dodjeli*, nego o *inicijalizaciji*). Na primjer, neka je potrebno definirati niz od 12 elemenata nazvan “mjeseci” koji će čuvati broj dana za svaki od 12 mjeseci u 2000. godini (oprez: ova godina je prestupna), i dodijeliti njegovim elementima vrijednosti. To sigurno možemo uraditi ovako:

```
int mjeseci[12];
mjeseci[0] = 31;
mjeseci[1] = 29;
mjeseci[2] = 31;
mjeseci[3] = 30;
mjeseci[4] = 31;
mjeseci[5] = 30;
mjeseci[6] = 31;
mjeseci[7] = 31;
mjeseci[8] = 30;
mjeseci[9] = 31;
mjeseci[10] = 30;
mjeseci[11] = 31;
```

Međutim, u jeziku C++ postoji znatno praktičniji način da postignemo isti efekat, pomoću sljedeće konstrukcije:

```
int mjeseci[12] = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Primijetimo da u ovom nizu mjesecu “januar” odgovara indeks 0 (a ne 1), mjesecu “februar” indeks 1, dok mjesecu “decembar” odgovara indeks 11.

U slučaju inicijaliziranih nizova, nije neophodno navoditi broj elemenata niza u uglastoj zagradi, jer je on određen brojem elemenata u listi navedenoj u vitičastim zagradama (koju najčešće nazivamo *inicijalizacijska lista*). Tako smo isti efekat mogli postići i naredbom:

```
int mjeseci[] = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Ukoliko koristimo ovakav način deklaracije niza, broj elemenata niza programski možemo odrediti koristeći trik sa “**sizeof**” operatorom, na primjer:

```
const int BrojMjeseci = sizeof mjeseci / sizeof mjeseci[0];
```

Opisani trik sa “**sizeof**” operatorom najviše se koristi upravo u ovakvim situacijama, kada eksperimentiramo sa inicijaliziranim nizovima kojima u fazi testiranja često dodajemo odnosno brišemo elemente. Na taj način, ne moramo nakon svake izmjene brojati koliko se u nizu zaista nalazi elemenata.

Inicijalizirane nizove, kod kojih se vrijednosti elemenata neće mijenjati tokom rada programa, dobro je označiti oznakom “**const**”, čime sprečavamo nehotične izmjene njihovog sadržaja (pokušaj promjene njihove vrijednosti doveće do prijave greške od strane kompjajlera). Tako smo prethodni niz mogli deklarirati sa “**const**” deklaracijom:

```
const int mjeseci[] = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Neinicijalizirani nizovi se ne mogu proglašiti konstantnim, iz razumljivih razloga (mada neki kompjajleri sintaksno dozvoljavaju takvu deklaraciju, ona je u suštini besmislena). Interesantno je da se, zbog nekih tehničkih razloga, elementi konstantnog niza ne tretiraju kao *prave konstante*, odnosno ne mogu se

upotrijebiti tamo gdje se očekuje prava konstanta ili konstantni izraz (u skladu sa ovim, nećemo koristiti veliko početno slovo za imenovanje konstantnih nizova). Tako, na primjer, nije dozvoljena deklaracija

```
int dani_januara[mjeseci[0]]
```

jer se izraz "mjeseci[0]" ne tretira kao prava konstanta, iako ima fiksnu i poznatu vrijednost (31).

Ukoliko upotrijebimo inicijalizacijsku listu koja sadrži manji broj elemenata od deklariranog broja elemenata, podrazumijeva se da su izostavljeni elementi jednaki nuli. Tako, na primjer, deklaracija

```
int a[8] = {2, 3, 5};
```

ima isti efekat kao i deklaracija

```
int a[8] = {2, 3, 5, 0, 0, 0, 0, 0};
```

Stoga je najbrži način da definiramo niz od 100 elemenata čiji su svi elementi na početku nula sljedeći:

```
int prazan_niz[100] = {};
```

Ovo je u svakom slučaju kraće od:

```
int prazan_niz[100];
for(int i = 0; i < 100; i++) prazan_niz[i] = 0;
```

Treba razlikovati situaciju kada se upotrijebi *prazna inicijalizacijska lista* od situacije u kojoj uopće nije upotrijebljena inicijalizacijska lista (u tom slučaju, početne vrijednosti elemenata niza su nedefinirane). Pri upotrebi prazne inicijalizacijske liste, broj elemenata niza se *mora navesti* (iz očiglednih razloga). Neki kompjajleri ne dozvoljavaju upotrebu prazne inicijalizacijske liste (mada je standard dozvoljava), već moramo navesti barem jedan element u inicijalizacijskoj listi, na primjer:

```
int prazan_niz[100] = {0};
```

Treba obratiti pažnju na čestu zabludu: deklaracija

```
int prazan_niz[100] = {5};
```

kreira niz od 100 elemenata od kojih je samo prvi element inicijaliziran na vrijednost 5 (a ostali na 0), a ne niz u kojem su svi elementi inicijalizirani na vrijednost 5, kao što bi neko mogao pomisliti.

Kao ilustraciju primjene inicijaliziranih nizova, prikazaćemo program koji prvo traži od korisnika da unese redni broj nekog mjeseca u opsegu od 1 do 12 (1 – januar, 2 – februar, itd.), zatim pita da li je godina prestupna, i na kraju ispisuje koliko dana ima uneseni mjesec. Radi kratkoće, ispuštena je provjera ispravnosti unesenih podataka:

```
#include <iostream>
using namespace std;
int main() {
    int mjeseci[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int mjesec;
    char prestupna;
    cout << "Unesi mjesec (1-12, 1=januar, 2=februar, itd.): ";
    cin >> mjesec;
    cout << "Da li je godina prestupna? ";
    cin >> prestupna;
```

```

// Oprez: februar se čuva pod indeksom 1 a ne 2!
if(prestupna == 'd' || prestupna == 'D') mjeseci[1] = 29;
cout << "Taj mjesec ima " << mjeseci[mjesec - 1] << "dana.";
return 0;
}

```

Uočite razliku između promjenljivih "mjesec" i "mjeseci". U ovom primjeru niz "mjeseci" nije deklariran kao konstantan, jer se mijenja vrijednost elementa "mjeseci[1]" (koji odgovara februaru). Također, ukoliko ste shvatili dosad izložene osobine nizova i njihovih indeksa, neće Vam biti čudno zašto je u posljednjoj naredbi unutar uglaste zagrade napisano "mjesec - 1" umjesto "mjesec".

Naredni primjer je nešto složeniji, ali izuzetno značajan. U njemu se od korisnika traži da unese neki datum u 2000. godini (godini kada je ovaj materijal prvi put napisan) u obliku DAN MJESEC (npr. 25 11), nakon čega program utvrđuje i ispisuje koji je to dan u sedmici. Rad programa se zasniva na činjenici da je 1. januar 2000. godine bila subota. Ako odredimo broj dana koji je protekao od tog datuma do unesenog datuma, tada će ostatak dijeljenja sa 7 proteklog broja dana odrediti koji je to dan u sedmici. Naime, ako je broj proteklih dana djeljiv sa 7, između ta dva datuma protekao je cijeli broj sedmica, pa i drugi datum mora takođe biti subota. Slično zaključujemo da ako je ostatak dijeljenja 1, drugi datum pada u nedjelju, itd. Ostaje još da riješimo kako da odredimo broj proteklih dana između 1. januara 2000. godine i unesenog datuma. Neka je uneseni datum bio npr. 14. maj. Očigledno je od 1. januara 2000. godine do 14. maja 2000. godine proteklo onoliko dana koliko ima u prva četiri mjeseca (januar, februar, mart i april) plus još 13 dana. U općem slučaju, od 1. januara do datuma DAN MJESEC proteklo je onoliko dana koliko imaju svi mjeseci zajedno do mjeseca MJESEC - 1 plus još DAN - 1 dana. Na osnovu ovoga, dobijamo sljedeći program (razmislite šta se dešava sa "for" petljom ako je uneseni mjesec januar):

```

#include <iostream>
using namespace std;

int main() {
    const int mjeseci[12] = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int dan, mjesec;
    bool pogresan_datum; // "true" ako je unesen neispravan datum
    do {
        cout << "Unesi datum u obliku DAN MJESEC: ";
        cin >> dan >> mjesec;
        pogresan_datum = !cin || mjesec < 1 || mjesec > 12
            || dan < 1 || dan > mjeseci[mjesec];
        if(pogresan_datum) cout << "Unijeli ste besmislen datum!\n";
        if(!cin) cin.clear();
        cin.ignore(10000, '\n');
    } while(pogresan_datum);
    int broj_proteklih_dana = dan - 1;
    for(int i = 0; i < mjesec - 1; i++)
        broj_proteklih_dana += mjeseci[i];
    switch(broj_proteklih_dana % 7) {
        case 0: cout << "Subota\n"; break;
        case 1: cout << "Nedjelja\n"; break;
        case 2: cout << "Ponedjeljak\n"; break;
        case 3: cout << "Utorak\n"; break;
        case 4: cout << "Srijeda\n"; break;
        case 5: cout << "Četvrtak\n"; break;
        case 6: cout << "Petak\n";
    }
}

```

```
    return 0;  
}
```

Važno je napomenuti da se višestruko pridruživanje liste vrijednosti elementima niza može obaviti samo prilikom *deklaracije niza*, odnosno u postupku *inicijalizacije*. Naknadna dodjeljivanja liste vrijednosti elementima niza nisu moguća. Na primjer, sljedeća konstrukcija *nije dozvoljena*:

```
int mjeseci[12];  
mjeseci = {31, 29, 31, 30, 31, 30, 31, 30, 31, 30, 31};
```

Treba primjetiti da *umetanje* novog elementa u niz između dva postojeća elementa nije direktno izvodivo bez prethodnog pomjeranja svih elemenata niza koji slijede nakon pozicije na koju želimo umetnuti element za jedno mjesto naviše. Na primjer, ukoliko u niz “niz” koji ima “broj_elemenata” elemenata želimo na mjesto sa indeksom “pozicija” umetnuti novu vrijednost “vrijednost” (pri čemu želimo element koji se već nalazio na tom mjestu kao i sve elemente koji slijede iza njega “istisnuti” za jedno mjesto naviše), možemo upotrijebiti sljedeću konstrukciju:

```
for(int i = broj_elemenata - 1; i > pozicija; i--)  
    niz[i] = niz[i - 1];  
niz[pozicija] = vrijednost;
```

Veoma je važno da sami shvatite zbog čega je u ovom primjeru bilo neophodno da petlja ide *unazad*, odnosno od većih vrijednosti indeksa ka nižim. Naravno, u ovom primjeru element koji se nalazio na posljednjem mjestu u nizu postaje *izgubljen*, s obzirom da se veličina niza ne može naknadno povećavati dodavanjem novih elemenata. S druge strane, *izbacivanje* elemenata niza zahtijeva pomjeranje svih elemenata niza koji mu slijede za jedno mjesto naniže. Na primjer, izbacivanje elementa sa indeksom “pozicija” možemo izvesti sljedećom konstrukcijom (razmislite zašto u ovom slučaju petlja mora ići *unaprijed*, odnosno od manjih vrijednosti indeksa ka višim):

```
for(int i = pozicija; i < broj_elemenata; i++)  
    niz[i] = niz[i + 1];
```

Pored *jednodimenzionalnih nizova* koje smo do sada opisivali, postoje i *višedimenzionalni nizovi*, čijim se elementima pristupa sa *više indeksa*. *Dvodimenzionalni nizovi* se obično nazivaju *matrice*. Na primjer, neka imamo sljedeću deklaraciju:

```
int matrica[5][3];
```

Ovim je definirana promjenljiva “matrica” koja predstavlja dvodimenzionalni niz (matricu) formata 5×3 , odnosno matricu sa 5 redova i 3 kolone. Elementima matrice “matrica” pristupa se preko dva indeksa, npr. izrazom poput “matrica[2][3]”. Na sličan način se deklariraju i koriste nizovi sa *više od dvije dimenzije*. Kako višedimenzionalni nizovi posjeduju obilje specifičnosti, njima će kasnije biti posvećeno posebno poglavlje.

Iz dosadašnjih izlaganja može se vidjeti da nizovi, bez obzira na svoju korisnost i fleksibilnost, posjeduju neke primjetne nedostatke. Svi ovi nedostaci su uglavnom posljedica činjenice da je rad sa nizovima naslijeden iz jezika C, koji je jezik *nižeg nivoa* od jezika C++ (u smislu da mu je filozofija razmišljanja više orijentirana ka načinu rada samog računara). Zbog toga, rad sa klasičnim nizovima, u izvjesnom smislu nije “u duhu” jezika C++. Da bi se ovi problemi izbjegli, u standard ISO C++98 je uveden novi tip podataka, nazvan “vector”, definiran u istoimenoj biblioteci (za korištenje ovog tipa podataka moramo uključiti u program zaglavje biblioteke “vector”). Ovaj tip podataka (zovimo ga prosto *vektor*) zadržava većinu svojstava koji posjeduju standardni nizovi, ali ispravlja neke njihove nedostatke. Promjenljive tipa “vector” deklariraju se na sljedeći način:

```
vector<tip_elemenata> ime_promjenljive(broj_elemenata);
```

Na primjer:

```
vector<double> padavine(12);
```

Primijetimo da "vector" nije ključna riječ, s obzirom da tip vektor nije tip ugrađen u sam jezik C++, već korisnički definirani tip (slično kao i npr. tip "complex"), definiran u istoimenoj biblioteci "vector". Ovako deklarirane promjenljive mogu se u gotovo svim slučajevima koristiti poput običnih nizovnih promjenljivih (za pristup individualnim elementima vektora, kao i kod običnih nizova, koriste se uglaste zgrade). S druge strane, rad sa ovakvim promjenljivim je fleksibilniji nego sa običnim nizovima, iako postoje i neki njihovi nedostaci u odnosu na klasične nizove. Najbitnije razlike između običnih nizovnih promjenljivih i promjenljivih tipa "vector" ogledaju se u sljedećim karakteristikama:

- Broj elemenata pri deklaraciji vektora navodi se u *običnim*, a ne u uglastim zgradama, čime je istaknuta razlika između zadavanja broja elemenata i navođenja indeksa za pristup elementima (u kasnijim poglavljima ćemo vidjeti da bismo upotreboom uglastih zagrada pri deklaraciji vektora zapravo deklarirali *niz čiji su elementi vektori*, što nije ono što u ovom slučaju želimo). Pored toga, broj elemenata može biti *proizvoljan izraz*, a ne samo konstantna vrijednost ili konstantan izraz. Drugim riječima, broj elemenata vektora ne mora biti apriori poznat.
- Elementi vektora se *automatski inicijaliziraju na nulu*, za razliku od običnih nizova, kod kojih elementi imaju nedefinirane vrijednosti, ukoliko se eksplicitno ne navede inicijalizaciona lista. S druge strane, kod vektora nije moguće zadati inicijalizacionu listu (mada postoje neki drugi načini za inicijalizaciju vektora, o kojima na ovom mjestu nećemo govoriti).
- Postoji mogućnost naknadnog mijenjanja broja elemenata u vektoru primjenom operacije "resize". Na primjer, ukoliko se u toku rada programa utvrdi da vektor "padavine" treba da sadrži 18 elemenata umjesto 12, moguće je izvršiti naredbu

```
padavine.resize(18);
```

nakon koje će broj elemenata vektora "padavine" biti povećan sa 12 na 18. Postojeći elementi zadržavaju svoje vrijednosti.

- Imena promjenljivih tipa vektor upotrijebljena sama za sebe (tj. bez indeksa) ne konvertiraju se automatski u pokazivače, kao što je to slučaj kod običnih nizova. Mada nekome to može djelovati kao prednost a ne mana vektora u odnosu na obične nizove, to kao posljedicu ima činjenicu da se tzv. *pokazivačka aritmetika*, koju ćemo upoznati u kasnijim poglavljima, ne može neposredno primjenjivati na vektore.
- Promjenljive tipa vektor mogu se dodjeljivati jedna drugoj pomoću operatora "=" (pri čemu se kopiraju svi elementi, odnosno vektor kao cjelina), kao i testirati na jednakost odnosno različitost pomoću operatora "==" i "!=" što, kao što smo već vidjeli, nije moguće sa običnim nizovima.
- Pomoću operacije "push_back" moguće je jednostavno dodavati nove elemente na kraj vektora, pri čemu se broj elemenata vektora pri takvom dodavanju povećava za jedan. Na primjer, naredba

```
padavine.push_back(27.3);
```

povećava broj elemenata vektora "padavine" za 1, i novododanom elementu (koji se nalazi na kraju vektora) dodjeljuje vrijednost "27.3". Sasvim je moguće deklarirati *prazan vektor*, odnosno vektor koji ne sadrži niti jedan element (u tom slučaju, u deklaraciji vektora, broj elemenata vektora zajedno sa pripadnim zgradama se izostavlja), a zatim operacijom "push_back" dodati onoliko elemenata u vektor koliko nam je potrebno. Ova strategija je naročito praktična u slučaju kada ne znamo unaprijed

koliko će vektor imati elemenata (npr. kada elemente vektora unosimo sa tastature, pri čemu se unos vrši sve dok na neki način ne signaliziramo da smo završili sa unosom).

- Postoje operacije, koje nećemo ovdje opisivati, pomoću kojih je moguće *ubaciti* novi element u vektor (između dva postojeća elementa) ili *ukloniti* element iz vektora, bez potrebe za korištenjem “**for**” petlje.
- Prilikom prenošenja običnih nizova kao parametara u funkcije, o čemu ćemo govoriti u kasnijim poglavljima, gotovo uvijek je potrebno prenijeti *broj elemenata niza* kao poseban parametar. Ovo nije potrebno pri prenosu vektora kao parametara u funkcije. Također, funkcija kao svoj rezultat ne može vratiti običan niz, ali može vektor (o čemu ćemo također govoriti kasnije).
- Operator “**sizeof**” primijenjen na promjenljive tipa vektor daje rezultat koji nije u skladu sa intuicijom. Zbog toga, trik za određivanje broja elemenata niza zasnovan na primjeni ovog operatorka *ne daje ispravan rezultat* ukoliko se primijeni na promjenljive tipa vektor. Umjesto toga, za određivanje broja elemenata vektora treba koristiti operaciju “**size**”. Na primjer:

```
cout << "Vektor \"padavine\" ima " << padavine.size() << " elemenata.";
```

Kao ilustraciju dopunskih mogućnosti tipa “vector” u odnosu na obične nizove, navedimo ponovo program koji obrađuje rezultate ispita, ali koji koristi promjenljive tipa “vector” umjesto običnih nizova. U ovom slučaju nemamo ograničenje na maksimalan broj kandidata, s obzirom da broj elemenata vektora možemo zadati na osnovu podatka o broju kandidata unesenom sa tastature:

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

using namespace std;

int main() {
    const PragZaProlaz(55);
    int broj_kandidata;
    cout << "Koliko kandidata je pristupilo ispitu? ";
    cin >> broj_kandidata;
    vector<int> poeni(broj_kandidata);
    for(int i = 0; i < broj_kandidata; i++) {
        cout << "Unesi broj poena za " << i + 1 << "kandidata: ";
        cin >> poeni[i];
    }
    cout << endl << "Kandidat Status\n";
    for(int i = 0; i < broj_kandidata; i++) {
        cout << setw(7) << i + 1 << ".";
        if(poeni[i] >= PragZaProlaz) cout << "POLOŽIO\n";
        else cout << "PAO\n";
    }
    return 0;
}
```

Također, činjenica da možemo deklarirati prazan vektor, a zatim operacijom “*push_back*” dodavati nove elemente na njegov kraj, često olakšava izvodenje mnogih operacija. Na primjer, program koji ispisuje unijetu rečenicu naopako postaje mnogo jednostavniji i fleksibilniji ukoliko umjesto običnih nizova koristimo vektor (čiji su elementi tipa “**char**”):

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

using namespace std;
```

```
int main() {
    char znak;
    vector<char> recenica;
    cout << "Unesi rečenicu: ";
    while((znak = cin.get()) != '\n') recenica.push_back(znak);
    cout << "Rečenica izgovorena naopako glasi: ";
    for(int i = recenica.size() - 1; i >= 0; i--) cout << recenica[i];
    return 0;
}
```

Zbog njihovih očiglednih prednosti, danas se savjetuje upotreba vektora umjesto običnih nizova. Osnovni razlog zbog kojeg ćemo mi u nastavku ipak pretežno koristiti obične nizove leži u činjenici da obični nizovi predstavljaju fundamentalni tip koji čini srž samog jezika C++, dok su vektori izvedeni tip koji je *napravljen* koristeći fundamentalne tipove jezika C++. Kako je nama, između ostalog, cilj i da naučimo praviti *vlastite korisnički definirane tipove*, to nije moguće ukoliko se prethodno temeljito ne upoznamo sa fundamentalnim tipovima podataka (u koje spadaju nizovi) i ukoliko se ne naviknemo na njihove osobine (svidale nam se one ili ne), jer oni predstavljaju temelj na kojem se kasnije grade izvedeni korisnički definirani tipovi podataka.

13. Korisnički imenovani i pobrojani tipovi

Do sada smo se upoznali sa velikim brojem osnovnih tipova podataka, kao što su “`int`”, “`float`”, “`char`”, “`bool`” itd. kao i nizovnim tipovima izvedenim iz ovih osnovnih tipova. Međutim, jezik C++ dozvoljava i samostalno definiranje i kreiranje *novih tipova podataka*, koji se ponašaju onako kako njihov kreator odredi. O kreiranju novih tipova podataka detaljno ćemo govoriti kasnije. U ovom poglavlju ćemo govoriti o *korisnički imenovanim* (engl. *user named*) kao i *pobrojanim* (engl. *enumerated*) tipovima podataka.

Korisnički imenovani tipovi podataka nastaju kao posljedica mogućnosti jezika C++ koja omogućava da pomoću naredbe (deklaracije) “`typedef`” damo *alternativna imena* (engl. *alias names*) postojećim tipovima, ili da damo *vlastita imena* raznim izvedenim tipovima poput nizovnih tipova, koji su manje ili više bezimene prirode (u smislu nepostojanja vlastitog posebnog imena). Na primjer, ukoliko izvršimo deklaracije

```
typedef int CijeliBroj;
typedef long int VelikiCijeliBroj;
typedef unsigned int PrirodanBroj;
```

definirali smo nove *tipove* nazvane “`CijeliBroj`”, “`VelikiCijeliBroj`” i “`PrirodanBroj`” koji nisu ništa drugo nego drugi nazivi za već postojeće tipove “`int`”, “`long int`” i “`unsigned int`”. Tako, ako kasnije izvršimo deklaracije poput

```
CijeliBroj a;
VelikiCijeliBroj b;
PrirodanBroj c;
```

definiraćemo tri promjenljive “`a`”, “`b`” i “`c`” sa tipovima “`int`”, “`long`” i “`unsigned int`” respektivno. Obratite pažnju da deklaracija “`typedef`” ne kreira *promjenljive*, nego samo (apstraktne) *tipove*, tako da, uz gore navedene deklaracije, nema smisla napisati nešto poput

```
CijeliBroj = 5;
```

niti:

```
cin >> CijeliBroj;
```

s obzirom da “`CijeliBroj`” nije *promjenljiva*, već *tip* (koji možemo upotrijebiti za deklaraciju *promjenljivih*). Imena tipova koje uvodimo deklaracijom “`typedef`” predstavljaju *identifikatore*, tako da pri njihovom imenovanju moramo poštovati pravila koja važe za sve ostale identifikatore (posebno, ova imena moraju se sastojati od *jedne riječi*, odnosno razmaci u imenu nisu dozvoljeni). U navedenim primjerima poštivali smo jednu konvenciju koje se mnogi C++ programeri pridržavaju: korisnički imenovanim i definiranim *tipovima* podataka daju se imena koja počinju *velikim početnim slovom*, slično kao i imena *pravih konstanti*. Upotreba znaka “_” se također *izbjegava* u imenima tipova. Poštovanje ove konvencije olakšava razlikovanje namjene pojedinih identifikatora u programu.

Jedna mogućnost primjene naredbe “`typedef`” u ovakovom obliku je uvođenje sinonima za imena tipova koji su inače nezgrapni. Na primjer, neka nam je često potrebna deklaracija *promjenljivih tipa* “`unsigned long int`”. U tom smislu ima smisla uvesti deklaraciju

```
typedef unsigned long int ULint;
```

nakon čega možemo koristiti ime “ULint” kao sinonim za tip “**unsigned long int**”. Pored uštede u pisanju, uvođenjem imena koje je jedna riječ omogućavamo primjenu funkcijeske notacije pri pretvorbi tipova. Na primjer, dok nije moguće pisati nešto poput

```
cijeli = unsigned long int(realni);
```

već samo

```
cijeli = (unsigned long int) realni;
```

to su, s druge strane, obje naredbe koje slijede posve legalne (i ravnopravne):

```
cijeli = ULint(realni);
cijeli = (ULint)realni;
```

Tipična situacija u kojoj ima smisla definirati alternativno ime za inače rogočatan naziv tipa je pri upotrebi kompleksnih tipova. Na primjer, sasvim je smisleno uvesti deklaraciju

```
typedef complex<double> Cplx;
```

nakon koje možemo pisati konstrukcije poput

```
Cplx a(3, 2), b;
b = 2 + b * Cplx(4, -1);
```

umjesto mnogo rogočatnijih konstrukcija poput

```
complex<double> a(3, 2), b;
b = 2 + b * complex<double>(4, -1);
```

Druga primjena uvođenja alternativnih imena postojićim tipovima je pravljenje jasnije specifikacije kakva je priroda vrijednosti pohranjenih u pojedinim promjenljivim, što može učiniti program jasnijim, pogotovo ukoliko se radi o velikim programima. Pretpostavimo, na primjer, da su nam potrebne promjenljive “m1”, “m2” i “m3” koje pamte informaciju o masama tri tijela, kao i promjenljive “v1”, “v2” i “v3” koje pamte informaciju o njihovim brzinama. Naravno, svih ovih šest promjenljivih možemo principijelno deklarirati kao promjenljive tipa “**double**”. Međutim, sasvim je razumno izvršiti deklaracije tipova

```
typedef double Masa;
typedef double Brzina;
```

a nakon toga, ove promjenljive deklarirati na sljedeći način:

```
Masa m1, m2, m3;
Brzina v1, v2, v3;
```

Na ovaj način, mnogo je jasnije šta ove promjenljive predstavljaju. Nažalost, tipovi “Masa” i “Brzina” ne predstavljaju suštinski različite tipove od tipa “**double**”, nego predstavljaju samo njihova alternativna imena, tako da se kompjajler neće pobuniti ukoliko npr. pokušamo izvršiti sabiranje promjenljivih “m1” i “v2”, koje je smisleno samo ako ga apstraktno posmatramo (odnosno, sasvim je smisleno sabrati dva realna broja), ali je besmisleno u konkretnom kontekstu (odnosno, besmisleno je sabirati mase i brzine). Stoga jezik C++ ne posjeduje *potpuno strogu tipizaciju*, kakvu posjeduje npr. jezik ADA, u kojem su su ovakve manipulacije poput sabiranja masa i brzina zabranjene (osim izričitim navođenjem zahtjeva za pretvorbom tipa), mada na apstraktnom nivou obje veličine predstavljaju samo realne brojeve.

Naredba “**typedef**” omogućava i imenovanje nekih manje ili više *bezimenih* (engl. *anonymous*) tipova, kao što su nizovni tipovi. Na primjer, pogledajmo sljedeću deklaraciju nizovne promjenljive “a”:

```
double a[3];
```

Zapitajmo se *kojeg je tipa* promjenljiva “a”. Ona sigurno nije tipa “**double**”, nego je tipa “*niz od tri elementa od kojih je svaki tipa “double”*” (ta promjenljiva može služiti npr. za čuvanje koordinata nekog vektora u prostoru, koji je potpuno određen sa tri realna broja). Ovakav složeni tip nema neko posebno ime, nego se predstavlja *opisno*, kao u upravo navedenoj rečenici. Često se uvodi i skraćeno neformalno zapisivanje, u kojem se kaže da je promjenljiva “a” tipa “**double [3]**”, mada treba imati na umu da C++ ne dozvoljava deklaraciju poput

```
double [3] a;
```

Međutim, naredba “**typedef**” omogućava da ovom polu-anonimnom tipu “**double [3]**” damo *konkretno ime*. Ukoliko će se ovaj tip zaista koristiti za čuvanje koordinata vektora u prostoru, ima smisla uvesti deklaraciju poput

```
typedef double Vektor[3];
```

Ovom deklaracijom smo uveli novi tip “Vektor” koji predstavlja tip *niza od tri realna elementa*, odnosno sinonim za tip koji smo neformalno nazvali “**double [3]**” (nemojte brkati ovaj novouvedeni tip sa standardnim tipom “*vector*” definiranim u istoimenoj biblioteci). Primjetimo da se i u ovoj deklaraciji, slično kao i u deklaraciji nizovnih promjenljivih, dimenzija vezuje za *ime onoga što deklariramo*, a ne uz ime tipa elemenata. Drugim riječima, sljedeća deklaracija je neispravna:

```
typedef double[3] Vektor;
```

Treba paziti da smo “**typedef**” deklaracijom samo definirali *novi tip* “Vektor”, ne kreirajući pri tome *niti jedan primjerak* promjenljive tog tipa (to moramo sami *eksplicitno* uraditi). Prema tome, nakon ovakve deklaracije nema smisla pisati nešto poput:

```
Vektor[2] = 17.6;
```

s obzirom da “Vektor” nije ime *promjenljive*, nego *tipa*. Stoga, da bi ovakva definicija tipa dobila konkretni smisao, ona se mora kasnije upotrijebiti za deklariranje promjenljivih tog tipa. Tako, na primjer, postaju smislene sljedeće deklaracije:

```
Vektor a, b, c;
```

Smisao ove deklaracije isti je kao da smo napisali

```
double a[3], b[3], c[3];
```

tako da, nakon takve deklaracije, iskazi poput

```
a[2] = 17.6;
cin >> b[0];
```

imaju smisla.

Treba voditi računa da sam pojam “tip” predstavlja *apstraktan pojam*, dok se pojam “promjenljiva”

uvijek odnosi na konkretni objekat. Razliku između pojma *tipa* i pojma *promjenljive* možemo ilustrirati na primjeru koji nije vezan za programiranje. Treba razlikovati pojam "automobil" kao *apstraktni pojam* od *konkretnog automobila*. Kada damo definiciju poput "automobil je vrsta putničkog vozila", mi smo samo definirali *šta je to automobil*, dok ta definicija sama po sebi *ne povlači postojanje* niti jednog konkretnog primjerka automobila. Tek ako kažemo nešto poput "moj automobil" ili "automobil Čarlija Čaplina", govorimo o konkretnom primjerku nečega što se generalno zove "automobil". Na isti način, naredbom poput

```
typedef double Vektor[3];
```

mi smo samo rekli sljedeće: "vektor je niz od 3 realna broja". Dakle, samo smo definirali *šta je to vektor kao pojam*, a nismo definirali *niti jedan konkretni vektor*. Tek ako kasnije kažemo

```
Vektor a, b, c;
```

tada smo rekli "neka su "a", "b" i "c" vektori", čime smo kreirali tri konkretna primjerka pojma "Vektor".

Za razliku od korisnički imenovanih tipova uvedenih naredbom "**typedef**", koji samo na drugi način izražavaju tipove koji već postoje, *pobrojani* (engl. *enumerated*) tipovi predstavljaju prvi korak ka *korisnički definiranim tipovima*. Pobrojani tipovi opisuju *konačan skup vrijednosti koje su imenovane i uređene* (tj. stavljene u poredak) od strane programera. Definiramo ih pomoću naredbe "**enum**", iza koje slijedi *ime tipa koji definiramo i popis mogućih vrijednosti tog tipa unutar vitičastih zagrada* (kao moguće vrijednosti mogu se koristiti proizvoljni identifikatori koji nisu već iskorišteni za neku drugu svrhu). Na primjer, pomoću deklaracija

```
enum Dani {Ponedjeljak, Utorka, Srijeda, Cetvrtak, Petak,  
Subota, Nedjelja};  
enum Rezultat {Poraz, Nerijeseno, Pobjeda};
```

definiramo dva nova *tipa nazvana* "Dan" i "Rezultat". Promjenljive pobrojanog tipa možemo deklarirati na uobičajeni način, na primjer:

```
Rezultat danasnji_rezultat;  
Dani danas, sutra;
```

Obratite pažnju na tačka-zarez iza završne vitičaste zagrade u deklaraciji pobrojanog tipa, s obzirom da je u jeziku C++ prilična rijetkost da se tačka-zarez stavlja neposredno iza zatvorene vitičaste zagrade. Razlog za ovu prividnu anomaliju leži u činjenici da sintaksa jezika C++ dozvoljava da se odmah nakon deklaracije pobrojanog tipa definiraju i konkretne promjenljive tog tipa, navođenjem njihovih imena iza zatvorene vitičaste zagrade (sve do završnog tačka-zareza). Na primjer, prethodne deklaracije tipova "Dan" i "Rezultat" i promjenljivih "danasni_rezultat", "danasy" i "sutra" mogu se pisati skupa, na sljedeći način:

```
enum Dani {Ponedjeljak, Utorka, Srijeda, Cetvrtak, Petak,  
Subota, Nedjelja} danas, sutra;  
enum Rezultat {Poraz, Nerijeseno, Pobjeda} danasnji_rezultat;
```

Stoga je tačka-zarez iza zatvorene vitičaste zagrade prosto signalizator da ne želimo odmah deklarirati promjenljive odgovarajućeg pobrojanog tipa, nego da ćemo to obaviti naknadno. Treba napomenuti da se deklaracija promjenljivih pobrojanog tipa istovremeno sa deklaracijom tipa, mada je dozvoljena, smatra *lošim stilom*.

Promjenljive ovako definiranog tipa "Dan" mogu uzimati samo vrijednosti "Ponedjeljak", "Utorak", "Srijeda", "Cetvrtak", "Petak", "Subota" i "Nedjelja" i ništa drugo. Ove vrijednosti nazivamo *pobrojane konstante tipa* "Dani". Slično, promjenljive tipa "Rezultat" mogu uzimati samo vrijednosti "Poraz", "Nerijeseno" i "Pobjeda" (pobrojane konstante tipa "Rezultat"). Slijede primjeri legalnih dodjeljivanja sa pobrojanim tipovima:

```
dan = Srijeda;  
danasnji_rezultat = Pobjeda;
```

Promjenljive pobrojanog tipa u jeziku C++ posjeduju mnoga svojstva pravih korisnički definiranih tipova, o kojima ćemo govoriti kasnije. Tako je, recimo, moguće definirati *kako će djelovati pojedini operatori* kada se primijene na promjenljive pobrojanog tipa, o čemu ćemo detaljno govoriti u poglavlju o preklapanju (preopterećivanju) operatora. Na primjer, moguće je definirati šta će se tačno desiti ukoliko pokušamo sabrati dvije promjenljive ili pobrojane konstante tipa "Dani", ili ukoliko pokušamo ispisati promjenljivu ili pobrojanu konstantu tipa "Dani". Međutim, ukoliko ne odredimo drugačije, svi izrazi u kojima se upotrijebe promjenljive pobrojanog tipa koji *ne sadrže bočne efekte koji bi mijenjali vrijednosti tih promjenljivih* (poput primjene operatora "++" itd.), izvode se tako da se promjenljiva (ili pobrojana konstanta) pobrojanog tipa *automatski konvertira u cjelobrojnu vrijednost* koja odgovara *rednom broju odgovarajuće pobrojane konstante u definiciji pobrojanog tipa* (pri čemu numeracija počinje od nule). Tako se, na primjer, pobrojana konstanta "Ponedjeljak" konvertira u cjelobrojnu vrijednost "0" (isto vrijedi za pobrojanu konstantu "Poraz"), pobrojana konstanta "Utorak" (ili pobrojana konstanta "Nerijeseno") u cjelobrojnu vrijednost "1", itd. Stoga će rezultat izraza

```
5 * Srijeda - 4
```

biti cjelobrojna vrijednost "6", s obzirom da se pobrojana konstanta "Srijeda" konvertira u cjelobrojnu vrijednost "2" (preciznije, ovo vrijedi samo ukoliko postupkom preklapanja operatora nije dat drugi smisao operatoru "*" primijenjenom na slučaj kada je drugi operand tipa "Dani"). Ista će biti i vrijednost izraza

```
5 * danas - 4
```

s obzirom da je promjenljivoj "dan" dodijeljena pobrojana konstanta "Srijeda". Iz istog razloga će naredba

```
cout << danas;
```

ispisati na ekran vrijednost "2", a ne tekst "Srijeda", kako bi neko mogao brzopleti pomisliti (osim ukoliko postupkom preklapanja operatora operatoru "<<" nije dat drugačiji smisao). Također, zahvaljujući automatskoj konverziji u cjelobrojne vrijednosti, između pobrojanih konstanti definiran je i *poredak*, na osnovu njihovog redoslijeda u popisu prilikom deklaracije. Na primjer, vrijedi

```
Srijeda < Petak  
Pobjeda > Nerijeseno
```

Treba napomenuti da se pobrojane konstante tretiraju kao *prave konstante*, u smislu da se mogu upotrijebiti bilo gdje gdje se očekuje prava konstanta ili konstantni izraz. Na primjer, deklaracija poput

```
int niz[Petak];
```

sasvim je legalna, i ekvivalentna je deklaraciji

```
int niz[4];
```

Razumije se da se, zbog automatske konverzije u cjelobrojnu vrijednost, pobrojane konstante i promjenljive pobrojanog tipa mogu koristiti i kao indeksi nizova.

Moguće je deklarirati i promjenljive *bezimenog pobrojanog tipa* (engl. *anonymous enumerations*). Na primjer, deklaracijom

```
enum {Poraz, Nerijeseno, Pobjeda} danasnji_rezultat;
```

deklariramo promjenljivu "danasnji_rezultat" koja je sigurno pobrojanog tipa, i koja sigurno može uzimati samo vrijednosti "Poraz", "Nerijeseno" i "Pobjeda", ali tom pobrojanom tipu nije dodijeljeno nikakvo ime koje bi se kasnije moglo iskoristiti za definiranje novih promjenljivih istog tipa.

Prilikom deklariranja pobrojanih tipova moguće je zadati *u koju će se cjelobrojnu vrijednost* konvertirati odgovarajuća pobrojana konstanta, prostim navođenjem znaka jednakosti i odgovarajuće vrijednosti iza imena pripadne pobrojane konstante. Na primjer, ukoliko izvršimo deklaraciju

```
enum Rezultat {Poraz = 3, Nerijeseno, Pobjeda = 7};
```

tada će se pobrojana konstante "Poraz", "Nerijeseno" i "Pobjeda" konvertirati respektivno u vrijednosti "3", "4" i "7". Obratite pažnju da se pobrojane konstante kojima nije eksplicitno pridružena odgovarajuća vrijednost uvijek konvertiraju u vrijednost koja je za 1 veća od vrijednosti u koju se konvertira prethodna pobrojana konstanta.

Činjenica da se pobrojane konstante, u slučajevima kada nije određeno drugačije, automatski konvertiraju u cjelobrojne vrijednosti, daje utisak da su pobrojane konstante samo prerusene cjelobrojne konstante. Naime, ukoliko postupkom preklapanja operatora nije eksplicitno određeno drugačije, pobrojane konstante "Ponedjeljak", "Utorak", itd. u deklaraciji

```
enum Dani {Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak,  
Subota, Nedjelja};
```

ponašaju se u izrazima praktički identično kao cjelobrojne konstante iz sljedeće deklaracije:

```
const int Ponedjeljak(0), Utorak(1), Srijeda(2), Cetvrtak(3),  
Petak(4), Subota(5), Nedjelja(6);
```

U jeziku C je zaista tako. Pobrojane konstante u jeziku C su zaista samo prerusene cjelobrojne konstante, dok su promjenljive pobrojanog tipa samo prerusene cjelobrojne promjenljive. Međutim, u jeziku C++ dolazi do bitne razlike. U prethodna dva primjera, u prvom slučaju konstante "Ponedjeljak", "Utorak", itd. su tipa "Dani", dok su u drugom slučaju tipa "int". To ne bi bila toliko bitna razlika, da u jeziku C++ nije *strogog zabranjena* dodjela cjelobrojnih vrijednosti promjenljivim pobrojanog tipa, *bez eksplicitnog navođenja konverzije tipa*. Drugim riječima, sljedeća naredba nije legalna (ovdje pretpostavljamo da je "dan" promjenljiva tipa "Dani"):

```
dan = 5;
```

Ukoliko je baš neophodna dodjela poput prethodne, možemo koristiti eksplicitnu konverziju tipa:

```
dan = (Dani)5;
```

ili, u funkcionalnoj notaciji:

```
danasm = Dani(5);
```

S druge strane, u jeziku C je dozvoljena dodjela cjelobrojne vrijednosti promjenljivoj pobjoranog tipa, bez eksplisitnog navođenja konverzije tipa (bez obzira što je jezik C++ pravljen tako da bude visoko kompatibilan sa jezikom C u smislu da gotovo sve što radi u jeziku C radi i u jeziku C++, postoje ipak neka svojstva jezika C koja su ukinuta u jeziku C++). Osnovni razlog zbog kojeg su autori jezika C++ zabranili ovaku dodjelu je *sigurnost* (ova zabrana je uvedena relativno kasno, pa je neki stariji kompjajleri ne poštuju). Naime, u protivnom, bile bi moguće opasne dodjele poput dodjele

```
danasm = 17;
```

nakon koje bi promjenljiva “*danasm*” imala vrijednost koja izlazi izvan skupa dopuštenih vrijednosti koje promjenljive tipa “*Dani*” mogu imati. Također, bile bi moguće besmislene dodjele poput sljedeće (ovakve dodjele su u jeziku C nažalost moguće), bez obzira što je konstanta “*Poraz*” tipa “*Rezultat*”, a promjenljiva “*danasm*” tipa “*Dani*”:

```
danasm = Poraz;
```

Naime, imenovana konstanta “*Poraz*” konvertirala bi se u cjelobrojnu vrijednost, koja bi mogla biti legalno dodijeljena promjenljivoj “*danasm*”. Komitet za standardizaciju jezika C++ odlučio je da ovakve konstrukcije zabrani. S druge strane, posve je legalno (mada ne uvijek i previše smisleno) dodijeliti pobjoranu konstantu ili promjenljivu pobjoranog tipa cjelobrojnoj promjenljivoj (ovdje dolazi do automatske konverzije tipa), s obzirom da ovakva dodjela nije rizična. Spomenimo i to da je u jeziku C moguće bez konverzije dodijeliti promjenljivoj pobjoranog tipa vrijednost druge promjenljive nekog drugog pobjoranog tipa, dok je u jeziku C++ takva dodjela također striktno zabranjena.

Sigurnosni razlozi su također bili motivacija za zabranu primjene operatora poput “*++*”, “*--*”, “*+ =*” itd. nad promjenljivim pobjoranim tipovima (što je također dozvoljeno u jeziku C), kao i za zabranu čitanja promjenljivih pobjoranih tipova sa tastature pomoću operatora “*>>*”. Naime, ukoliko bi ovakve operacije bile dozvoljene, postavlja se pitanje šta bi trebala da bude vrijednost promjenljive “*danasm*” nakon izvršavanja izraza “*danasm++*” ukoliko je njena prethodna vrijednost bila “*Nedjelja*”, kao i kakva bi trebala da bude vrijednost promjenljive “*danasnji_rezultat*” nakon izvršavanja izraza “*danasnji_rezultat--*” ukoliko je njena prethodna vrijednost bila “*Poraz*”. U jeziku C operatori “*++*” i “*--*” prosto povećavaju odnosno smanjuju pridruženu cjelobrojnu vrijednost za 1 (npr. ukoliko je vrijednost promjenljive “*danasm*” bila “*Srijeda*”, nakon “*danasm++*” nova vrijednost postaje “*Cetvrtak*”), bez obzira da li novodobijena cjelobrojna vrijednost zaista odgovara nekoj pobjoranoj konstanti. U jeziku C++ ovakve nesigurne konstrukcije nisu dopuštene.

Razumije se da je u jeziku C++ moglo biti uvedeno da operatori “*++*” odnosno “*--*” uvijek prelaze na *sljedeći* odnosno *prethodnu* cjelobrojnu konstantu, i to na *kružnom principu* (po kojem iza konstante “*Nedjelja*” slijedi ponovo konstanta “*Ponedjeljak*”, itd.). Međutim, na taj način bi se ponašanje operatora poput “*++*” razlikovalo u jezicima C i C++, što se ne smije dopustiti. Pri projektiranju jezika C++ postavljen je striktan zahtjev da sve ono što istovremeno radi u jezicima C i C++ mora raditi *isto u oba jezika*. U suprotnom, postojali bi veliki problemi pri prenošenju programa iz jezika C u jezik C++. Naime, moglo bi se desiti da program koji radi ispravno u jeziku C počne raditi neispravno nakon prelaska na C++. Mnogo je manja nevolja ukoliko nešto što je radilo u jeziku C potpuno prestane da radi u jeziku C++. Naime, u tom slučaju, kompjajler za C++ će prijaviti grešku u prevodenju, tako da uvijek možemo ručno izvršiti modifikacije koje su neophodne da program proradi (što je svakako bolje u odnosu na program koji se *ispravno prevodi*, ali *ne radi ispravno*). Tako, na primjer, ukoliko želimo da simuliramo ponašanje operatora “*++*” nad promjenljivim cjelobrojnog tipa iz jezika C u jeziku C++, uvijek možemo umjesto “*danasm++*” pisati nešto poput

```
danasm = Dani(danasm + 1);
```

Doduše, istina je da na taj način nismo logički riješili problem na koji smo ukazali, već smo samo postigli da se program može ispravno prevesti. Alternativno, moguće je postupkom preklapanja operatora *dati značenje* operatorima poput “++”, “+”, “>>” itd. čak i kada se primijene na promjenljive pobrojanog tipa. Pri tome je moguće ovim operatorima dati značenje *kakvo god želimo* (npr. moguće je simulirati njihovo ponašanje iz jezika C, ili definirati neko inteligentnije ponašanje). O ovome ćemo detaljno govoriti u poglavlju o preklapanju operatora.

U C++ programima često se mogu vidjeti deklaracije poput sljedeće:

```
enum {X = 5, Y = 8, Z = 20};
```

Jasno je da su na ovaj način deklarirane tri pobrojane konstante “X”, “Y”, “Z” bezimenog tipa, ali pri tome nije definirana niti jedna promjenljiva odgovarajućeg tipa kojoj bi se ovakve konstante mogle dodjeljivati. Ove tri konstante ipak nisu neupotrebljive, s obzirom da se one mogu upotrebljavati u izrazima, pri čemu će njihove vrijednosti biti konvertirane redom u cijelobrojne vrijednosti “5”, “8” i “20”. Efektivno, možemo reći da nema nikakve razlike između prethodne deklaracije i deklaracije

```
const int X(5), Y(8), Z(20);
```

Jedina suštinska razlika između ove i prethodne deklaracije leži u činjenici da je prethodna deklaracija (pobrojanog tipa) dopuštena i u nekim kontekstima (npr. unutar deklaracija klase, koje ćemo upoznati kasnije) u kojima nije moguće na drugi način deklarirati *prave konstante*. Također, ova praksa je dijelom naslijedena iz jezika C, u kojem konstante deklarirane sa “**const**” nikada nisu bile prave konstante, dok pobrojane konstante deklarirane sa “**enum**” uvijek jesu prave konstante.

Osnovni razlog za uvođenje pobrojanih tipova leži u činjenici da njihovo uvođenje, mada ne doprinosi povećanju *funkcionalnosti* samog programa (u smislu da pomoću njih nije moguće uraditi ništa što ne bi bilo moguće uraditi bez njih), znatno povećava *razumljivost* samog programa, pružajući mnogo prirodniji model za predstavljanje pojmove iz stvarnog života, kao što su dani u sedmici, ili rezultati nogometne utakmice. Ukoliko rezultat utakmice može biti samo poraz, neriješen rezultat ili pobjeda, znatno prirodnije je definirati poseban tip koji može imati samo vrijednosti “Poraz”, “Nerijeseno” i “Pobjeda”, nego koristiti neko šifrovanje pri kojem ćemo jedan od ova tri moguća ishoda predstavljati nekim cijelim brojem. Također, uvođenje promjenljivih pobrojanog tipa može često ukloniti neke dileme sociološke prirode. Zamislimo, na primjer, da nam je potrebna promjenljiva “pol_zaposlenog” koja čuva podatak o polu zaposlednog radnika (radnice). Pošto pol može imati samo dvije moguće vrijednosti (“muško” i “žensko”, ako ignoriramo eventualne medicinske fenomene), neko bi mogao ovu promjenljivu deklarirati kao promjenljivu tipa “**bool**”, s obzirom da takve promjenljive mogu imati samo dvije moguće vrijednosti (“**true**” i “**false**”). Međutim, ovakva deklaracija otvara ozbiljne sociološke dileme da li vrijednost “**true**” iskoristiti za predstavljanje muškog ili ženskog pola. Da bismo izbjegli ovakve dileme, mnogo je prirodnije izvršiti deklaraciju tipa

```
enum Pol {Musko, Zensko};
```

a nakon toga prosto deklarirati promjenljivu “pol_zaposlenog” kao

```
Pol pol_zaposlenog;
```

Kao primjer upotrebe promjenljivih pobrojanog tipa, napisaćemo ponovo program koji prati pohađanje petodnevног kursa, koji je bio napisan u poglavlju o nizovima, ali ovaj put uz upotrebu pobrojanih tipova:

```

#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    enum RadniDan {Pon, Uto, Sri, Cet, Pet};
    int broj_polaznika[5];
    for(RadniDan dan = Pon; dan < Pet; dan = RadniDan(dan + 1)) {
        cout << "Unesi broj polaznika u toku " << dan + 1 << "dana: ";
        cin >> broj_polaznika[dan];
    }
    cout << "Dijagram prisustva\n"
        "-----\n";
    "Dan\n";
    for(RadniDan dan = Pon; dan < Pet; dan = RadniDan(dan + 1))
        cout << setw(2) << dan + 1 << " " << setfill('*')
            << setw(broj_polaznika[dan]) << "" << endl;
    return 0;
}

```

Paradoksalno je da, iako su promjenljive pobrojanog tipa uvedene da *povećaju čitljivost* programa, i učine program *lakšim za razumijevanje*, praksa je pokazala da, mada se iskusni programeri zaista slažu da je program koji koristi promjenljive pobrojanog tipa *jasniji, čitljiviji*, lakši i za *razumijevanje* i za *opravljanje*, njihova upotreba početnika često *zbunjuje*, i potrebno je dosta vremena da se početnik navikne na njihovo korištenje.

Na kraju napomenimo da se u jeziku C ključna riječ “**enum**” morala pisati i prilikom definiranja promjenljivih pobrojanog tipa, a ne samo pri deklaraciji pobrojanog tipa. Na primjer, promjenljiva “*danasjni_rezultat*” tipa “*Rezultat*” deklarirala bi se ovako:

```
enum Rezultat danasnji_rezultat;
```

Ovo se moglo izbjegći upotrebom naredbe “**typedef**”, na primjer na sljedeći način:

```
typedef enum _Rezultat_ {Poraz, Nerijeseno, Pobjeda} Rezultat;
```

nakon čega postaje moguća definicija promjenljive bez navođenja ključne riječi “**enum**” (ovdje zapravo uvodimo dva tipa: pomoći tip “*_Rezultat_*”, za koji je *potrebna* ključna riječ “**enum**”, i njegov sinonim “*Rezultat*” za koji ova ključna riječ nije potrebna). Mada ove zavrzlame rade i u jeziku C++, one više nisu potrebne.

14. Potprogrami i vidokrug identifikatora

Potprogrami (engl. *subroutines*) predstavljaju *samostalne dijelove kôda (programa)* koji *izvršavaju određeni zadatak*. Svrha potprograma je da poboljšaju strukturu programa tako da oni budu *modularni*, tj. da budu načinjeni od manjih jedinica koje su *neovisne jedna od druge koliko god je to moguće*. Modularni programi su jednostavniji kako za pisanje, tako i za čitanje, razumijevanje i vršenje naknadnih izmjena. Veoma često se dobro napisan potprogram koji je definiran i korišten u jednom programu *môže upotrijebiti bez ikakve izmjene u drugom programu*, što štedi vrijeme i trud, i omogućava *ponovnu iskoristivost napisanog kôda* (engl. *code reusability*).

Mnogi programski jezici (poput Pascal-a) poznaju dvije vrste potprograma: *procedure* i *funkcije*. U jeziku C++ javlja se samo jedna vrsta potprograma, i to su *funkcije*, mada se njihova suština često razlikuje od matematskog pojma funkcije. Zbog toga ćemo u ovom tekstu, dok se detaljnije ne upoznamo sa pojmom funkcije u jeziku C++ radije upotrebljavati izraz *potprogram*, jer je manje zbumujući za početnike. Ovdje treba napomenuti da pojmovi “potprogram” i “funkcija” *nisu sinonimi*. Naime, “potprogram” je *konceptualni pojam* nevezan za konkretni programski jezik, dok je “funkcija” *jedan od konkretnih načina realizacije* ovog koncepta (ujedno i *jedini način* realizacije ovog koncepta u jeziku C++).

Svakom potprogramu (funkciji) uvijek se *daje ime* (koje može biti bilo koji valjan *identifikator* koji nije iskorišten za nešto drugo), za koje je preporučljivo da bude vezano sa *zadatkom koji potprogram obavlja*. Pozivanje potprograma vrši se prostim navođenjem *imena potprograma* iza koje slijedi *par malih zagrada* (kasnije ćemo vidjeti da ovaj par zagrada zapravo formira *operator poziva funkcije*), čime se *daje analog potprogramu da obavi svoj zadatak* (što se zapravo svodi na *izvršavanje naredbi u tijelu potprograma*). Ovo će biti ilustrirano na jednostavnom primjeru programa u kojem je definiran jedan potprogram, nazvan “*PredstaviSe*”:

```
#include <iostream>
using namespace std;
void PredstaviSe() {
    cout << "Ja sam PC računar.\n";
    cout << "Star sam 3 godine.\n";
}
int main() { // Glavni program (glavna funkcija)
    cout << "Dobro jutro!\n";
    PredstaviSe();
    cout << "Želim vam lijep dan.\n";
    return 0;
}
```

Kada pokrenemo ovaj program, dobićemo sljedeći ispis na ekranu:

```
Dobro jutro!
Ja sam PC računar.
Star sam 3 godine.
Želim vam lijep dan.
```

Primijetimo da potprogram "PredstaviSe" ima istu formu kao i funkcija "main", samo što je *tip povratne vrijednosti* (*povratni tip*) "**int**" zamijenjen tipom "**void**", što u suštini znači da *povratne vrijednosti* *zapravo nema* (riječ "void" znači *ništa*). Iz istog razloga, potprogram ne sadrži naredbu "**return**" (ovakvi potprogrami zapravo odgovaraju pojmu *procedura* u programskim jezicima kao što je Pascal). O potprogramima koji *imaju povratnu vrijednost* govorićemo u kasnijim poglavljima. Što se tiče *imenama potprograma*, preporuka je da se za njihova imena koriste *glagolske forme u imperativu* (npr. "PredstaviSe") a ne imenice (npr. "Predstavljanje"), sa *velikim početnim slovom*.

Kao što je već rečeno na samom početku, kada se program pokrene, izvršavanje uvijek započinje od funkcije nazvane "main", bez obzira gdje se ona nalazi unutar programa. Njeno tijelo je "glavni program", dok su sve ostale funkcije potprogrami. Potprogrami se izvršavaju *onog trenutka kada se eksplicitno pozovu*. Osim u slučaju nekim veoma specifičnim situacijama, tijelo potprograma se *nikada neće izvršiti* prije nego što se potprogram *eksplicitno pozove*. Poziv potprograma se vrši navođenjem *njegovog imena* iza kojeg slijede zagrada, unutar kojih se navode *argumenti* (ili *parametri*) *potprograma* ukoliko postoje. Dok ne vidimo šta su argumenti i kako se koriste, naši potprogrami ih neće imati, tako da zgrade ostavljamo prazne. Nakon što se izvrši čitavo tijelo potprograma, izvršavanje programa se nastavlja od *sljedeće naredbe iza mesta gdje je potprogram pozvan*. Pri tome je sasvim moguće *isti potprogram pozvati sa više različitih mesta*. Tijelo potprograma će se propisno izvršiti svaki put kada se potprogram pozove. Strogo rečeno, i funkciju "main" možemo shvatiti kao potprogram, ali koji se poziva *iz samog operativnog sistema* onog trenutka kada se pokrene program.

Potprogram može imati i svoje vlastite promjenljive (uskoro ćemo vidjeti da su promjenljive u različitim potprogramima potpuno neovisne međusobno, čak i ukoliko imaju ista imena). Na primjer, ovdje je dat jednostavan potprogram koji čita dva broja i ispisuje njihov zbir:

```
void SaberiUlaze() {
    int prvi, drugi;
    cin >> prvi >> drugi;
    cout << prvi + drugi;
}
```

Za poziv ovog potprograma, koristimo naredbu koja se sastoji samo od *imena potprograma* i zagrada:

```
SaberiUlaze();
```

Ova naredba će uzrokovati da se kôd sadržan u potprogramu *izvrši*.

Može se postaviti pitanje kakva je korist od potprograma. Na prvom mjestu, oni nam omogućavaju da složenije programe iscijepkamo na *manje neovisne dijelove* koje je lakše pratiti i održavati. Dalje, potprogram se može u programu pozvati proizvoljan broj puta. Pretpostavimo, na primjer, da želimo da nam program ispiše tekst neke pjesme koja ima tri strofe, i refren nakon svake strofe. Bez pomoći potprograma, mi bismo tekst refrena morali ispisivati tri puta, nakon svake strofe. Uz pomoć potprograma, mogli bismo napisati program koji bi principijelno izgledao ovako:

```
#include <iostream>
using namespace std;
void IspisiPrvuStrofu() {
    cout << "Tekst prve strofe..." << endl;
}
void IspisiDruguStrofu() {
```

```

        cout << "Tekst druge strofe..." << endl;
    }

void IspisiTrecuStrofu() {
    cout << "Tekst treće strofe..." << endl;
}

void IspisiRefren() {
    cout << "Tekst refrena..." << endl;
}
int main() {
    IspisiPrvuStrofu();
    IspisiRefren();
    cout << endl;
    IspisiDruguStrofu();
    IspisiRefren();
    cout << endl;
    IspisiTrecuStrofu();
    IspisiRefren();
    return 0;
}

```

U ovom primjeru, umjesto da tri puta pišemo refren pjesme, mi smo tri puta pozvali potprogram „IspisiRefren” koji ispisuje tekst refrena, što dovodi do kraćeg i jasnijeg programa (naročito ukoliko je tekst refrena dugačak). Doduše, sličan efekat bi se, u principu, mogao postići i bez potprograma, pisanjem programa poput sljedećeg:

```

#include <iostream>
using namespace std;

int main() {
    for(int strofa = 1; strofa <= 3; strofa++) {
        switch(strofa) {
            case 1:
                cout << "Tekst prve strofe..." << endl;
                break;
            case 2:
                cout << "Tekst druge strofe..." << endl;
                break;
            case 3:
                cout << "Tekst treće strofe..." << endl;
        }
        cout << "Tekst refrena..." << endl;
        cout << endl;
    }
    return 0;
}

```

Međutim, neosporno je da je varijanta sa potprogramima jasnija, preglednija i lakša za održavanje.

Primijetimo da poredek u kojem se potprogrami definiraju *nije bitan*. Naime, rezultat izvršavanja programa zavisi od *redoslijeda kojim se potprogrami pozivaju*, a ne od *redoslijeda u kojem su definirani*, što je vidljivo i iz navedenog primjera.

Već smo ranije govorili da je vrijeme života promjenljivih koje su definirane unutar nekog bloka ograničeno do završetka bloka u kojem su definirane (osim ukoliko se neka promjenljiva specijalno ne proglaši za *statičku promjenljivu*, o čemu ćemo govoriti malo kasnije). To, naravno, vrijedi i za blok koji

predstavlja tijelo potprograma. Pored toga, promjenljivim definiranim unutar bloka nije ograničeno samo vrijeme života, već i *vidljivost*. Naime, ostatak programa koji se ne nalazi unutar istog bloka *uopće ne zna za njihovo postojanje!* Kaže se da su promjenljive definirane unutar nekog bloka *lokalne*: za njihovo postojanje se praktički ne zna izvan tog bloka. Dio programa u kojem je neko ime promjenljive ili bilo koji drugi identifikator dostupno naziva se *vidokrug*, *doseg* ili *opseg vidljivosti* (engl. *scope*) *identifikatora*.

Vidokrug promjenljivih ilustriraćemo na konkretnim primjerima. U sljedećem programu upotrebljen je potprogram nazvan “*IspisiPozdrav*” koji ne radi ništa posebno, osim što četiri puta ispisuje riječ “*Pozdrav!*” na ekran:

```
#include <iostream>
using namespace std;
void IspisiPozdrav() {
    int i;
    for(i = 1; i <= 4; i++) cout << "Pozdrav!\n";
}
int main() {
    int i = 10;
    cout << i << endl;
    IspisiPozdrav();
    cout << i << endl;
    return 0;
}
```

Nakon što pokrenemo ovaj program, dobićemo sljedeći ispis:

```
10
Pozdrav!
Pozdrav!
Pozdrav!
Pozdrav!
10
```

U ovom primjeru, potprogram “*IspisiPozdrav*” i glavna funkcija “*main*” koriste promjenljivu nazvanu “*i*”. Međutim, promjenljiva “*i*” definirana unutar potprograma “*pozdrav*” i promjenljiva “*i*” definirana unutar glavne funkcije “*main*” predstavljaju dvije *potpuno različite promjenljive*, bez obzira što imaju isto ime. Da bismo se uvjerili u to, ubacili smo dvije naredbe za ispis promjenljive “*i*” prije i nakon poziva potprograma “*IspisiPozdrav*”. Vidimo da iako je potprogram “*IspisiPozdrav*” koristio i mijenjao promjenljivu “*i*”, vrijednost promjenljive “*i*” iz glavne funkcije “*main*” je ostala kakva je bila i prije poziva potprograma “*IspisiPozdrav*”. Ovo je sasvim prirodno, jer su to dvije posve različite promjenljive, sa različitim vidokruzima. Vidokrug prve promjenljive “*i*” je tijelo potprograma “*IspisiPozdrav*”, dok je vidokrug druge promjenljive “*i*” tijelo glavnog programa (“*main*” funkcije). Na ovaj način je omogućeno da više različitih ljudi pišu razne potprograme istog programa neovisno jedan od drugog, tj. bez potrebe da znaju koje će promjenljive koristiti ostali potprogrami. Dvije promjenljive istog imena u dva različita potprograma “neće smetati” jedna drugoj.

Da je vidokrug promjenljivih ograničen samo na blok unutar kojeg su definirane, vidljivo je iz sljedećeg primjera:

```

#include <iostream>
using namespace std;
void Potprogram() {
    int i = 5;
}
int main() {
    Potprogram();
    cout << i;
    return 0;
}

```

Ovaj program se neće izvršiti, jer će kompjajler prijaviti grešku unutar funkcije “`main`”, s obzirom da vidokrug u kojem se promjenljiva “`i`” može koristiti prestaje završetkom tijela potprograma “`Potprogram`”. Također, vrijednosti promjenljivih se ne prenose automatski u potprogram prilikom poziva potprograma, tako da ni sljedeći program *neće raditi*:

```

#include <iostream>
using namespace std;
void Potprogram() {
    cout << i;
}
int main() {
    int i = 5;
    Potprogram();
    return 0;
}

```

Promjenljiva “`i`” naprosto *nije vidljiva* unutar potprograma “`Potprogram`”, s obzirom da je njen vidokrug ograničen na tijelo glavne funkcije. Kako se prenose promjenljive iz jednog potprograma u drugi, vidjećemo u sljedećem poglavljju.

Pored lokalnih promjenljivih postoje i *globalne promjenljive*. To su promjenljive koje su deklarirane *izvan svih blokova*. Njihov je vidokrug čitav program počev od mjesta na kojem su definirane pa sve do kraja programa. Također, njihovo vrijeme života je *od početka do kraja programa*. Na primjer, u sljedećem programu, promjenljiva “`i`” je globalna promjenljiva:

```

#include <iostream>
using namespace std;
int i;
void IspisiPozdrav() {
    for(i = 1; i <= 4; i++) cout << "Pozdrav!\n";
}
int main() {
    i = 10;
    cout << i << endl;
    IspisiPozdrav();
    cout << i << endl;
    return 0;
}

```

Ovdje je promjenljiva “`i`” zajednička i za potprogram “`IspisiPozdrav`” i za funkciju “`main`”, u šta se možemo uvjeriti ako prikažemo rezultate izvršavanja ovog programa:

```
10
Pozdrav!
Pozdrav!
Pozdrav!
Pozdrav!
5
```

Vidimo da promjenljiva “`i`” nije ostala sačuvana nakon poziva potprograma “`IspisiPozdrav`”. Zbog toga, upotrebu globalnih promjenljivih treba izbjegavati kada god je to moguće, jer njihova upotreba često dovodi do neželjenih tzv. *bočnih efekata* (engl. *side effects*). Mada smo pojам bočnog efekta već susreli u drugom kontekstu, ovdje pod bočnim efektima podrazumijevamo pojavu da poziv nekog potprograma izazove posljedice koje *nisu očigledne iz načina kako je potprogram pozvan*. Na primjer, prostim posmatranjem poziva potprograma “`IspisiPozdrav`” nije moguće zaključiti da će on izmijeniti vrijednost promjenljive “`i`”. U dugačkim programima ovakvi bočni efekti se često previde, što dovodi do programa koji ne rade onako kako korisnik očekuje. U sljedećem poglavlju ćemo vidjeti da se upotreba globalnih promjenljivih gotovo uvijek može potpuno izbjegći upotrebot parametara odnosno argumenata funkcija. Generalno, kao globalne promjenljive treba definirati samo one strukture podataka koje zajednički i planski treba da koristi više potprograma u istom programu, i to samo ako je upotreba argumenata za njihov prenos nepraktična.

Kod konstanti nema velike opasnosti da se deklariraju kao *globalne*, jer se njihova vrijednost ne može mijenjati. Tako, ukoliko želimo da deklariramo konstantu “`PI`” deklaracijom

```
const double PI(3.141592654);
```

pametno je da ovu deklaraciju stavimo *izvan svih blokova*, jer će tada njena vrijednost biti dostupna *svim potprogramima u programu*, a ne samo onom potprogramu unutar kojeg je definirana.

Primijetimo da u prethodnom primjeru u glavnom programu ne piše

```
int i = 10;
```

nego samo

```
i = 10;
```

Da smo napisali “`int i = 10`” (ili, što je isto, “`int i(10)`”) deklarirali bismo *lokalnu promjenljivu “i”* sa *istim imenom kao i globalna promjenljiva “i”*, i njen vidokrug bi bio tijelo funkcije “`main`”. Uskoro ćemo vidjeti šta se tačno dešava kada postoje lokalna i globalna promjenljiva istog imena.

Slijedi još jedan primjer koji demonstrira razliku između lokalnih i globalnih promjenljivih. U ovom programu, globalne promjenljive “`a`”, “`b`” i “`c`” mogu se koristiti unutar tijela potprograma “`P1`” i “`P2`”, ali lokalna promjenljiva “`d`”, na primjer, ne može se koristiti unutar tijela potprograma “`P2`”, ili unutar glavnog programa. Krajnji rezultat ovog programa je ispis brojeva 2, 3 i 2:

```
#include <iostream>
```

```

using namespace std;

int a, b, c;

void P1() {
    int d, e;
    d = a + 1;
    e = c;                                Vidokrug od "d" i "e"
    cout << d << " ";
    cout << e << " ";
}

void P2() {
    int f;                                Vidokrug od "f"
    f = b;
    cout << f << " ";
}

int main() {
    a = 1;                               // Glavni program
    b = 2;
    c = 3;
    P1();
    P2();
    return 0;
}

```

Ako u programu postoje i globalna i lokalna promjenljiva *istog imena*, lokalna promjenljiva *ima prioritet u pristupu* unutar svog vidokruga. Kažemo da je globalna promjenljiva *skrivena* (engl. *hidden*) istoimenom lokalnom promjenljivom. U ovo se možemo uvjeriti ako izvršimo sljedeći program:

```

#include <iostream>
using namespace std;

int i;

void IspisiPozdrav() {
    int i;
    for(i = 1; i <= 4; i++)           // Ovdje se pristupa lokalnoj promjenljivoj "i"
        cout << "Pozdrav!\n";
}

int main() {
    i = 10;
    cout << i << endl;
    IspisiPozdrav();
    cout << i << endl;
    return 0;
}

```

U ovom programu će oba puta biti isписан broj 10, jer se unutar potprograma “IspisiPozdrav” pristupa *lokalnoj* a ne *globalnoj* promjenljivoj “i”. Ista stvar je i u sljedećem programu:

```

#include <iostream>
using namespace std;

int i;

void IspisiPozdrav() {
    int i;

```

```

    for(i = 1; i <= 4; i++) cout << "Pozdrav!\n";
}

int main () {
    int i = 10;
    cout << i << endl;
    IspisiPozdrav();           // I ovdje se pristupa lokalnoj promjenljivoj "i"
    cout << i << endl;
    return 0;
}

```

Potpuno analognu situaciju smo razmatrali ranije, u kojoj promjenljiva definirana unutar nekog unutrašnjeg bloka koji je smješten unutar nekog drugog (spoljašnjeg) bloka skriva eventualnu istoimenu promjenljivu definiranu unutar spoljašnjeg bloka. U slučaju da je iz bilo kojeg razloga potrebno pristupiti skrivenoj promjenljivoj, možemo ispred njenog imena navesti unarni operator “`>:`”. Ipak, najbolje je izbjegći ovakva dupliranja imena (takva dupliranja se smatraju toliko lošom praksom da su potpuno zabranjena u jeziku Java, koji ima dosta sličnosti sa jezikom C++).

Globalne promjenljive, poput svih ostalih promjenljivih, mogu se inicijalizirati odmah prilikom njihovog definiranja. Pri tome se njihova inicijalizacija obavlja *prije nego što se glavni program uopće počne izvršavati* (drugim riječima, garantira se da će u trenutku kada se počne izvršavati prva naredba unutar tijela funkcije “`main`”, sve globalne promjenljive već biti inicijalizirane). Za razliku od lokalnih promjenljivih, sve globalne promjenljive koje nisu eksplicitno inicijalizirane *automatski se inicijaliziraju na vrijednost 0*. Drugim riječima, garantira se da će u trenutku kada program započne, sve globalne promjenljive kojima nije eksplicitno zadana početna vrijednost, imati vrijednost nula.

Skrivanje promjenljivih ilustriraćemo još jednim primjerom, u kojem je globalna promjenljiva “`b`” *skrivena* lokalnom promjenljivom “`b`” unutar tijela potprograma “`P2`” (ovdje se ponovo radi o dva potpuno različita i neovisna objekta):

```

#include <iostream>

using namespace std;

int a, b, c;

void P1() {
    int d;
    d = b;                      Vidokrug od      Vidokrug od
    cout << d << " ";          "d"                "(globalne)" "b"
}

void P2() {
    int b;
    b = a + c;                  Vidokrug od      Vidokrug od
    cout << b << " ";          "(lokalne)" "b"     "Vidokrug od"
}

int main() {
    a = 1;                      // Glavni program
    b = 2;
    c = 3;
    P1();
    P2();
    return 0;
}

```

Na ovom mjestu lokalna promjenljiva "b" skriva globalnu promjenljivu "b" prekidajući njen vidokrug "a" i "c"

Promjenljiva “*b*” koja se koristi u tijelu potprograma “*P1*” predstavlja *globalnu promjenljivu*. Stoga će efekat poziva potprograma “*P1*” u glavnom programu biti ispis broja “2”. Promjenljiva “*b*” kojoj je pridružena vrijednost u potprogramu “*P2*” predstavlja *lokalnu promjenljivu* koja sakriva vrijednost istoimene globalne promjenljive (kojoj bismo, u slučaju potrebe, eventualno mogli pristupiti konstrukcijom “*:b*”). Rezultat poziva ovog potprograma u glavnom programu biće ispis broja “4”, ali globalna promjenljiva “*b*” i dalje zadržava vrijednost “2”, u šta se možemo uvjeriti ukoliko ispišemo njenu vrijednost nakon poziva potprograma “*P2*”. Ovaj primjer jasno pokazuje da vidokrug i vrijeme života promjenljive nisu jedno te isto. Globalna promjenljiva “*b*” očigledno “živi” i tokom izvršavanja potprograma “*P2*”, mada u njemu nije vidljiva. Razlika između vidokruga i vremena života postaće još uočljivija nakon što se upoznamo sa pojmom statičkih promjenljivih.

Potprogrami podržavaju tehniku razvoja programa koja se obično naziva razvoj programa *odozgo na dolje* (engl. *top-down approach*). Naime, prilikom razvoja većih programa, programer obično ne može odmah uočiti sve neophodne aspekte programa. Zbog toga se programi obično razvijaju u etapama. U prvoj etapi skicira se grubo struktura programa. U drugoj etapi razrađuje se detaljnije svaki od koraka opisan u prvoj etapi. U svakoj narednoj etapi razrađuju se oni koraci koji su u prethodnoj etapi ostali nedovoljno razrađeni, itd. Postupak se ponavlja sve dok svaki od koraka ne bude razrađen dovoljno detaljno da se neposredno može prevesti u odgovarajuće instrukcije programskog jezika. Pri tome, potprogrami mogu učiniti razvoj programa po etapama lakšim, jer se svaki od koraka može implementirati neovisno jedan od drugog.

Opisani pristup ilustriraćemo na jednom relativno jednostavnom primjeru (koji ćemo, iz edukativnih razloga, “iscjepkati” više nego što je u stvarnim primjenama zaista neophodno). Prepostavimo da želimo napraviti program koji štampa lik šahovske table, koristeći praznine i zvjezdice, kao na sljedećoj slici, pri čemu se željena visina i širina svakog kvadrata može zadavati:

```
***** ***** ***** *****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
**** * *** * *** * ****
```

U prvoj etapi razvoja, odmah postaje jasno da od korisnika moramo tražiti unos širine i visine polja (izraženu u broju znakova, odnosno linija), prije nego što pristupimo na samo iscrtavanje šahovske table. Kako će ovi podaci morati biti dostupni u svim dijelovima programa, promjenljive koje čuvaju ove podatke deklariraćemo kao globalne promjenljive (dok ne upoznamo bolje načine kako se može ostvariti razmjena informacija između pojedinih potprograma):

```
int sirina_polja; // Širina kvadratnog polja
int visina_polja; // Visina kvadratnog polja
```

Učitavanje informacija o širini i visini polja je trivijalan zadatak. Glavnina posla je u samom iscrtavanju table. Odlučimo li se da ovaj dio problema izdvojimo u poseban potprogram (nazvan npr. "StampajTablu"), glavni program bi mogao izgledati ovako:

```
int main() {
    cout << "Ovaj program prikazuje šahovsku tablu.\n\n"
    "Unesite širinu svakog kvadrata, u znakovima: ";
    cin >> sirina_polja;
    cout << "Unesi visinu svakog kvadrata, u linijama: ";
    cin >> visina_polja;
    cout << "\n\n";
    StampajTablu();
    return 0;
}
```

Ovim smo završili prvu etapu razvoja. U drugoj etapi razmotrimo kako bi se mogao realizirati potprogram "StampajTablu". Možemo primjetiti da se u šahovskoj tabli razlikuje štampanje parnih i neparnih redova šahovske table, i da se četiri puta ponavlja identično iscrtavanje prvo neparnog pa parnog reda šahovske table. Odlučimo li se da štampanje neparnih i parnih redova razbijemo u posebne potprograme, potprogram "StampajTablu" mogao bi izgledati ovako:

```
void StampajTablu() {
    for(int i = 1; i <= 4; i++) {
        StampajNeparniRed();
        StampajParniRed();
    }
}
```

```
}
```

U narednoj etapi možemo uočiti da se štampanje neparnog reda šahovske table svodi na štampanje jedne linije neparnog reda onoliko puta koliko iznosi visina jednog polja šahovske table. Isto vrijedi i za štampanje parnog reda šahovske table. Stoga bi traženi potprogrami "StampajParniRed" i "StampajParniRed" mogli izgledati ovako (imena upotrijebljenih potprograma su pomalo neprirodno dugačka, ali ovdje radi razumljivosti želimo jasno istaći njihovu namjenu):

```
void StampajNeparniRed() {
    for(int i = 1; i <= visina_polja; i++) StampajLinijuNeparnogReda();
}

void StampajParniRed() {
    for(int i = 1; i <= visina_polja; i++) StampajLinijuParnogReda();
}
```

U sljedećoj etapi razvoja, potrebno je razmotriti kako odštampati jednu liniju koja pripada neparnom ili parnom redu. Za štampanje linije neparnog reda potrebno je četiri puta odštampati grupu razmaka iza koje slijedi grupa zvjezdica, nakon čega treba preći u novi red. Za štampanje linije parnog reda sve je isto, samo se mijenja uloga razmaka i zvjezdica. Stoga potprogrami "StampajLinijuNeparnogReda" i "StampajLinijuParnogReda" mogu izgledati ovako:

```
void StampajLinijuNeparnogReda() {
    for(int i = 1; i <= 4; i++) {
        StampajRazmake();
        StampajZvjezdice();
    }
    cout << endl;
}

void StampajLinijuParnogReda() {
    for(int i = 1; i <= 4; i++) {
        StampajZvjezdice();
        StampajRazmake();
    }
    cout << endl;
}
```

Konačno, ostaje nam da preciziramo šta treba da rade potprogrami "StampajRazmake" odnosno "StampajZvjezdice". Očigledno je potrebno ispisati onoliko razmaka (odnosno zvjezdica) koliko iznosi širina jednog polja šahovske table, tako da bi ovi potprogrami mogli izgledati ovako (eventualno je "for" petlju moguće izbjegći upotrebo manipulatora "setw" i "setfill"):

```
void StampajRazmake() {
    for(int i = 1; i <= sirina_polja; i++) cout << " ";
}

void StampajZvjezdice() {
    for(int i = 1; i <= sirina_polja; i++) cout << "*";
}
```

Ovim je razrada algoritma gotova. Sada ćemo sve razrađene dijelove programa (odnosno pojedine potprograme) sklopiti u cjeloviti program. Pri ovom sklapanju se javlja jedan praktičan problem. Naime, slično kao što u programu nije moguće koristiti promjenljivu koja nije prethodno definirana, tako nije moguće pozvati potprogram ukoliko on nije prethodno definiran u programu, ili ukoliko kompjajler nije prethodno na neki način barem obavijesten o njegovom postojanju. Uskoro ćemo naučiti način kako

možemo obavijestiti kompjajler o postojanju nekog potprograma *bez njegovog definiranja*, uz obećanje da će isti potprogram biti definiran *naknadno* (u ovom slučaju govorimo samo o *deklaraciji* ali ne i o *definiciji* potprograma). Dok to ne naučimo, jedini način da sklopimo ove potprograme u program koji funkcioniра je da poredamo potprograme tako da se ni jedan potprogram ne poziva *prije nego što bude u potpunosti definiran*. Na taj način, dolazimo do sljedećeg programa:

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

int sirina_polja; // Širina kvadratnog polja
int visina_polja; // Visina kvadratnog polja

void StampajRazmake() {
    for(int i = 1; i <= sirina_polja; i++) cout << " ";
}

void StampajZvjezdice() {
    for(int i = 1; i <= sirina_polja; i++) cout << "*";
}

void StampajLinijuNeparnogReda() {
    for(int i = 1; i <= 4; i++) {
        StampajRazmake();
        StampajZvjezdice();
    }
    cout << endl;
}

void StampajLinijuParnogReda() {
    for(int i = 1; i <= 4; i++) {
        StampajZvjezdice();
        StampajRazmake();
    }
    cout << endl;
}

void StampajNeparniRed() {
    for(int i = 1; i <= visina_polja; i++) StampajLinijuNeparnogReda();
}

void StampajParniRed() {
    for(int i = 1; i <= visina_polja; i++) StampajLinijuParnogReda();
}

void StampajTablu() {
    for(int i = 1; i <= 4; i++) {
        StampajNeparniRed();
        StampajParniRed();
    }
}

int main() {
    cout << "Ovaj program prikazuje šahovsku tablu.\n\n"
    "Unesite širinu svakog kvadrata, u znakovima: ";
    cin >> sirina_polja;
    cout << "Unesite visinu svakog kvadrata, u linijama: ";
    cin >> visina_polja;
    cout << "\n\n";
```

```

    StampajTablu();
    return 0;
}

```

U ovom, kao i u svim dosadašnjim primjerima, svaki potprogram je bio *u potpunosti definiran prije nego što je pozvan*. Ukoliko želimo da neki potprogram definiramo *iza mesta na kojem ga pozivamo*, tada se negdje u programu prije mesta poziva obavezno mora navesti *deklaracija* ili, kako se to često kaže, *prototip* tog potprograma. Deklaracija (odnosno prototip) potprograma sastoji se *samo od zaglavlja potprograma*, bez tijela, pri čemu umjesto tijela slijedi znak tačka-zarez (dakle, tijelo prototipa je *prazno*). Da smo u primjeru koji koristi potprogram "IspisiPozdrav" htjeli da prvo definiramo glavni program, pa tek onda potprogram, napisali bismo sljedeće:

```

#include <iostream>

using namespace std;

void IspisiPozdrav(); // Ovo je prototip potprograma "IspisiPozdrav"

int main() {
    int i = 10;
    cout << i << endl;
    IspisiPozdrav();
    cout << i << endl;
    return 0;
}

void IspisiPozdrav() {
    for(int i = 1; i <= 4; i++) cout << "Pozdrav!\n";
}

```

Da nismo upotrebili prototip, kompjajler bi prijavio grešku na mjestu poziva potprograma. Koja je svrha prototipova? Kompajler *ne mora* da zna u trenutku poziva potprograma šta potprogram radi niti kako je definiran, ali *mora da zna* da li taj potprogram ima argumente, i ako ih ima, koliko ih ima i kakvog su tipa (da bi mogao da prijavi grešku ako zadamo pogrešan broj argumenata, ili pogrešne tipove argumenata). Također, kompjajler mora da zna eventualni tip povratne vrijednosti koju potprogram eventualno vraća. Neko bi mogao pomisliti da bi kompjajler sve ovo mogao utvrditi i bez prototipova, tako što bi prilikom poziva potprograma prosto pretražio čitav program, bez oslanjanja samo na ono što je do tada navedeno. Međutim, ovo u općem slučaju nije nimalo praktično, jer se potprogrami uopće ne moraju definirati u istoj datoteci u kojoj se nalazi poziv potprograma! Naime, pri razvoju velikih programa, čest je slučaj da se program razbija u više datoteka (koje se objedinjuju u tzv. *projekte*), koji se mogu prevoditi i razvijati posve neovisno. U tom slučaju, takozvani *povezivač* ili *linker* preuzima na sebe zadatku povezivanja neovisno prevedenih dijelova programa u kompletan program. Prototipovi su tada zaista neophodni, jer prevodilac (kompajler) ne može znati da li je potprogram možda definiran u nekoj drugoj datoteci. Ukoliko se desi da je postojanje potprograma *najavljen* (navođenjem prototipa), a njegova definicija ne bude pronađena *nigdje*, biće prijavljena greška, koja eventualno može biti otkrivena tek u fazi povezivanja (to se dešava u slučaju razbijanja programa u više datoteka).

Prototip potprograma se može navesti i unutar bloka u kojem potprogram pozivamo (pri čemu onda taj prototip važi samo unutar tog bloka), kao u sljedećem primjeru:

```

#include <iostream>

using namespace std;

int main() {
    void IspisiPozdrav(); // prototip
    int i = 10;
    cout << i << endl;
}

```

```

    IspisiPozdrav();
    cout << i << endl;
}

void IspisiPozdrav() {
    for(int i = 1; i <= 4; i++) cout << "Pozdrav!\n";
}

```

Obratimo pažnju na još jedan sitni detalj: promjenljiva “*i*” u potprogramu “IspisiPozdrav” je, na izvjestan način, *dvostruko lokalna*: ona ne samo da je ograničena samo na tijelo potprograma “IspisiPozdrav”, već je ograničena na *tijelo “for” petlje unutar ovog potprograma*.

Kao što smo rekli, prototipovi potprograma nam omogućavaju da u programu prvo navedemo *poziv potprograma*, a da tek kasnije definiramo šta (i kako) potprogram treba da radi (ta definicija se, kao što smo već rekli, može čak nalaziti i u drugoj datoteci, samo što u tom slučaju kompjajler treba na neki način obavijestiti koje datoteke sačinjavaju program, što se postiže definiranjem tzv. *projekata*). Ovakav koncept mnogo je bliži prirodnom toku misli koji koristimo u etapnom razvoju programa. Na primjer, kada smo razvijali program za prikaz šahovske table, prvo smo zaključili da se štampanje table svodi na naizmjenično štampanje parnih i neparnih redova, a tek smo kasnije analizirali kako se stampaju parni i neparni redovi, itd. Prethodna verzija programa za štampanje šahovske table bila je neprirodno ispreturana, zbog potrebe da potprogrami budu definirani prije mjesta poziva. Slijedi modificirana verzija ovog programa, koja *tačno slijedi tok misli* koji smo koristili pri etapnom razvoju:

```

#include <iostream>

using namespace std;

int sirina_polja;                                // Širina kvadratnog polja
int visina_polja;                                // Visina kvadratnog polja

int main() {                                       // Glavni program
    void StampajTablu();
    cout << "Ovaj program prikazuje šahovsku tablu.\n\n"
        "Unesite širinu svakog kvadrata, u znakovima: ";
    cin >> sirina_polja;
    cout << "Unesite visinu svakog kvadrata, u linijama: ";
    cin >> visina_polja;
    cout << "\n\n";
    StampajTablu();
}

void StampajTablu() {                            // Štampa šahovsku tablu
    void StampajNeparniRed();
    void StampajParniRed();
    for(int i = 1; i <= 4; i++) {
        StampajNeparniRed();
        StampajParniRed();
    }
}

void StampajNeparniRed() {                      // Štampa neparni red
    void StampajLinijuNeparnogReda();
    for(int i = 1; i <= visina_polja; i++) StampajLinijuNeparnogReda();
}

void StampajParniRed() {                        // Štampa parni red
    void StampajLinijuParnogReda();
    for(int i = 1; i <= visina_polja; i++) StampajLinijuParnogReda();
}

```

```

}

void StampajLinijuNeparnogReda() { // Štampa liniju neparnog reda
    void StampajRazmake();
    void StampajZvjezdice();
    for(int i = 1; i <= 4; i++) {
        StampajRazmake();
        StampajZvjezdice();
    }
    cout << endl;
}

void StampajLinijuParnogReda() { // Štampa liniju parnog reda
    void StampajRazmake();
    void StampajZvjezdice();
    for(int i = 1; i <= 4; i++) {
        StampajZvjezdice();
        StampajRazmake();
    }
    cout << endl;
}

void StampajRazmake() { // Štampa "sirina_polja" razmaka
    for(int i = 1; i <= sirina_polja; i++) cout << " ";
}

void StampajZvjezdice() { // Štampa "sirina_polja" zvjezdica
    for(int i = 1; i <= sirina_polja; i++) cout << "*";
}

```

Klasične lokalne promjenljive nazivaju se i *automatske promjenljive*, s obzirom da se one automatski kreiraju (i eventualno inicijaliziraju) svaki put kada tok programa dovede do njihove deklaracije, i automatski uništavaju svaki put kada tok programa dovede do kraja bloka unutar kojeg su definirane. Na primjer, u sljedećoj sekvenci naredbi

```

for(int i = 1; i <= 10; i++) {
    int kvadrat = i * i;
    cout << i << " na kvadrat je " << kvadrat;
}

```

promjenljiva “kvadrat” se stvara (uz inicijalizaciju) i uništava 10 puta, odnosno pri svakom prolazu kroz petlju. Može se postaviti pitanje kako ovo utiče na efikasnost. U suštini, stvaranje odnosno uništavanje promjenljivih jednostavnih tipova (kakvi su svi tipovi koje smo do sada upoznali) veoma je prosta operacija sa aspekta računara, tako da je eventualni gubitak na efikasnosti sasvim zanemarljiv. Međutim, u slučaju složenih tipova koje ćemo kasnije upoznati (čije kreiranje i uništavanje zahtijeva pozivanje tzv. *konstruktora i destruktora*) gubitak efikasnosti može biti primjetan. O tome ćemo detaljnije govoriti kasnije.

Pored automatskih lokalnih promjenljivih, postoje i tzv. *statičke lokalne promjenljive*. Sa aspekta vidljivosti, ove promjenljive se ponašaju poput običnih lokalnih promjenljivih, odnosno dostupne su samo unutar bloka unutar kojeg su definirane. Međutim, sa aspekta vremena života, ove promjenljive ponašaju se poput *globalnih promjenljivih*. Naime, njihov život ne prestaje po završetku bloka unutar kojeg su definirane, već traje do kraja programa. Ove promjenljive se stvaraju i inicijaliziraju onog trenutka kada tok programa prvi put dovede do njihove deklaracije, i samo tada. Da bismo bolje uvidjeli razliku između automatskih i statičkih promjenljivih, razmotrimo sljedeći program:

```
#include <iostream>
```

```

using namespace std;

void P() {
    int a = 5;
    cout << a;
    a++;
    cout << a;
}

int main() {
    P();
    P();
    P();
    return 0;
}

```

Nije teško uvidjeti da je efekat ovog programa ispis "565656", s obzirom da se lokalna promjenljiva "a" iznova *stvara i inicijalizira* svaki put kada započne izvršavanje potprograma "P" (izazvano njegovim pozivom), i *uništava* po njegovom završetku. Međutim, situacija postaje posve drugačija proglašimo li lokalnu promjenljivu "a" statičkom, što se postiže dodavanjem ključne riječi "**static**" ispred njene deklaracije (slično, ključna riječ "**auto**" ispred deklaracije označava automatsku promjenljivu, ali se ova ključna riječ praktično nikad ne koristi, s obzirom da se podrazumijeva u slučaju da se ne navede ključna riječ "**static**"), kao u sljedećem programu:

```

#include <iostream>

using namespace std;

void P() {
    static int a = 5;
    cout << a;
    a++;
    cout << a;
}

int main() {
    P();
    P();
    P();
    return 0;
}

```

Za razliku od prethodnog programa, ovaj program dovodi do ispisa "566778". Naime, statička promjenljiva "a" se stvara i inicijalizira na vrijednost "5" samo pri prvom pozivu potprograma. Nakon što se njena vrijednost pod dejstvom operatora "++" poveća za 1, ona *nastavlja da živi* i nakon završetka potprograma. Pri drugom pozivu potprograma, ponovnim nailaskom na deklaraciju promjenljive "a", uočava se da *ona već postoji*, zbog čega se *ne vrši njena ponovna inicijalizacija* (ne zaboravimo da se inicijalizacija uvijek obavlja isključivo nad objektom koji je upravo stvoren), odnosno vrijednost ove promjenljive ostaje "6". Nije teško upratiti šta se dalje dešava. S druge strane, ukoliko umjesto *inicijalizacije* upotrijebimo *dodjelu*, odnosno ukoliko potprogram "P" napišemo na sljedeći način:

```

void P() {
    static int a;
    a = 5;
    cout << a;
    a++;
}

```

```

    cout << a;
}

```

tada će efekat izvršavanja programa ponovo biti ispis “565656”, s obzirom da se dodjela vrši nad objektom *koji već postoji*, uništavanjem njegovog prethodnog sadržaja. Ovim se na veoma jasan način uočava razlika između *inicijalizacije* i *dodgele*, mada prethodni primjeri djeluju *dosta zbumujuće*, s obzirom da se u oba primjera koristi znak “=”. To je još jedan od razloga zbog čega se preporučuje da se za inicijalizaciju uvijek koristi sintaksa u kojoj se ne koristi znak “=”, nego se početna vrijednost navodi u zagradama. Zaista, do manje će zabune doći ukoliko potprogram “P” napišemo na sljedeći način:

```

void P() {
    static int a(5);
    cout << a;
    a++;
    cout << a;
}

```

Bitno je napomenuti da spomenuto svojstvo statičkih promjenljivih ne vrijedi samo za promjenljive deklarirane unutar potprograma, već za *lokalne promjenljive uopće* (tj. za bilo kakve promjenljive definirane unutar blokova). Na primjer, iako će sekvenca naredbi

```

for(int i = 1; i <= 3; i++) {
    int a = 5;
    cout << a;
    a++;
    cout << a;
}

```

dvesti do očekivanog ispisa “565656”, sljedeća sekvenca naredbi

```

for(int i = 1; i <= 3; i++) {
    static int a = 5;
    cout << a;
    a++;
    cout << a;
}

```

će ispisati “566778”, s obzirom da će statička promjenljiva “a” biti stvorena (i inicijalizirana) samo pri prvom prolasku kroz petlju. Da bi se smanjila konfuzija, prethodna dva primjera bi bolje bilo pisati kao

```

for(int i = 1; i <= 3; i++) {
    int a(5);
    cout << a;
    a++;
    cout << a;
}

```

odnosno kao

```

for(int i = 1; i <= 3; i++) {
    static int a(5);
    cout << a;
    a++;
    cout << a;
}

```

Statičke promjenljive se ne koriste osobito često, i služe uglavnom kada je potrebno da neka informacija “preživi” kraj potprograma i bude dostupna pri njegovom ponovnom pozivu. Na primjer, neka je potrebno napraviti potprogram “*IspisiBrojPoziva*” koji ispisuje koliko je puta pozvan. Jedna mogućnost je da definiramo neku *globalnu promjenljivu*, s obzirom da ona postoji *neovisno od potprograma* (tako da može “preživjeti” njegov završetak), kao u sljedećem primjeru:

```
#include <iostream>
using namespace std;
int broj_poziva(1);
void IspisiBrojPoziva() {
    cout << "Ovo je " << broj_poziva << ". poziv\n";
    broj_poziva++;
}
int main() {
    for(int i = 1; i <= 5; i++) IspisiBrojPoziva();
    IspisiBrojPoziva();
    return 0;
}
```

Pokretanjem ovog programa dobijamo sljedeći ispis:

```
Ovo je 1. poziv
Ovo je 2. poziv
Ovo je 3. poziv
Ovo je 4. poziv
Ovo je 5. poziv
Ovo je 6. poziv
```

Naime, potprogram “*IspisiBrojPoziva*” je zaista pozvan 6 puta (pet puta iz “**for**” petlje, i šesti put nakon nje). Mana ovog rješenja je u tome što promjenljiva “*broj_poziva*” koja je iskorištena za brojanje poziva ima vidokrug koji se proteže na čitav program, mada je njeno funkcioniranje neophodno samo unutar potprograma “*IspisiBrojPoziva*”. To ostavlja mogućnost da njena vrijednost bude nehotično promijenjena negdje izvan potprograma (npr. negdje unutar funkcije “*main*”), čime će biti narušen ispravan rad ovog potprograma. Također, nije dobro što je promjenljiva “*broj_poziva*”, koja je od vitalnog značaja za rad potprograma “*IspisiBrojPoziva*”, definirana *izvan njega*, odnosno *ne predstavlja njegov sastavni dio*, čime potprogram gubi na samostalnosti i neovisnosti od ostatka programa (što je veoma bitan cilj, kao što ćemo uskoro pokazati). Stoga je mnogo bolje *suziti vidokrug ove promjenljive* tako da obuhvati samo tijelo potprograma “*IspisiBrojPoziva*”. To možemo uraditi kao u sljedećem rješenju:

```
#include <iostream>
using namespace std;
void IspisiBrojPoziva() {
    static int broj_poziva(1);
    cout << "Ovo je " << broj_poziva << ". poziv\n";
    broj_poziva++;
}
int main() {
```

```
    for(int i = 1; i <= 5; i++) IspisiBrojPoziva();  
    IspisiBrojPoziva();  
    return 0;  
}
```

Statička promjenljiva je neophodna da bi “preživila” kraj potprograma (u suprotnom bi njena vrijednost pri svakom pozivu bila ponovo inicijalizirana na vrijednost “1”). Dakle, statičke lokalne promjenljive su *lokalne samo po vidokrugu*, ali su *globalne po vremenu života*.

Na kraju napomenimo da statičke promjenljive posjeduju još jednu osobinu koja ih povezuje sa globalnim promjenljivim. Naime, vrijednosti svih statičkih promjenljivih se, poput globalnih promjenljivih, *automatski inicijaliziraju na nulu* prilikom njihovog kreiranja, ukoliko eksplisitno nije navedena njihova početna vrijednost.

15. Prenos parametara u potprograme

Upotreba globalnih promjenljivih često može dovesti do grešaka u programima koje se teško otkrivaju, s obzirom da im je vidokrug izuzetno velik, tako da je velika i mogućnost njihove upotrebe na pogrešan način. Također, upotreba globalnih promjenljivih u više različitih potprograma dovodi do stvaranja neprirodne zavisnosti između potprograma, koji ovise od imena dijeljenih globalnih promjenljivih. Međutim, do sada nam je upotreba globalnih promjenljivih bila jedini način da izvršimo razmjenu informacija između više različitih potprograma. Srećom, C++ nudi mnogo praktičniji i sigurniji način razmijene informacija između potprograma zasnovan na tzv. *prenosu parametara*, koji ne dovodi do stvaranja zavisnosti između pojedinih potprograma.

Potrebu za prenosom parametara ilustriraćemo na primjeru sljedećeg programa koji računa i štampa obim i površinu kruga:

```
#include <iostream>
using namespace std;
const double PI(3.141592654);
double poluprecnik;
// Štampa obim i površinu kruga sa poluprečnikom "poluprecnik"
void ProracunajKrug() {
    cout << "Obim: " << 2 * PI * poluprecnik << endl
        << "Površina: ", PI * poluprecnik * poluprecnik << endl;
}
int main() { // Glavni program
    cin >> poluprecnik;
    ProracunajKrug();
    return 0;
}
```

Iako nema nikakve sumnje da ovaj program radi ispravno, u njemu se mogu uočiti i brojni problemi, koji nisu vezani za njegovo *funkcioniranje*, već za njegovu *strukturu i mogućnost prilagođavanja*. Na prvom mjestu, komunikacija između glavnog programa i potprograma “ProracunajKrug” ostvarena je preko zajedničke globalne promjenljive “poluprecnik”. Ukoliko se njeno ime promijeni, potprogram će imati grešku, s obzirom da se oslanja na vrijednost nedeklarirane promjenljive. U nekom složenijem okruženju, moglo bi se čak desiti da izmjena imena promjenljive dovede do toga (što je još gore) da program formalno nema sintaksnu grešku, ali da radi pogrešno (to se, na primjer, može desiti ukoliko se u nekom drugom potprogramu neočekivano upotrebni ista promjenljiva, a da te činjenice ovaj potprogram nije svjestan). To nije ono što zaista želimo. Naime, potprogram bi trebao biti neovisna jedinica kôda, *neovisna od ostatka programa koliko god je to moguće*. Trebali bismo biti veoma pažljivi, i možda bismo morali izmijeniti dio kôda, ukoliko bismo željeli da potprogram “ProracunajKrug” prosto upotrijebimo u nekom drugom programu. Međutim, dobro napisan potprogram ne bi trebao ništa da “zna” o tome šta se nalazi u ostatku programa, niti kako će i gdje će on biti upotrebljen u ostatku programa. Dobro napisan potprogram trebao bi samo da “radi” posao koji mu je povjeren, bez “razmišljanja” o tome *kome i zašto* je taj posao potreban.

Na sličan problem nailazimo i ukoliko želimo proširiti naš program koji ispisuje pozdrav na ekranu, tako da možemo zadati koliko puta želimo da se ispiše riječ “Pozdrav!”. Ovo zadavanje trebamo izvršiti u glavnom programu, koji poziva potprogram “IspisiPozdrav”, prije njegovog poziva. Međutim,

potprogram “IspisiPozdrav” mora imati način da sazna ovu vrijednost. Ukoliko se ograničimo samo na ono što smo do sada utvrdili, jedino što možemo uraditi je da iskoristimo globalne promjenljive, kao u sljedećem primjeru:

```
#include <iostream>
using namespace std;
int broj_ponavljanja;
void IspisiPozdrav() {
    for(int i = 1; i <= broj_ponavljanja; i++) cout << "Pozdrav!\n";
}
int main() {
    cout << "Koliko puta želite pozdrav? ";
    cin >> broj_ponavljanja;
    IspisiPozdrav();
    return 0;
}
```

Loše strane ovog programa su iste kao u prethodnom primjeru. Rješenje ovih problema nadeno je uvođenjem tehnike *prenosa parametara*. Pri tome se razlikuje pojam *formalnih i stvarnih* (ili *aktualnih*) parametara. *Formalni parametri* su specijalna vrsta lokalnih promjenljivih koje ne inicijalizira sam potprogram, već njihovom inicijalizacijom upravlja *onaj ko poziva potprogram*. Za razliku od običnih lokalnih promjenljivih, deklaracija formalnih parametara se navodi *unutar zagrada* koje se nalaze u *zaglavlju potprograma*. Inicijalne vrijednosti formalnih parametara zadaju se *navođenjem željenih inicijalnih vrijednosti* (koje se zovu *stvarni parametri*) unutar zagrada prilikom *poziva potprograma*. Slijedi poboljšana verzija programa za računanje obima i površine kruga, koja koristi ovu tehniku.

```
#include <iostream>
using namespace std;
const double PI(3.141592654);
// Štampa obim i povrsinu kruga sa poluprečnikom zadanim kao parametar
void ProracunajKrug(double r) {
    cout << "Obim: " << 2 * PI * r << endl;
    cout << "Površina: " << PI * r * r << endl;
}
int main() { // Glavni program
    double poluprecnik;
    cin >> poluprecnik;
    ProracunajKrug(poluprecnik);
    return 0;
}
```

Korištenje parametara omogućava potprogramu “ProracunajKrug” da koristi svoje *vlastito lokalno ime* za poluprečnik kruga (u navedenom primjeru “*r*”) koje ne mora nužno imati nikakve veze sa imenom promjenljive (u navedenom primjeru “*poluprecnik*”) koju koristi glavni program (koji poziva ovaj potprogram) za čuvanje informacije o poluprečniku kruga. Prilikom poziva potprograma “ProracunajKrug”, tekuća vrijednost promjenljive “*poluprecnik*” (stvarni parametar) koristi se za inicijalizaciju lokalne promjenljive (formalnog parametra) “*r*”, odnosno vrijednost promjenljive “*poluprecnik*” se *kopira* u formalni parametar “*r*”. Poput svih drugih lokalnih promjenljivih (osim statičkih), formalni parametri su vidljivi samo unutar odgovarajućeg potprograma, i automatski se uništavaju nakon završetka potprograma. Formalni parametri ne mogu biti statički.

Kao što je već rečeno, formalni parametri se deklariraju unutar samog zaglavlja potprograma. Deklaracija parametara se često naziva *popis parametara* (engl. *parameter list*). Iz izloženog slijedi da su formalni parametri *jedinstveni* za konkretni potprogram, odnosno oni su jednoznačno definirani samim zaglavljem potprograma:

popis parametara

```
void ProracunajKrug(double r)  
    formalni parametar
```

S druge strane, stvarni parametri se navode prilikom poziva potprograma sa ciljem da inicijaliziraju odgovarajuće formalne parametre:

```
ProracunajKrug(poluprecnik)  
    stvarni parametar
```

U navedenom primjeru, stvarni parametar “*poluprecnik*” kopira se u formalni parametar “*r*”. S obzirom da se stvarni parametri navode prilikom poziva, a isti potprogram je moguće pozvati koliko god puta želimo, u svakom pozivu moguće je zadati drugačije vrijednosti stvarnih argumenata, odnosno stvarni argumenti *nisu jedinstveno određeni* samim potprogramom. U tom slučaju, prilikom svakog izvršavanja potprograma, formalni parametri će imati drugačije početne vrijednosti (određene vrijednostima stvarnih parametara).

Za razliku od formalnih parametara koji su *promjenljive*, stvarni parametri ne moraju biti promjenljive, već mogu biti *bilo koje vrijednosti ispunjivog tipa*, što uključuje promjenljive, konstante ili izraze. Stoga su pozivi poput sljedećih sasvim korektni (uz pretpostavku da postoji i promjenljiva “*precnik*” tipa “**double**”):

```
ProracunajKrug(poluprecnik);  
ProracunajKrug(7 * 3.18 - 2.27);  
ProracunajKrug(3.15);  
ProracunajKrug(precnik / 2);
```

Broj stvarnih parametara koji se prenose u potprogram *mora biti jednak* broju formalnih parametara (u našem primjeru jedan), osim u slučaju postojanja tzv. *podrazumijevanih parametara*, o kojima ćemo govoriti nešto kasnije. Svaki stvarni parametar *mora odgovarati po tipu* odgovarajućem formalnom parametru. Na primjer, u prethodnom primjeru, formalni parametar “*r*” je tipa “**double**”, isto kao i stvarni parametar “*poluprecnik*”. Neslaganje u tipu parametara je dozvoljeno jedino u slučajevima u kojima je podržana *automatska konverzija tipa* iz tipa stvarnog parametra u tip formalnog parametra. Na primjer, ako je formalni parametar tipa “**double**”, stvarni parametar može biti tipa “**int**”, s obzirom da postoji automatska konverzija (promotivne prirode) iz tipa “**int**” u tip “**double**”. Načelno je moguće i obrnuto, tj. proslijediti stvarni parametar tipa “**double**” potprogramu koji posjeduje formalni parametar tipa “**int**”, s obzirom da je također podržana i automatska konverzija iz tipa “**double**” u tip “**int**”. Međutim, moramo biti svjesni da će u ovom slučaju doći do *odsječanja decimala* prilikom prenosa stvarnog parametra u formalni.

Potprogram “StampajPrazneLinije” u sljedećem primjeru ispisuje “n” praznih linija, gdje je “n”

formalni parametar:

```
void StampajPrazneLinije(int n) {
    for(int i = 1; i <= n; i++) cout << endl;
}
```

Potprogram možemo kasnije pozvati bilo gdje u programu. Na primjer, kada nam zatreba 5 praznih linija, možemo napisati:

```
StampajPrazneLinije(5);
```

Opisani način prenosa parametara pri kojem se *vrijednost* stvarnog parametra kopira u odgovarajući formalni parametar, naziva se *prenos po vrijednosti* (engl. *passing by value*). Kasnije ćemo vidjeti da jezik C++ podržava i drugi način prenosa parametara, nazvan *prenos po referenci* (engl. *passing by reference*), mada u jeziku C++ ova razlika u prenosu nije toliko striktne forme koliko u nekim drugim jezicima kao što je Pascal. Strogo rečeno, C++ zapravo i ne podržava prenos po referenci, ali se on može simulirati tako što se kao formalni parametar upotrijebi specijalna vrsta objekata nazvana *referenca*. O ovome ćemo govoriti u kasnijim poglavljima.

Pored termina *parametar*, koristi se i termin *argument*. Mada su termini parametar i argument sinonimi, u literaturi se veoma često kada se upotrijebi termin “parametar” bez konkretnе specifikacije da li se radi o formalnom ili stvarnom parametru češće misli na *formalni parametar*, dok je pri upotrebi termina *argument* obrnuto (tj. češće se misli na *stvarni argument*). Na primjer, dosta često se govori o “parametrima koje potprogram prima”, odnosno o “argumentima koje prosljeđujemo potprogramu”.

Formalni i stvarni parametar u principu mogu imati *ista imena*, ali treba voditi računa da se i u tom slučaju radi o *različitim objektima*, a da su njihova istovjetna imena samo stvar slučaja. Dakle, čak i ukoliko formalni i stvarni parametar slučajno imaju isto ime (npr. “n”), formalni parametar “n” je neovisan od stvarnog parametra “n”, mada se pri pozivu potprograma “IspisiPozdrav” stvarni parametar “n” *kopira* u formalni parametar “n”. Ovo je ilustrirano u sljedećem (potpuno legalnom) primjeru, koji predstavlja modificirani program za ispis pozdrava zadani broj puta, uz tehniku prenosa parametara:

```
#include <iostream>
using namespace std;
void IspisiPozdrav(int n) {
    for(int i = 1; i <= n; i++) cout << "Pozdrav!\n";
}
int main() {
    int n;
    cout << "Koliko puta želiš pozdrav? ";
    cin >> n;
    IspisiPozdrav(n);
    return 0;
}
```

Ovdje se prilikom poziva potprograma “IspisiPozdrav” *stvarni parametar* “n” (koji je lokalna promjenljiva potprograma “main”) kopira u *formalni parametar* “n” (koji je lokalna promjenljiva potprograma “IspisiPozdrav”). Sljedeći primjer će nas uvjeriti da stvarni i formalni parametri predstavljaju različite objekte (iako se stvarni kopira u formalni) čak i kada imaju isto ime:

```
#include <iostream>
```

```

using namespace std;

void Potprogram(int n) {
    cout << n;
    n += 3;
    cout << n;
}

int main() {
    int n(5);
    cout << n;
    Potprogram(n);
    cout << n;
    return 0;
}

```

Svoje rezonovanje provjerite tako što ćete odgovoriti na pitanje šta ispisuje ovaj program. Ispravan odgovor je “5585”. Također, radi provjere da li ste shvatili činjenicu da su formalni i stvarni parametri neovisni objekti probajte analizirati šta će ispisati sljedeći (prilično konfuzan) program. Ispravan odgovor je “255225”;

```

#include <iostream>

using namespace std;

void Potprogram(int a, int b) {
    cout << a << b;
}

int main() {
    int a(2), b(5);
    cout << a << b;
    Potprogram(b, a);
    cout << a << b;
    return 0;
}

```

Prenos parametara po vrijednosti može se koristiti kada god želimo prenijeti neku informaciju u potprogram, pri čemu nas dalje ne zanima šta će taj potprogram uraditi sa prenesenom informacijom (tj. da li će prenesena informacija pretrpiti izmjene ili ne; to je privatna stvar potprograma). Osnovna svrha parametara je da učine potprogram generalnijim, tako da se on lakše može ponovo iskoristiti u drugim programima. Na primjer, u programu za ispis šahovske table imali smo potprogram nazvan “StampajZvjezdice” koji je stampao “sirina_polja” zvjezdica na ekranu, pri čemu je “sirina_polja” bila globalna promjenljiva:

```

void StampajZvjezdice() { // Štampa "sirina_polja" zvjezdica
    for(int i = 1; i <= sirina_polja ; i++) cout << "*";
}

```

Ovaj potprogram može biti poboljšan, uvođenjem formalnog parametra (nazvanog npr. “brzv”, skraćeno od “broj zvjezdica”) koji omogućava da potprogram ne mora ništa “znati” o tome kakva se imena promjenljivih koriste u ostatku programa:

```

void StampajZvjezdice(int brzv) { // Štampa "brzv" zvjezdica
    for(int i = 1; i <= brzv; i++) cout << "*";
}

```

Ovim je potprogram postao mnogo generalniji, jer omogućava štampanje onoliko zvjezdica koliko želimo, kada god to zatražimo. Pri tome, željeni broj zvjezdica jednostavno zadajemo prilikom poziva potprograma. Na primjer,

```
StampajZvjezdice(5);
```

Primijetimo da pored toga što potprogram "StampajZvjezdice" ne mora znati kako se zovu promjenljive u ostatku programa, onaj ko poziva potprogram (glavni program u našem slučaju) također ne mora ništa da zna o tome kako se zove formalni parametar potprograma (to je "privatna stvar" potprograma).

U istom programu postoji također i potprogram nazvan "StampajRazmake" koji štampa nekoliko razmaka. Njegov kôd identičan je kao u potprogramu "StampajZvjezdice", izuzev što na objekat "cout" šaljemo razmak, a ne zvjezdicu. Dodajući još jedan parametar, možemo postići da isti potprogram radi oba posla:

```
// Štampa "brzn" znakova "znak"
void StampajZnakove(int brzn, char znak) {
    for(int i = 1; i <= brzn; i++) cout << znak;
}
```

Ovo je još generalniji potprogram, koji štampa proizvoljan broj bilo kojeg znaka koji želimo. Na primjer, pozivom

```
StampajZnakove(4, 'A');
```

odštampaćemo četiri slova 'A'. Isto tako, pozivom

```
StampajZnakove(sirina_polja, '*');
```

odštampaćemo onoliko zvjezdica koliko iznosi trenutna vrijednost promjenljive "sirina_polja".

Primijetimo da se, za slučaj kada potprogram ima više parametara, i stvari i formalni parametri razdvajaju *zarezom*. Pri tome se pri navođenju *formalnih parametara* tip svakog od njih mora navesti odvojeno, kao u sljedećem primjeru programa:

```
#include <iostream>
using namespace std;
void IspisiZbir(int p, int q) {
    cout << p + q;
}
int main() {
    IspisiZbir(3, 2);
    return 0;
}
```

Ovaj program će, naravno, ispisati broj "5". S druge strane, sljedeća definicija nije ispravna:

```
void IspisiZbir(int p, q) {
    cout << p + q;
```

```
}
```

Kada koristimo prototipove potprograma koji imaju parametre, njihova imena *nije neophodno* navoditi i u prototipu, jer je kompjajleru dovoljno da zna *njihov broj i tip* u trenutku kada se potprogram poziva. Tako je sljedeći program *potpuno korektan*:

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

int main(void) {
    void IspisiZbir(int, int);
    IspisiZbir(3, 2);
    return 0;
}

void IspisiZbir(int p, int q) {
    cout << p + q;
}
```

Nije greška navesti imena parametara *i u prototipu*, tako da bi u prethodnom programu sasvim legalan bio i prototip

```
void IspisiZbir(int p, int q);
```

U suštini, kako su imena formalnih parametara potpuno nebitna kada se radi o prototipu, kompjajler ih potpuno ignorira. Kao posljedica te činjenice, imena parametara u prototipu potprograma i u stvarnoj definiciji potprograma *uopće se ne moraju slagati*. Na primjer, u prethodnom programu bio bi bez ikakvih problema prihvачen i prototip poput sljedećeg (bez obzira što se u definiciji potprograma formalni parametri zovu “p” i “q”):

```
void IspisiZbir(int prvi_sabirak, int drugi_sabirak);
```

Opisanu osobinu programeri često koriste, dajući mnogo deskriptivnija imena parametrima u prototipu nego u samoj realizaciji potprograma. Naime, ukoliko je prototip dovoljno deskriptivan, njegovim se posmatranjem često može zaključiti šta potprogram radi, bez potrebe za analizom same realizacije potprograma. S druge strane, upotreba isuviše dugačkih naziva parametara u samom potprogramu bila bi prilično zamorna.

Sljedeći primjer ilustrira potprogram “StampajTablicuMnozenja” sa dva parametra “m” i “n”, koji štampa tablicu množenja za sve proizvode oblika $m \times i$ pri čemu i uzima vrijednosti od 1 do n. Na primjer, ako pozovemo ovaj potprogram pozivom “StampajTablicuMnozenja(3, 5)”, biće ispisana tablica množenja za proizvode $3 \times 1, 3 \times 2, 3 \times 3, 3 \times 4$ i 3×5 :

```
void StampajTablicuMnozenja(int m, int n) {
    for(int i = 1; i <= n; i++)
        cout << m << " x " << i << " = " << m * i << endl;
}
```

Parametri mogu biti *bilo kojeg legalnog tipa* (uključujući i *nizovne tipove*, o čemu ćemo govoriti nešto kasnije). U sljedećem primjeru koristimo parametar koji je *pobrojanog tipa*. Definiran je pobrojani tip “Dani”, i potprogram “StampajKalendar” sa dva parametra. Prvi parametar “broj_dana” je tipa “int”, dok je drugi parametar “pocetni_dan” tipa “Dani”. Parametar “broj_dana” određuje broj dana u mjesecu, parametar “pocetni_dan” određuje dan u sedmici kojim započinje taj mjesec, a potprogram “StampajKalendar” štampa kalendar za taj mjesec. U programu je definiran i glavni program koji

ilustrira kako se poziva taj potprogram za slučaj kada želimo odštampati kalendar za mjesec koji ima 31 dan, a počinje srijedom:

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

using namespace std;

enum Dani { Ponedjeljak, Utorka, Srijeda, Cetvrtak, Petak,
            Subota, Nedjelja};

void StampajKalendar(int broj_dana, Dani pocetni_dan) {
    cout << " P U S Č P S N\n"
        << setw(3 * pocetni_dan) << "";
    for(int j = 1; j <= broj_dana; j++) {
        cout << setw(3) << j;
        if(pocetni_dan != Nedjelja)
            pocetni_dan = Dani(pocetni_dan + 1);
        else {
            pocetni_dan = Ponedjeljak;
            cout << endl;
        }
    }
}

int main() {
    StampajKalendar(31, Srijeda);
    return 0;
}
```

Kao efekat izvršavanja ovog programa dobićemo sljedeći ispis:

P	U	S	Č	P	S	N
1	2	3	4	5		
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

U ovom programu, uz pomoć “`setw`” manipulatora ispisujemo određeni broj praznih mesta, da bismo doveli poziciju za ispis ispod slova koje predstavlja odgovarajući dan. Kako je svaka kolona široka 3 znaka, broj dodatnih praznina koje treba ispisati jednak je trostrukoj vrijednosti rednog broja odgovarajućeg dana (redni brojevi dana počinju od nule, što nam upravo odgovara, jer za slučaj da je početni dan ponедјелjak, nikakvi dopunski razmaci nisu potrebni). Nakon toga, “`for`” petljom ispisujemo sve brojeve redom od 1 do vrijednosti parametra “`broj_dana`”, ažurirajući pri tom svaki put promjenljivu “`pocetni_dan`” tako da ukazuje na sljedeći dan, osim za slučaj kada je njena vrijednost “`Nedjelja`”. U tom slučaju, vrijednost promjenljive “`pocetni_dan`” vraćamo nazad na vrijednost “`Ponedjeljak`”, uz ispis jednog praznog reda, čime zapravo prelazimo na ispis novog reda kalendar-a.

U prethodnom primjeru, deklarirali smo tip “`Dani`” sa *globalnom vidljivošću*, samim tim što smo ga deklarirali *izvan svih blokova*. Stoga je ovaj tip dostupan i u potprogramu “`StampajKalendar`” i u glavnoj funkciji “`main`”, što nam je upravo i potrebno. Da smo tip “`Dani`” deklarirali *unutar potprograma* “`StampajKalendar`”, on ne bi bio vidljiv nigdje izvan tijela tog potprograma, što znači da pobrojana konstanta “`Srijeda`” ne bi bila vidljiva u “`main`” funkciji (tako da program ne bi uopće radio). Treba napomenuti da se, za razliku od deklaracije globalnih promjenljivih, deklaracija tipova sa globalnom vidljivošću *ne smatra štetnom*, i često je veoma potrebna. Naime, dok se informacije mogu razmjenjivati

između potprograma putem prenosa parametara, ne postoji sličan način kojim bi potprogrami mogli međusobno razmjenjivati *tipove*, stoga je upotreba tipova sa globalnom vidljivošću često jedino rješenje.

Veoma je važno napomenuti da iako jezik C++ garantira da će vrijednosti svih stvarnih parametara biti izračunate prije nego što njihove vrijednosti budu proslijedene formalnim parametrima (odnosno prije početka izvršavanja potprograma), jezik C++ ne propisuje *redoslijed kojim će stvarni parametri biti izračunati*. Mada redoslijed izračunavanja stvarnih parametara u većini slučajeva uopće nije bitan, on može biti značajan u slučaju kada stvarni parametri predstavljaju izraze koji sadrže *bočne efekte*. Na primjer, razmotrimo sljedeći jednostavni potprogram koji prosto ispisuje vrijednosti svojih formalnih parametara:

```
void Potprogram(int a, int b) {
    cout << a << " " << b << endl;
}
```

Prepostavimo sada da smo izvršili sljedeći poziv:

```
int x(5);
Potprogram(x++, x++);
```

Kakav će biti ispis nakon izvršavanja ovog potprograma? Odgovor uopće nije propisan standardom jezika C++, odnosno ispis se ne može predvidjeti! Jedino što je garantirano je da će oba izraza “`x++`” biti izvršena, tako da će na kraju promjenjiva “`x`” sigurno imati vrijednost 7. Međutim, nije definirano da li će prvo biti izračunat *lijevi* ili *desni* izraz “`x++`”. Ukoliko se prvo izračuna lijevi izraz, u formalni parametar “`a`” će biti prenesena vrijednost 5 (ne zaboravimo da je vrijednost izraza “`x++`” vrijednost promjenljive “`x`” prije uvećanja), a u parametar “`b`” vrijednost 6, tako da će ispis biti “5 6”. Međutim, ukoliko se prvo izračuna desni izraz, dobićemo ispis “6 5”. Ne smijemo pomisliti da problem nastaje isključivo zbog upotrebe dva bočna efekta u istoj naredbi (što se svakako smatra nedozvoljenim). Naime, podjednako je problematičan i sljedeći poziv:

```
int x(5);
Potprogram(x++, x);
```

Naime, nije teško provjeriti da bi, u zavisnosti od redoslijeda izvršavanja, mogli dobiti ispis “5 6” ili “5 5”. Slične probleme bismo imali i u sljedećem pozivu, u kojem bismo, zavisno od redoslijeda izračunavanja stvarnih parametara, mogli dobiti ispis “4 6” ili “7 4”:

```
int x(5);
Potprogram(x = 4, x + 2);
```

Zbog spomenutih nedoumica, standardom jezika C++ je *zabranjeno* u nekom od stvarnih parametara koristiti promjenljivu nad kojom se u nekom drugom stvarnom parametru istog potprograma obavlja bočni efekat. Ova zabrana *nije sintaksne prirode*, tako da će kompjajler *dopustiti* ovakve pozive, ali njihov efekat nije predvidljiv. Naravno da je standard jezika C++ *mogao* propisati redoslijed izračunavanja stvarnih parametara (npr. slijeva nadesno). Međutim, komitet za standardizaciju je smatrao da je nametanje redoslijeda nepotrebno ograničenje za autore kompjajlera, s obzirom da postoje računarske arhitekture kod kojih je izračunavanje slijeva nadesno *efikasnije* od izračunavanja zdesna naljevo, dok postoje i računarske arhitekture kod kojih vrijedi obrnuto. Nemojte pokušavati da utvrdite koju strategiju koristi Vaš kompjajler, pa da se ubuduće oslanjate na utvrđenu strategiju. Na prvom mjestu, tako napisan program ne mora raditi ukoliko ga probate prevesti nekim drugim kompjajlerom. Što je još gore, po standardu kompjajleri imaju puno pravo da u zavisnosti od okolnosti izaberu redoslijed izračunavanja parametara. Drugim riječima, ukoliko ste eksperimentiranjem utvrdili da kompjajler koji koristite izračunava parametre zdesna naljevo, ne mora značiti da on to radi *uvijek*. Naime, mnogi kompjajleri mogu promijeniti svoj

ustaljeni redoslijed izračunavanja parametara ukoliko u nekoj konkretnoj situaciji zaključe da će promjena redoslijeda dovesti do efikasnijeg prevedenog kôda!

U jeziku C++ je podržana mogućnost da se prilikom pozivanja potprograma navede *manji broj stvarnih parametara* nego što iznosi broj formalnih parametara. Međutim, to je moguće samo ukoliko se u definiciji potprograma na neki način naznači *kakve će početne vrijednosti dobiti* oni formalni parametri koji nisu inicijalizirani odgovarajućim stvarnim parametrom, s obzirom da formalni parametri *uvijek moraju biti inicijalizirani*. Da bismo pokazali kako se ovo postiže, razmotrimo sljedeći potprogram sa tri parametra, koji na ekranu iscrtava pravougaonik od znakova sa visinom i širinom koje se zadaju kao prva dva parametra, dok treći parametar predstavlja znak koji će se koristiti za iscrtavanje pravougaonika:

```
void CrtajPravougaonik(int visina, int sirina, char znak) {
    for(int i = 1; i <= visina; i++) {
        for(int j = 1; j <= sirina; j++) cout << znak;
        cout << endl;
    }
}
```

Ukoliko sada, na primjer, želimo iscrtati pravougaonik formata 5×8 sastavljen od zvjezdica, koristićemo sljedeći poziv:

```
CrtajPravougaonik(5, 8, '*');
```

Prepostavimo sada da u većini slučajeva želimo za iscrtavanje pravougaonika koristiti zvjezdnicu, dok neki drugi znak želimo koristiti samo u iznimnim slučajevima. Tada stalno navođenje zvjezdice kao trećeg parametra pri pozivu potprograma možemo izbjegći ukoliko formalni parametar "znak" proglašimo za *podrazumijevani* (engl. *default*) *parametar* (mada je preciznije reći *parametar sa podrazumijevanom početnom vrijednošću*). To se postiže tako što se iza imena formalnog parametra navede znak "=" iza kojeg slijedi vrijednost koja će biti iskorištena za inicijalizaciju formalnog parametra *u slučaju da se odgovarajući stvarni parametar izostavi*. Slijedi modificirana verzija potprograma "CrtajPravougaonik" koja koristi ovu ideju:

```
void CrtajPravougaonik(int visina, int sirina, char znak = '*') {
    for(int i = 1; i <= visina; i++) {
        for(int j = 1; j <= sirina; j++) cout << znak;
        cout << endl;
    }
}
```

Sa ovakvom definicijom potprograma, poziv poput

```
CrtajPravougaonik(5, 8);
```

postaje sasvim legalan, bez obzira što je broj stvarnih argumenata manji od broja formalnih argumenata. Naime, u ovom slučaju formalni parametar "znak" ima podrazumijevanu vrijednost '*', koja će biti iskorištena za njegovu inicijalizaciju, u slučaju da se odgovarajući stvarni parametar izostavi. Stoga će prilikom navedenog poziva, formalni parametar "znak" dobiti vrijednost '*', tako da će taj poziv proizvesti isti efekat kao i poziv

```
CrtajPravougaonik(5, 8, '*');
```

Treba napomenuti da se podrazumijevana vrijednost formalnog parametra koristi za njegovu inicijalizaciju *samo u slučaju da se izostavi odgovarajući stvarni argument* prilikom poziva potprograma.

Tako će, ukoliko izvršimo poziv

```
CrtajPravougaonik(5, 8, '0');
```

formalni parametar "znak" dobiti vrijednost stvarnog parametra '0', odnosno dobijemo pravougaonik iscrtan od znakova '0'.

Moguće je imati i više parametara sa podrazumijevanom vrijednošću. Međutim, pri tome postoji ograničenje da ukoliko neki parametar ima podrazumijevanu vrijednost, svi parametri koji se u listi formalnih parametara nalaze desno od njega *moraju također imati podrazumijevane vrijednosti* (razloge za ovo ograničenje uvidjećemo uskoro). Odavde slijedi da u slučaju da samo jedan parametar ima podrazumijevanu vrijednost, to može biti samo *posljednji parametar* u listi formalnih parametara. Sljedeći primjer ilustrira varijantu potprograma "pravougaonik" u kojem se javljaju dva parametra sa podrazumijevanim vrijednostima:

```
void CrtajPravougaonik(int visina, int sirina = 10, char znak = '*') {
    for(int i = 1; i <= visina; i++) {
        for(int j = 1; j <= sirina; j++) cout << znak;
        cout << endl;
    }
}
```

Stoga ovaj potprogram možemo pozvati sa *tri, dva ili jednim* stvarnim argumentom, na primjer:

```
CrtajPravougaonik(5, 8, '+');
CrtajPravougaonik(5, 8);
CrtajPravougaonik(5);
```

Posljednja dva poziva ekvivalentna su pozivima

```
CrtajPravougaonik(5, 8, '*');
CrtajPravougaonik(5, 10, '*');
```

Moguće je i da svi parametri imaju podrazumijevane vrijednosti. Takav potprogram je moguće pozvati i bez navođenja ijednog stvarnog argumenta (pri tome se zgrade, koje označavaju poziv potprograma, ne smiju izostaviti, već samo ostaju prazne, kao u slučaju potprograma bez parametara).

Važno je napomenuti da se pri zadavanju podrazumijevanih vrijednosti mora koristiti sintaksa sa znakom "=", a ne konstruktorska sintaksa sa zagradama, koja je dozvoljena (i preporučena) pri običnoj inicijalizaciji promjenljivih. Stoga se zaglavljje prethodnog programa nije moglo napisati ovako:

```
void CrtajPravougaonik(int visina, int sirina(10), char znak('*'))
```

U slučaju da se koriste prototipovi, eventualne podrazumijevane vrijednosti parametara navode se *samo u prototipu*, ali ne i u definiciji potprograma, inače će kompjuter prijaviti grešku (kao i pri svakoj drugoj dvostrukoj definiciji). Za prethodni potprogram, prototip bi mogao izgledati na primjer ovako.

```
void CrtajPravougaonik(int visina, int sirina = 10, char znak = '*');
```

S obzirom da se imena parametara u prototipovima ignoriraju, i mogu se izostaviti, sasvim je legalan i sljedeći prototip (koji djeluje pomalo čudno, jer izgleda kao da se tipovima dodjeljuju vrijednosti):

```
void CrtajPravougaonik(int, int = 10, char = '*');
```

Teoretski, podrazumijevane vrijednosti je moguće definirati u definiciji potprograma a ne u prototipu.

Međutim, u tom slučaju ukoliko se pri pozivu potprograma izostavi odgovarajući stvarni argument, kompjajler će prijaviti grešku ukoliko nije prethodno vidio čitavu definiciju potprograma, jer u suprotnom neće znati da odgovarajući argument uopće ima podrazumijevanu vrijednost.

Mogućnost da više od jednog parametra ima podrazumijevane vrijednosti osnovni je razlog zbog kojeg nije dozvoljeno da bilo koji parametar ima podrazumijevane vrijednosti, nego samo parametri koji čine završni dio liste formalnih parametara. Naime, razmotrimo šta bi se desilo kada bi bio dozvoljen potprogram poput sljedećeg, u kojem *prvi* i *treći* parametar imaju podrazumijevane vrijednosti:

```
void OvoNeRadi(int a = 1, int b, int c = 2) {
    cout << a << " " << b << " " << c << endl;
}
```

Ovakav potprogram bi se očigledno mogao pozvati sa tri, dva ili jednim stvarnim argumentom. Pri tome su pozivi sa tri ili jednim argumentom posve nedvosmisleni. Međutim, u slučaju poziva sa dva stvarna argumenta (odnosno, u slučaju kada je jedan od argumenata izostavljen), javlja se dvosmislica po pitanju *koji* je argument izostavljen (prvi ili treći). Još veće dvosmislice mogle bi nastati u slučaju još većeg broja parametara, od kojih neki imaju podrazumijevane vrijednosti, a neki ne. U jeziku C++ ovakve dvosmislice su otklonjene striktnim ograničavanjem koji parametri mogu imati podrazumijevane vrijednosti, a koji ne mogu.

Principijelno, podrazumijevane vrijednosti ne moraju biti konstante već mogu biti i izrazi. Međutim, zbog ograničenja vidljivosti, eventualne promjenljive u ovim izrazima mogu biti samo globalne promjenljive, čija se primjena svakako ne preporučuje. Stoga su podrazumijevane vrijednosti gotovo uvijek konstantne veličine. Nažalost, kako vidokrug formalnih parametara počinje tek unutar tijela odgovarajućeg potprograma, nije moguće u podrazumijevanoj vrijednosti iskoristiti vrijednost nekog drugog parametra koji se nalazi u listi formalnih parametara, čak i ukoliko se on nalazi *lijeko* od parametra kojem zadajemo vrijednost. Na primer, nije moguće napisati potprogram sa zagлавljem poput sljedećeg, u kojem bi podrazumijevana vrijednost formalnog parametra "sirina" trebala da bude jednaka vrijednosti formalnog parametra "visina":

```
void CrtajPravougaonik(int visina, int sirina = visina, char znak = '*')
```

Uskoro ćemo vidjeti da postoji *drugi način* da se ostvari efekat koji smo željeli postići ovom (neispravnom) definicijom.

U jeziku C++ je dozvoljeno imati više potprograma sa *istim imenima*, pod uvjetom da je iz načina kako je potprogram pozvan moguće *nedvosmisleno odrediti* koji potprogram treba pozvati. Neophodan uvjet za to je da se potprogrami koji imaju ista imena moraju razlikovati ili po broju parametara, ili po tipu odgovarajućih formalnih parametara, ili i po jednom i po drugom. Ova mogućnost naziva se *preklapanje* ili *preopterećivanje* (engl. *overloading*) potprograma (funkcija). Na primjer, u sljedećem primjeru imamo preklopljena dva potprograma istog imena "P1" koji ne rade ništa korisno (služe samo kao demonstracija preklapanja):

```
void P1(int a) {
    cout << "Jedan parametar: " << a << endl;
}

void P1(int a, int b) {
    cout << "Dva parametra: " << a << " i " << b << endl;
}
```

U ovom slučaju se radi o *preklapanju po broju parametara*. Stoga su legalna oba sljedeća poziva (pri

čemu će u prvom pozivu biti pozvan drugi potprogram, sa dva parametra, a u drugom pozivu prvi potprogram, sa jednim parametrom):

```
P1(3, 5);  
P1(3);
```

Sljedeći primjer demonstrira preklapanje *po tipu parametara*. Oba potprograma imaju isto ime “P2” i oba imaju jedan formalni parametar, ali im se tip formalnog parametra razlikuje:

```
void P2(int a) {  
    cout << "Parametar tipa int: " << a << endl;  
}  
  
void P2(double a) {  
    cout << "Parametar tipa double: " << a << endl;  
}
```

Jasno je da će od sljedeća četiri poziva, uz pretpostavku da je “n” cijelobrojna promjenljiva, prva dva poziva dovesti do poziva prvog potprograma, dok će treći i četvrti poziv dovesti do poziva drugog potprograma:

```
P2(3);  
P2(1 + n / 2);  
P2(3.);  
P2(3.14 * n / 2.21);
```

Ovi primjeri jasno ukazuju na značaj pojma *tipa vrijednosti* u jeziku C++, i potrebe za razlikovanjem podatka “3” (koji je tipa “int”) i podatka “3.” (koji je tipa “double”).

Prilikom određivanja koji će potprogram biti pozvan, kompjuter prvo pokušava da pronađe potprogram kod kojeg postoji *potpuno slaganje* po broju i tipu između formalnih i stvarnih parametara. Ukoliko se takav potprogram ne pronađe, tada se pokušava ustanoviti *indirektno slaganje* po tipu parametara, odnosno slaganje po tipu uz pretpostavku da se izvrši automatska pretvorba stvarnih parametara u navedene tipove formalnih parametara (uz pretpostavku da su takve automatske pretvorbe dozvoljene, poput pretvorbe iz tipa “char” u tip “int”). Ukoliko se ni nakon toga ne uspije uspostaviti slaganje, prijavljuje se greška.

U slučaju da se potpuno slaganje ne pronađe, a da se indirektno slaganje može uspostaviti sa *više različitih potprograma*, daje se prioritet slaganju koje zahtijeva “logičniju” odnosno “manje drastičnu” konverziju. Tako se konverzija iz jednog cijelobrojnog tipa u drugi (npr. iz tipa “char” u tip “double”) ili iz jednog realnog tipa u drugi (npr. iz “float” u “double”) smatra “logičnjom” odnosno “direktnijom” od konverzije iz cijelobrojnog u realni tip. Stoga će, za slučaj prethodnog primjera, poziv

```
P2('A');
```

u kojem je stvarni parametar tipa “char”, dovesti do poziva potprograma “P2” sa formalnim parametrom tipa “int”, s obzirom da je konverzija iz tipa “char” u tip “int” neposrednija nego (također dozvoljena) konverzija u tip “double”. Također, konverzije u ugrađene tipove podataka smatraju se logičnjim od konverzija u korisničke tipove podataka, koje ćemo upoznati kasnije. Međutim, može se desiti da se indirektno slaganje može uspostaviti sa *više različitih potprograma*, preko konverzija koje su međusobno *podjednako logične*. U tom slučaju smatra se da je poziv *nejasan* (engl. *ambiguous*), i prijavljuje se greška. Na primjer, ukoliko postoje dva potprograma istog imena od kojih jedan prima parametar tipa “float” a drugi parametar tipa “double”, biće prijavljena greška ukoliko kao stvarni parametar

upotrijebimo podatak tipa “**int**” (osim ukoliko postoji i treći potprogram istog tipa koji prima parametar cjelobrojnog tipa). Naime, obje moguće konverzije iz tipa “**int**” u tipove “**float**” i “**double**” podjednako su logične, i kompjuter ne može odlučiti koji potprogram treba pozvati. Na ovakve nejasnoće već smo ukazivali pri opisu matematičkih funkcija iz biblioteke “**cmath**”, kod kojih nastaju upravo opisane nejasnoće u slučaju da im se kao stvarni argumenti proslijede cjelobrojne vrijednosti.

Uz izvjestan oprez, moguće je miješati tehniku korištenja podrazumijevanih parametara, preklapanja po broju parametara i preklapanja po tipu parametara. Oprez je potreban zbog činjenice da se kombiniranjem ovih tehnika povećava mogućnost da nepažnjom formiramo definicije koje će dovesti do nejasnih poziva. Na primjer, razmotrimo sljedeća dva potprograma istog imena “P3”:

```
void P3(int a) {
    cout << "Jedan parametar: " << a << endl;
}

void P3(int a, int b = 10) {
    cout << "Dva parametra: " << a << " i " << b << endl;
}
```

Jasno je da je, uz ovako definirane potprograme, poziv poput “P3(10)” nejasan, jer ne postoji mogućnost razgraničenja da li se radi o pozivu prvog potprograma, ili pozivu drugog potprograma sa izostavljenim drugim argumentom. U oba slučaja ostvaruje se potpuno slaganje tipova. Međutim, uz neophodnu dozu opreza, moguće je formirati korisne potprograme koji kombiniraju opisane tehnike. Na primjer, razmotrimo sljedeće potprograme:

```
void CrtajPravougaonik(int visina, int sirina, char znak = '*') {
    for(int i = 1; i <= visina; i++) {
        for(int j = 1; j <= sirina; j++) cout << znak;
        cout << endl;
    }
}

void CrtajPravougaonik(int visina, char znak = '*') {
    for(int i = 1; i <= visina; i++) {
        for(int j = 1; j <= visina; j++) cout << znak;
        cout << endl;
    }
}
```

Prvi od ova dva potprograma je već razmotreni potprogram za crtanje pravougaonika, a drugi potprogram je njegova neznatno modificirana varijanta (bolje rečeno, specijalni slučaj) koji iscrtava pravougaonik sa jednakom širinom i visinom (tj. kvadrat). Sa ovako definiranim potprogramima mogući su pozivi poput sljedećih, pri čemu se u prva dva slučaja poziva prvi potprogram, a u trećem i četvrtom slučaju drugi potprogram:

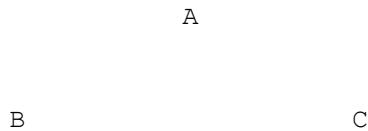
```
CrtajPravougaonik(5, 10, '+');
CrtajPravougaonik(5, 10);
CrtajPravougaonik(5, '+');
CrtajPravougaonik(5);
```

Naročito je interesantno razmotriti drugi i treći poziv. Iako oba poziva imaju po dva stvarna argumenta, drugi poziv poziva prvi potprogram, jer se sa prvim potprogramom ostvaruje potpuno slaganje tipova stvarnih i formalnih argumenata uz pretpostavku da je treći argument izostavljen. S druge strane, treći poziv poziva drugi potprogram, jer se potpuno slaganje tipova stvarnih i formalnih argumenata ostvaruje sa drugim potprogramom, dok je sa prvim potprogramom moguće ostvariti samo indirektno

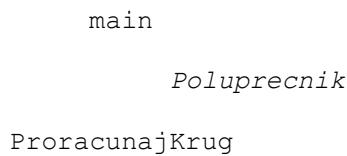
slaganje, u slučaju da se izvrši konverzija tipa “**char**” u tip “**int**”.

Iz izloženog je jasno da se preklapanje potprograma može koristiti kao alternativa korištenju podrazumijevanih vrijednosti parametara, i da je pomoću preklapanja moguće postići efekti koji nisu izvodljivi upotrebom podrazumijevanih vrijednosti. Međutim, treba obratiti pažnju da se kod preklapanja ne radi o *jednom*, nego o *više različitih potprograma*, koji samo imaju isto ime. Ovi potprogrami koji dijele isto ime trebali bi da obavljaju slične zadatke, inače se postavlja pitanje zašto bi uopće koristili isto ime za potprograme koji rade različite stvari. Inače, prije nego što se odlučimo za preklapanje, treba razmisliti da li je zaista potrebno korištenje istog imena, jer prevelika upotreba preklapanja može dovesti do zbrke. Na primjer, u prethodnom primjeru možda je pametnije drugi potprogram nazvati “*CrtajKvadrat*”, jer on u suštini zaista iscrtava *kvadrat* (koji doduše jeste specijalan slučaj pravougaonika, tako da i ime “*CrtajPravougaonik*” ima opravdanja, u smislu da se drugi potprogram može smatrati kao specijalan slučaj prvog). Naročito su sporna preklapanja po tipu parametara, jer je dosta upitno zbog čega bi trebali imati dva potprograma istog imena koji prihvataju argumente različitog tipa. Ukoliko ovi potprogrami rade različite stvari, trebali bi imati i različita imena. Stoga se preklapanje po tipu obično koristi u slučaju kada više potprograma logički gledano obavlja isti zadatak, ali se taj zadatak za različite tipove izvodi na različite načine (npr. postupak računanja stepena a^b osjetno se razlikuje za slučaj kada je b cijeli broj i za slučaj kada je b realan). Sa primjerima ispravno upotrijebljenih preklapanja susretaćemo se u kasnijim poglavljima. Kao opću preporuku, u slučajevima kada se isti efekat može postići preklapanjem i upotrebom parametara sa podrazumijevanim vrijednostima, uvijek je bolje i sigurnije koristiti parametre sa podrazumijevanim vrijednostima. Preklapanje može biti veoma korisno, ali samo pod uvjetom da tačno znamo šta i zašto radimo. U suprotnom, upotreba preklapanja samo dovodi do zbrke.

Kod programa koji sadrže mnoštvo potprograma, neophodno je poznavati *strukturu programa*, odnosno informaciju o tome koje potprograme on sadrži, i kakva je *pozivna hijerarhija* (tj. koji potprogrami zovu koje potprograme). Pregledan način za prikazivanje strukture programa predstavljaju tzv. *strukturni dijagrami*. Na primjer, sljedeći dijagram označava da potprogram “A” poziva potprograme “B” i “C”:



Strukturni dijagrami mogu također prikazivati i *prenos parametara*. Na primjer, sljedeći dijagram prikazuje da glavni program poziva potprogram “krug” i da mu pri tome prenosi vrijednost promjenljive “poluprecnik” kao parametar:



Ne treba miješati strukturne dijagrame sa tzv. *dijagramima toka* (engl. *flowcharts*). Naime, strukturni dijagrami prikazuju samo hijerarhiju potprograma, a ne i algoritam kako pojedini potprogrami djeluju (niti kako funkcioniraju).

Mehanizam prenosa parametara je od ključnog značaja za razvoj programa. Naime, u dobro

napisanom programu, niti jedan dio programa *ne bi trebao da ima pristup onim promjenljivim koje mu nisu potrebne* (ovaj princip postaje još izražajniji u tzv. *objektno zasnovanom pristupu programiranju*, koji ćemo razmatrati kasnije). Na primjer, sa gledišta onog ko poziva potprogram, svaki potprogram treba djelovati kao "crna kutija" (pod ovim pojmom se u tehnici obično podrazumijeva uređaj za koji možemo utvrditi *šta* radi, ali ne i *kako* radi, s obzirom da nam je njegova unutrašnjost posve nedostupna). Naime, onaj ko poziva potprogram treba samo da potprogramu predigne spravne parametre i da im prepusti da odrade svoj posao, *ne ulazeći u to kako će oni to uraditi*. S druge strane, potprogrami samo primaju parametre od pozivaoca, i ne tiče ih se šta radi ostatak programa. Oni se brinu *samo kako da obave zadatok koji im je povjeren*. Također, u dobro napisanom programu, ni jedan potprogram ne bi trebao da radi više različitih poslova (svakom poslu treba dodijeliti poseban potprogram).

Mehanizam koji obezbjeđuje ispunjenje ovih principa naziva se *sakrivanje informacija* (engl. *information hiding*). U jeziku C++ ovaj mehanizam se ostvaruje tako što svaku promjenljivu definiramo tako da joj je vidokrug što je god moguće manji. Da se izbjegne upotreba globalnih promjenljivih u funkcijama, treba *intenzivno koristiti parametre*. Ovaj koncept je ilustriran na poboljšanoj verziji modularnog programa za prikaz šahovske tabele, u kojem je izbjegnuta upotreba globalnih promjenljivih, korištenjem mehanizma prenosa parametara:

```
#include <iostream>
using namespace std;
const char Razmak(' '), Zvjezdica('*');

int main() { // Glavni program
    void StampajTablu(int, int); // Prototip potprograma "StampajTablu"
    int m; // Širina kvadratnog polja
    int n; // Visina kvadratnog polja

    cout << "Ovaj program prikazuje šahovsku tablu.\n\n"
        "Unesite širinu svakog kvadrata, u znakovima: ";
    cin >> m;
    cout << "Unesite visinu svakog kvadrata, u linijama: ";
    cin >> n;
    cout << "\n\n";
    StampajTablu(n, m);
    return 0;
}

// Štampa šahovsku tablu
void StampajTablu(int visina, int sirina) {
    void StampajNeparniRed(int, int);
    void StampajParniRed(int, int);
    for(int i = 1; i <= 4; i++) {
        StampajNeparniRed(visina, sirina);
        StampajParniRed(visina, sirina);
    }
}

// Štampa neparni red
void StampajNeparniRed(int visina, int sirina) {
    void StampajLinijuNeparnogReda(int);
    for(int i = 1; i <= visina; i++) StampajLinijuNeparnogReda(sirina);
}

// Štampa parni red
void StampajParniRed(int visina, int sirina) {
```

```

void StampajLinijuParnogReda(int) ;
    for(int i = 1; i <= visina; i++) StampajLinijuParnogReda(sirina);
}
// Štampa liniju neparnog reda
void StampajLinijuNeparnogReda(int sirina_kvadrata) {
    void StampajZnakove(int, char);
    for(int i = 1; i <= 4; i++) {
        StampajZnakove(sirina_kvadrata, Razmak);
        StampajZnakove(sirina_kvadrata, Zvjezdica);
    }
    cout << endl;
}

// Štampa liniju parnog reda
void StampajLinijuParnogReda(int sirina_kvadrata) {
    void StampajZnakove(int, char);
    for(int i = 1; i <= 4; i++) {
        StampajZnakove (sirina_kvadrata, Zvjezdica);
        StampajZnakove (sirina_kvadrata, Razmak);
    }
    cout << endl;
}

// Štampa niz znakova
void StampajZnakove(int broj_znakova, char znak) {
    for(int i = 1; i <= broj_znakova; i++) cout << znak;
}

```

Slijedi i kompletan strukturni dijagram za ovaj program:



Primijetimo da strukturni dijagram *ne opisuje* korišteni algoritam, što je već ranije istaknuto.

Algoritmi za svaku funkciju prikazanu na struktturnom dijagramu moraju biti opisani korištenjem *pseudokôda* (način koji smo već u više navrata koristili za opis funkcioniranja pojedinih dijelova programa). U današnje vrijeme se izbjegava opisivanje algoritama pomoću *dijagrama toka* (engl. *flowcharts*), koji su se nekada intenzivno koristili za njihovo opisivanje, jer se smatra da dijagrami toka nisu prilagođeni konceptima modernih programskega jezika, i da njihova upotreba podstiče nemodularni pristup u razvoju programa.

Na ovom mjestu je neophodno naglasiti da modularni programi nisu niti najkraći niti najefikasniji, ali su sigurno lakši i za razumijevanje i za održavanje (tj. za eventualne modifikacije, korekcije, dopune itd.) od odgovarajućih nemodularnih programa. Na primjer, slijedeći (nemodularni) program sigurno je dosta kraći od prethodno napisanog modularnog programa, ali je teži za razumijevanje (koristi četiri petlje jedna unutar druge), teži je za izmjene, a naročito je loša strana što se ni jedan dio ovog programa ne može upotrebiti kao potprogram u nekom drugom programu:

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

int main() {
    int m, n;
    cout << "Ovaj program prikazuje šahovsku tablu.\n\n"
        "Unesite širinu svakog kvadrata, u znakovima: ";
    cin >> m;
    cout << "Unesite visinu svakog kvadrata, u linijama: ";
    cin >> n;
    cout << "\n\n";
    for(int i = 1; i <= 4; i++) {
        for(int j = 1; j <= n; j++) {
            for(int k = 1; k <= 4; k++) {
                for(int l = 1; l <= m; l++) cout << " ";
                for(int l = 1; l <= m; l++) cout << "*";
            }
            cout << endl;
        }
        for(int j = 1; j <= n; j++) {
            for(int k = 1; k <= 4; k++) {
                for(int l = 1; l <= m; l++) cout << "*";
                for(int l = 1; l <= m; l++) cout << " ";
            }
            cout << endl;
        }
    }
    return 0;
}
```

Upotreba malih uloženih petlji koje koriste promjenljivu “*l*” kao brojač mogla bi se izbjegći korištenjem manipulatora “*setw*” i “*setfill*”. Ova modifikacija ostavlja se čitatelju odnosno čitateljki za vježbu.

Treba naglasiti da su *efikasnost*, *kratkoća* i *modularnost* tri *potpuno oprečna zahtjeva*. U vrijeme kada su računari bili spori i kada je kapacitet radne memorije bio mali, efikasnost i kratkoća su bili dominantni zahtjevi. Danas, kada je od svih računarskih “komponenti” ubjedljivo najskuplji radni sat programera, primarni zahtjev je modularnost. Pokazuje se da je, uz dobar kompjajler, dužina izvršne verzije (tj. nakon kompjajliranja) modularnog programa tek neznatno veća od dužine ekvivalentnog nemodularnog programa, a razlika u efikasnosti je skoro neprimjetna. U kasnijim poglavljima ćemo se upoznati sa principima *objektno zasnovanog* i *objektno orijentiranog dizajna* koji su još pogodniji za razvoj velikih

programa (sa aspekta jasnoće, razumijevanja i mogućnosti održavanja), mada su sa aspekta kratkoće i efikasnosti programa ovi principi još nepovoljniji. Bez obzira na to, danas se gotovo svi veći programi projektiraju koristeći ove principe.

Osobina modularnih programa koja omogućava da se jednom napisani potprogrami mogu upotrebljavati u drugim programima dovodi do velike uštede u vremenu prilikom razvoja obimnih programskih paketa. S druge strane, skraćivanje programa obično zahtijeva oslanjanje na neke “trikove” specifične za problem koji se rješava, što čini razrađeni algoritam potpuno “vezanim” za problem koji se rješava. Pogledajmo, na primjer, sljedeću verziju programa za isrtavanje šahovske tabele. On je nesumnjivo izuzetno kratak, i koristi samo dvije petlje, ali i prilično težak za razumijevanje, jer se zasniva na nekim “karakterističnim” svojstvima “šare” koju čini šahovska tabla. Također, ovaj program koristi i prilično “konfuzni” ternarni operator “? :”. Nemojte se previše zabrinjavati ukoliko ne uspijete razumjeti kako radi ovaj program. On je više “mozgalica” za enigmatičare nego primjer kako bi trebalo pisati dobre programe:

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

int main() {
    int m, n;
    cout << "Ovaj program prikazuje šahovsku tablu.\n\n"
        "Unesite širinu svakog kvadrata, u znakovima: ";
    cin >> m;
    cout << "Unesite visinu svakog kvadrata, u linijama: ";
    cin >> n;
    cout << "\n\n";
    for(int i = 0; i < 8 * n; i++) {
        for(int k = 0; k < 8 * m; k++)
            cout << ((i / n + k / m) % 2 ? "*" : " ");
        cout << endl;
    }
    return 0;
}
```

Na primjeru ovog programa također možemo vidjeti i da su kratkoća i efikasnost međusobno sukobljeni zahtjevi. Tako je ovaj program vjerovatno “šampion kratkoće”, ali je manje efikasan od prethodnog, jer se u ovom programu unutar unutrašnje petlje izvršavaju tri operacije dijeljenja. Kako se unutrašnja petlja izvršava $8 \cdot m$ puta a spoljašnja $8 \cdot n$ puta, slijedi da se u ovom programu izvršava ukupno $192 \cdot m \cdot n$ dijeljenja (koje je relativno spora operacija), dok se u prethodnom programu ne izvršava niti jedno dijeljenje! Efikasnost ovog programa se može osjetno popraviti (uz zadržavanje kratkoće) upotrebom manipulatora “setw” i “setfill”, što prepustamo enigmatski nastrojenim čitateljima i čitateljkama kao korisnu vježbu.

16. Funkcije koje vraćaju vrijednost

Mana svih dosad napisanih potprograma je u tome što oni ne omogućavaju da se ikakav rezultat iz potprograma *vrati nazad* na mjesto njegovog poziva. Šta se pod ovim misli, vidjećemo iz sljedećeg razmatranja. Već smo vidjeli da C++ poznaje funkciju “`sqrt`” (definiranu u biblioteci “`cmath`”) koja računa kvadratni korijen svog argumenta, i da tu funkciju možemo koristiti u konstrukcijama poput sljedećih (naravno, uz pretpostavku da su propisno deklarirane odgovarajuće promjenljive koje se u ovim primjerima spominju):

```
korijen = sqrt(broj);
hipotenuza = sqrt(katetal * kateta1 + kateta2 * kateta2);
stranica = dijagonala / sqrt(2);
cout << sqrt(3);
```

Odavde vidimo da funkcija “`sqrt`” *ima svoju vrijednost* (koja je jednaka kvadratnom korijenu njenog argumenta), koja joj omogućava da se ona *može koristiti unutar izraza*. Pokušajmo sada da sami napravimo funkciju koja kao rezultat vraća kvadrat broja koji je zadan kao argument. Sa dosadašnjim znanjem, sve što možemo da uradimo je da napravimo “funkciju” koja na ekran *ispisuje* kvadrat svog argumenta, konstrukcijom poput:

```
void Kvadrat(double x) {
    cout << x * x;
}
```

Iako ovu funkciju možemo upotrebiti da ispišemo kvadrat nekog broja, pozivima poput

```
Kvadrat(3);
Kvadrat(2 * a + 5);
```

i slično, ovaku funkciju možemo upotrebiti samo *samu za sebe*, a ne i unutar nekog izraza. Razlog je u tome što je ovako definirana funkcija *objekat bez vrijednosti*, tj. ona ne vraća nikakvu vrijednost nazad na mjesto poziva koja bi mogla da bude upotrebljena unutar nekog izraza. Zbog toga su sljedeće konstrukcije besmislene, i kompjuter će prijaviti grešku pri pokušaju da napišemo nešto slično:

```
c = Kvadrat(a + b);
cout << Kvadrat(5);
cout << Kvadrat(a) + Kvadrat(b);
hipotenuza = sqrt(Kvadrat(katetal) + Kvadrat(kateta2));
```

Iz istog razloga, potpuno su besmislene i konstrukcije poput

```
a = StampajTablu(m, n);
b = StampajTablicuMnozenja(3, 5);
```

gdje su “`StampajTablu`” i “`StampajTablicuMnozenja`” funkcije (potprogrami) koje su navedene kao primjeri u prethodnom poglavlju. Da bismo omogućili da se funkcija može koristiti unutar izraza, moramo napisati *funkciju koja vraća neku vrijednost*. Definicija ovakve funkcije je slična definiciji običnog potprograma (funkcije) koji ne vraća vrijednost, osim što se umjesto riječi “`void`” na početku piše *tip vrijednosti (rezultata) koji funkcija vraća*, koji često nazivamo i *povratni tip* (engl. *return type*). Rezervirana riječ “`void`” zapravo ukazuje na *odsustvo povrante vrijednosti*. Sljedeća definicija definira funkciju “`Kvadrat`” koja vraća vrijednost tipa “`double`”:

```

double Kvadrat(double x) {
    return x * x;
}

```

Barem jedna naredba (najčešće posljednja) unutar funkcije koja vraća vrijednost mora biti naredba “**return**”, pomoću koje se vraća vrijednost iz funkcije. Po nailasku na ovu naredbu, izvršavanje tijela funkcije se prekida, pri čemu se vrijednost izraza navedenog iza ključne riječi “**return**” vraća kao rezultat funkcije na mjesto odakle je funkcija pozvana. Program se dalje nastavlja izvršavati kao da je čitav poziv funkcije (zajedno sa njenim argumentima) prosto zamijenjen vrijednošću koja je vraćena iz funkcije. Na primjer, ovako definiranu funkciju “Kvadrat” bez problema možemo koristiti unutar izraza, tako da su potpuno legalne sljedeće naredbe:

```

cout << Kvadrat(5);
c = Kvadrat(a);
cout << "Zbir kvadrata brojeva 3 i 4 je " << Kvadrat(3) + Kvadrat(4);
hipotenuza = sqrt(Kvadrat(katetal) + Kvadrat(kateta2));

```

Izraz naveden iza ključne riječi “**return**” mora se slagati po tipu sa navedenim povratnim tipom funkcije, osim u slučajevima u kojima je podržana automatska konverzija tipova iz jednog tipa u drugi. Funkcija može vratiti rezultat *bilo kojeg tipa osim nizovnog tipa*. U slučaju da ne upotrijebimo naredbu “**return**” unutar funkcije koja vraća vrijednost, vraćena vrijednost će biti *nedefinirana* (odnosno *slučajna*, ili bolje rečeno *nepredvidljiva*) što sigurno nije poželjno. Ova situacija smatra se greškom. Stoga mnogi kompjajleri (ali nažalost ne svi) javljaju grešku u slučaju da se unutar funkcije koja vraća vrijednosti nigdje ne upotrijebi naredba “**return**”.

Možemo primijetiti da su funkcije koje vraćaju vrijednost znatno bliže *pojmu funkcije u matematskom smislu*, za razliku od funkcija koje ne vraćaju vrijednost. Zbog toga smo, pri razmatranju funkcija koje ne vraćaju vrijednost, izbjegavali upotrebu termina “funkcija”, nego smo koristili općenitiji pojam “potprogram”. S druge strane, funkcija “Kvadrat” ne radi ništa drugo osim što vraća vrijednost nazad, pri čemu se onaj ko je pozvao funkciju brine o tome šta će biti urađeno sa vraćenom vrijednošću (tj. sama funkcija se o tome ne brine). Dakle, ova funkcija, *sama za sebe ne radi ništa* što bi ostavilo nekog traga, tako da sljedeća naredba, iako principijelno nije zabranjena, ne radi ništa smisleno:

```
Kvadrat(5);
```

Ovom naredbom zatražili smo da se izračuna koliki je kvadrat od 5, i funkcija će ga zaista izračunati. Međutim, kako nismo ništa rekli šta treba uraditi sa izračunatom vrijednošću (niti je ispisujemo, niti je dodjeljujemo nečemu, niti je koristimo unutar nekog izraza) ona se prosto *ignorira*. Ova konstrukcija je podjednako besmislena kao da smo napisali naredbu poput

```
sqrt(3);
```

ili naredbu poput

```
2 + 3;
```

Iz izloženog razmatranja se može izvući zaključak da funkcije koje *vraćaju vrijednost* obično imaju smisla *samo ako se upotrebe unutar nekog izraza*. Doduše, ovo ne treba shvatiti kao izričito pravilo, jer da je uvijek tako, C++ ne bi ni dozvoljavao da se funkcija koja vraća vrijednost upotrijebi samostalno, izvan izraza (takva je situacija recimo u programskom jeziku Pascal). Naime, u rijetkim slučajevima, kada funkcija pored toga što vraća vrijednost radi i nešto drugo što može ostaviti neki vidljivi trag ili proizvesti efekat koji utiče na ostatak programa, tada ima smisla pozivati funkciju koja vraća vrijednost samostalno

(tj. ne unutar izraza). Ovo su ipak relativno specifični slučajevi koje ćemo komentirati onog trenutka kada na njih naiđemo. Također, kako funkcije koje vraćaju vrijednost nemaju tako jak *imperativni karakter* (karakter naredbe) kakav imaju potprogrami koji vraćaju vrijednost, pri njihovom imenovanju ne moramo se čvrsto držati konvencije da im imena trebaju imati glagolsku prirodu.

U sljedećem primjeru naveden je ponovo program koji računa i ispisuje obim i površinu kruga, samo što se u njemu koriste i funkcije koje vraćaju vrijednost (razmotrite zašto je u ovom primjeru dobro da je konstanta “PI” definirana sa globalnom vidljivošću):

```
#include <iostream>

using namespace std;

const double PI(3.141592654);

double Obim(double r) { // Računa obim kruga
    return 2 * PI * r;
}

double Povrsina(double r) { // Računa površinu kruga
    return PI * r * r;
}

void Proracunajkrug(double r) { // Štampa obim i površinu kruga
    cout << "Obim: " << Obim(r) << endl;
    cout << "Površina: " << Povrsina(r) << endl;
}

int main() { // Glavni program
    double poluprecnik;
    cin >> poluprecnik;
    ProracunajKrug(poluprecnik);
    return 0;
}
```

Funkcije “Obim” i “Povrsina” imaju imena koja nisu u glagolskoj formi. Ukoliko bismo ipak insistirali na glagolskim formama, prihvatljiva imena mogu biti “RacunajObim” i “RacunajPovrsinu”.

Funkcije koje vraćaju vrijednost često su kratke, i nekad se njihovo tijelo sastoji samo od “**return**” naredbe (takve funkcije pogodno je realizirati kao tzv. *umetnute funkcije*, o kojima ćemo govoriti kasnije). Na primjer, u sljedećem primjeru definirana je funkcija “Kub” koja kao rezultat vraća kub broja zadanog kao parametar, kao i funkcija “DaLiJePrestupna” koja kao rezultat vraća logičku vrijednost “**true**” ili “**false**” zavisno da li je godina zadana kao parametar prestupna ili ne. Napomenimo da je po trenutno važećem Gregorijanskom kalendaru godina prestupna ako je djeljiva sa 4, osim ukoliko je djeljiva sa 100 a nije istovremeno djeljiva sa 400. Dakle, svaka četvrta godina nije uvijek prestupna, nego se u razdoblju od 400 godina tri puta pojavi razmak od 8 godina između dvije prestupne godine:

```
double Kub(double x) {
    return x * x * x;
}

bool DaLiJePrestupna(int godina) {
    return (godina % 4 == 0) && (godina % 100 != 0 || godina % 400 == 0);
}
```

Ovako definirane funkcije mogu se koristiti u konstrukcijama poput:

```
cout << 5 + Kub(3.7) / 2.5;
```

```
if(DaLiJePrestupna(ova_godina)) broj_dana[Februar] = 29;
```

Razumije se da se tijelo funkcije ne mora se sastojati od samo jedne naredbe. Tijelo funkcije može biti onoliko složeno koliko je potrebno da se izračuna vrijednost koja će biti vraćena iz funkcije. U sljedećem primjeru definirana je funkcija "Faktorijel" koja računa faktorijel broja datog kao parametar (što je najlakše uraditi primjenom "for" petlje). Ova funkcija je iskorištena u testnom programu ("main" funkciji) koji napisanu funkciju koristi za izračunavanje binomnih koeficijenata " n nad k ", koji se mogu izraziti pomoću faktorijela upotrebom sljedeće formule:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

```
#include <iostream>

using namespace std;

long int Faktorijel(int n) {
    long int p(1);
    for(int i = 1; i <= n; i++) p *= i;
    return p;
}

int main() {
    int n, k;
    cout << "n = ";
    cin >> n;
    cout << "k = ";
    cin >> k;
    cout << "n nad k = "
        << Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
    return 0;
}
```

Nema nikavog razloga da jednom definiranu funkciju ne upotrijebimo unutar druge funkcije, kao u sljedećem primjeru u kojem funkcija "n_nad_k" poziva funkciju "Faktorijel":

```
#include <iostream>

using namespace std;

int n_nad_k(int n, int k) {
    long int Faktorijel(int);
    return Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
}

long int Faktorijel(int n) {
    long int p(1);
    for(int i = 1; i <= n; i++) p *= i;
    return p;
}

int main() {
    int n, k;
    cout << "n = ";
    cin >> n;
    cout << "k = ";
    cin >> k;
    cout << "n nad k = " << n_nad_k(n, k);
    return 0;
}
```

}

U funkciji “n_nad_k” morali smo navesti prototip funkcije “Faktorijel”, s obzirom da smo funkciju “Faktorijel” definirali tek *nakon* njenog poziva unutar funkcije “n_nad_k”.

Primijetimo da upotrijebljena definicija binomnog koeficijenta (preko svođenja na faktorijel) *nije najpodesnija* za računanje na računaru. Na primjer, neka je $n = 50$ i $k = 2$. Mada je “ n nad k ” u ovom slučaju sasvim mali broj (1225), računar ga *neće moći izračunati*, jer će pri računanju faktorijela od 50 doći do prekoračenja (to je broj od preko 60 cifara). Ovo na jasan način ilustrira činjenicu da mnoge matematičke formule, koje su u principu posve tačne, mogu biti potpuno neprimjenljive za svrhe programiranja, jer ne uzimaju u obzir ograničenja koja postoje u računarima pri reprezentaciji brojčanih podataka u računarskoj memoriji. Stoga je često potrebno primjenjivati zaobilazna rješenja, koja ne koriste neposredno matematičke definicije. Razmislite sami kako biste mogli napisati funkciju “n_nad_k” koja ne koristi faktorijel, i u kojoj se ne bi javljao ovakav problem!

Mnogi od programa koji smo ranije pisali mogu se učiniti mnogo fleksibilnijim upotrebom funkcija. Na primjer, sljedeći program definira funkciju “NZD” koja računa najveći zajednički djelilac svoja dva argumenta, a zatim koristi definiranu funkciju za računanje najmanjeg zajedničkog sadržioca (NZS) dva broja unesena sa tastature koristeći činjenicu da je $\text{NZS}(p, q) = p \cdot q / \text{NZD}(p, q)$:

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

int NZD(int p, int q) {
    int ostatak;
    do {
        ostatak = p % q; p = q; q = ostatak;
    } while(ostatak);
    return p;
}

int main() {
    int a, b;
    cout << "Unesite dva broja: ";
    cin >> a >> b;
    cout << "Njihov NZS je " << a * b / NZD(a, b) << endl;
    return 0;
}
```

Ilustrativan je i sljedeći primjer. Iz numeričke matematike je poznato da se određeni integral neke funkcije $f(x)$ može približno izračunati uz pomoć Simpsonovog pravila, prema kojem je:

$$\int_a^b f(x) dx \approx \frac{h}{3} \left[f(a) + 4 \cdot \sum_{(i=1,3,5...)}^{n-1} f(a + i \cdot h) + 2 \cdot \sum_{(i=2,4,6...)}^{n-2} f(a + i \cdot h) + f(b) \right]$$

gdje je n broj podintervala na koji dijelimo interval (a, b) , koji mora biti *paran* (veći broj podintervala daje veću tačnost računanja), a h je dužina svakog podintervala (tj. $h = (b - a) / n$). U sljedećem programu je definirana funkcija “SimpsonIntegral” koja računa integral funkcije $f(x)$ na intervalu (a, b) koristeći n podjela, pri čemu je za testnu funkciju uzeta funkcija $f(x) = x^3$:

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

```

double F(double x) {
    return x * x * x;
}

double SimpsonIntegral(double a, double b, int n) {
    double h = (b - a) / n, suma = F(a) + F(b);
    for(int i = 1; i < n; i++)
        if(i % 2 != 0) suma += 4 * F(a + i * h);
    else suma += 2 * F(a + i * h);
    return (h / 3) * suma;
}

int main() {
    double a, b;
    int n;
    cout << "Unesi granice: ";
    cin >> a >> b;
    cout << "Broj podjela: ";
    cin >> n;
    cout << "Vrijednost integrala je: " << SimpsonIntegral(a, b, n);
    return 0;
}

```

Funkcija “SimpsonIntegral” se može i optimizirati, po cijenu da je učinimo *teško čitljivom*, kao na primjer u sljedećoj izvedbi:

```

double SimpsonIntegral(double a, double b, int n) {
    double h = (b - a) / n, suma = F(a) + F(b);
    for(int i = 1; i < n; i++) suma += 2 * (i % 2) * F(a + i * h);
    return h * suma / 3;
}

```

Naredba “**return**” se, unutar tijela funkcije, po potrebi može javiti i više od jedanput, kao u sljedeća dva primjera koji definiraju funkciju “Sgn” koja predstavlja “signum” funkciju (koja kao rezultat vraća 1, 0 ili -1 ovisno o znaku argumenta), i funkciju “Minimum” koja vraća kao rezultat najmanji od njena tri realna argumenta:

```

int Sgn(double x) {                                // Određuje znak broja
    if(x > 0) return 1;
    else if(x == 0) return 0;
    else return -1;
}

// Vraća najmanji od realnih brojeva "a", "b" i "c" kao rezultat
double Minimum(double a, double b, double c) {
    if(a <= b && a <= c) return a;
    else if(b < c) return b;
    else return c;
}

```

Funkcija “Minimum” predstavlja dobar primjer u kojem ima smisla koristiti preklapanje funkcija *po tipu* parametara. Naime, ova funkcija je predviđena za rad sa *realnim* parametrima, mada bi mogla biti sasvim upotrebljiva i za cjelobrojne parametre. Zapravo, ona već onako kako je napisana *radi* i sa cjelobrojnim parametrima, ali ne baš posve efikasno. Razmotrimo, na primjer, šta se dešava prilikom sljedećeg poziva:

```
int a(3), b(5), c(4), min;
```

```
min = Minimum(a, b, c);
```

U ovom primjeru se *cjelobrojne* vrijednosti stvarnih parametara “a”, “b” i “c” prvo konvertiraju u *realne* vrijednosti prije nego što se proslijede u istoimene *realne* formalne parametre. Nakon toga, sva poređenja unutar tijela funkcije obavljaju se nad *realnim* vrijednostima. Vrijednost koja se vraća iz funkcije je također *realnog* tipa koja se konvertira u *cjelobrojnu* vrijednost (odsjecanjem decimala) prilikom pridruživanja vraćene vrijednosti promjenljivoj “*min*”. Doduše, ovo odsjecanje neće dovesti do gubitka decimala, s obzirom da vraćena vrijednost svakako ne sadrži decimalne (s obzirom da je vraćena vrijednost jednaka jednoj od vrijednosti formalnih parametara “a”, “b” i “c”, koji ne sadrže decimalne zbog činjenice da su inicijalizirani vrijednostima koje su nastale konverzijom cjelobrojnih vrijednosti). Bez obzira na to, izvršavanje ove funkcije nad cjelobrojnim argumentima je *neefikasno*, jer se nepotrebno vrše četiri konverzije (tri iz cjelobrojnog tipa u realni tip i jedna iz realnog tipa u cjelobrojni), i sva poređenja se vrše nad realnim vrijednostima (ne zaboravimo da svaka manipulacija sa realnim vrijednostima zahtijeva mnogo više vremena nego istovjetna manipulacija sa cjelobrojnim vrijednostima). Zbog toga ima smisla napisati funkciju koja obavlja *isti postupak*, ali prihvata cjelobrojne argumente i vraća cjelobrojnu vrijednost:

```
// Vraća najmanji od cijelih brojeva "a", "b" i "c" kao rezultat
int Minimum(int a, int b, int c) {
    if(a <= b && a <= c) return a;
    else if(b < c) return b;
    else return c;
}
```

U ovom slučaju, ukoliko pozovemo funkciju “Minimum” sa cjelobrojnim argumentima, biće pozvana verzija funkcije koja prihvata cjelobrojne argumente (i koja vraća cjelobrojni rezultat), dok će u svim ostalim slučajevima biti pozvana verzija funkcije koja prihvata realne argumente. Ovo uključuje i slučajeve kada su neki od stvarnih argumenata cjelobrojni, a neki realni. Naime, mada su moguće dvostrukе konverzije između cjelobrojnih i realnih tipova, konverzije iz cjelobrojnog tipa u realni smatraju se logičnijim, jer pri njima ne dolazi do gubitka informacija. Stoga će se pri takvom pozivu vrijednosti stvarnih parametara koje nisu cjelobrojne konvertirati u realne (a ne obrnuto), i biće pozvana verzija funkcije koja prihvata realne parametre, što je uostalom i logično.

U prethodnom primjeru imali smo dvije preklopljene funkcije čije je tijelo *potpuno identično*. Kasnije ćemo vidjeti da se ovakva preklapanja elegantnije izvode pomoću tzv. *šablona* (engl. *templates*). Međutim, često se dešava da postoje *efikasniji načini* da se neki postupak izvede za neke specijalne tipove podataka nego za opći slučaj. Razmotrimo, na primjer, kako bismo mogli realizirati funkciju koja računa vrijednost stepena x^y (zanemarimo činjenicu da takva funkcija, pod nazivom “*pow*”, već postoji u biblioteci “*cmath*”). U općem slučaju, stepenovanje proizvoljne baze proizvoljnim eksponentom možemo svesti na stepenovanje sa bazom e uz pomoć logaritamske funkcije, jer vrijedi formula $x^y = e^{y \ln x}$ (strog uvezvi, ova formula je tačna samo za $x > 0$). Ova formula dovodi do sljedeće definicije:

```
double Stepen(double x, double y) {
    return exp(y * log(x));
}
```

Mada je ova formula tačna i za *cjelobrojne* vrijednosti eksponenta, ona računanje stepena svodi na veoma složena računanja komplikiranih funkcija “*exp*” i “*log*”, stoga je ovakva realizacija veoma neefikasna za slučaj kada je eksponent cijeli broj. Naime, u slučaju kada je eksponent cijeli broj, vrijednost stepena je moguće jednostavnije izračunati ponovljenim množenjem, koje se veoma jednostavno realizira pomoću “*for*” petlje (u slučaju kada je eksponent *negativan*, potrebno je i još jedno dijeljenje). Stoga ima smisla napisati i preklopljenu verziju funkcije “*Stepen*” koja će biti pozivana u slučaju kada se kao drugi argument navede cjelobrojna vrijednost:

```

double Stepen(double x, int n) {
    double p(1);
    for(int i = 1; i <= abs(n); i++) p *= x;
    if(n >= 0) return p;
    else return 1 / p;
}

```

U ovom slučaju, obje verzije funkcije “Stepen” obavljaju *isti zadatak* (računanje stepena), ali na *različite načine*, ovisno o tipu drugog parametra. Ovo je tipičan primjer kada zaista ima smisla koristiti preklapanje po tipu argumenata.

Mada je gore prikazana verzija funkcije “Stepen” koja koristi “**for**” petlju gotovo uvijek mnogo efikasnija od verzije koja se oslanja na funkcije “*exp*” i “*log*” (koju moramo koristiti za slučaj kada eksponent nije cjelobrojan), ona je još uvijek dosta neefikasnja za slučaj velikih eksponenata. Na primjer, za slučaj kada je eksponent $n=100$, potrebno je izvršiti 100 množenja, što nije tako mnogo, ali je činjenica da je moguće isti rezultat dobiti pomoću mnogo manje množenja, koristeći algoritam poznat pod nazivom *kvadriraj-i-množi* (engl. *square-and-multiply*). Ovaj algoritam se zasniva na razlaganju eksponenta na stepene broja 2 (što se može uraditi na isti način kao pri pretvaranju broja u binarni brojni sistem). Na primjer, kako vrijedi

$$100 = 2^6 + 2^5 + 2^2 = 64 + 32 + 4$$

to vrijedi i

$$x^{100} = x^{64+32+4} = x^{64} \cdot x^{32} \cdot x^4$$

Slijedi da ukoliko poznamo koliko iznose x^{64} , x^{32} i x^4 , možemo izračunati x^{100} pomoću samo dva množenja. Međutim, ove vrijednosti možemo dobiti uz pomoć svega 6 množenja, koristeći uzastopno kvadriranje:

$$\begin{aligned} x^2 &= x \cdot x, & x^4 &= (x^2)^2 = x^2 \cdot x^2, & x^8 &= (x^4)^2 = x^4 \cdot x^4, & x^{16} &= (x^8)^2 = x^8 \cdot x^8, \\ x^{32} &= (x^{16})^2 = x^{16} \cdot x^{16}, & x^{64} &= (x^{32})^2 = x^{32} \cdot x^{32} \end{aligned}$$

Konačni zaključak je da se stepen x^{100} može izračunati koristeći svega 8 množenja. Ovaj postupak se može generalizirati za svaki cjelobrojni eksponent. Mada na prvi pogled izgleda da je ovaj postupak težak za implementaciju, on na kraju dovodi do posve jednostavne funkcije “Stepen”, koja je, za iole veće vrijednosti eksponenta, osjetno efikasnija nego prethodna verzija koja koristi uzastopno množenje:

```

double Stepen(double x, int n) {
    double p(1), n1 = abs(n);
    while(n1 != 0) {
        if(n1 % 2 == 1) p *= x;
        n1 /= 2; x *= x;
    }
    if(n >= 0) return p;
    else return 1 / p;
}

```

Potrudite se da sami analizirate i razumijete kako ova funkcija radi (najbolje je pratiti tok njenog izvršavanja na konkretnom primjeru). Ideje koje su iskorištene za njenu realizaciju mogu se veoma uspješno iskoristiti za rješavanje mnogih srodnih problema.

Možda iz do sada navedenih primjera nije očigledno da nailazak na naredbu “**return**” prekida izvršavanje tijela funkcije, i vrši trenutačan povratak na mjesto odakle je funkcija pozvana. U sljedećem primjeru ovo svojstvo je iskorišteno u funkciji “**DaLiJeProst**” koja vraća logičku vrijednost “**true**” ili “**false**” u zavisnosti da li je parametar koji joj je proslijeden prost broj ili nije (algoritam na kojem se zasniva ova funkcija već je korišten ranije). Ta funkcija je iskorištena kao potprogram u programu koji ispisuje sve proste brojeve od 1 do 1000.

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

using namespace std;

bool DaLiJeProst(long int n) {
    long int korijen = sqrt(n);
    if(n != 2 && n % 2 == 0) return false;
    else
        for(long int i = 3; i <= korijen; i += 2)
            if(n % i == 0) return false;
    return true;
}

int main() {
    cout << "Prosti brojevi od 1 do 1000:\n";
    for(int i = 1; i <= 1000; i++)
        if(DaLiJeProst(i)) cout << i << " ";
    return 0;
}
```

Naredba “**return**” može se koristiti i unutar funkcija koje ne vraćaju nikakav rezultat, samo se u tom slučaju iza ove naredbe *ne stavlja ništa* (tj. nikakva vrijednost). Dejstvo ove naredbe je *prekidanje izvršavanje tijela funkcije i povratak na mjesto poziva* (ona se može koristiti ukoliko želimo da pod određenim uvjetima prekinemo izvršavanje tijela funkcije prije nego što program nađe na njen kraj, slično naredbi “**break**” koja prekida izvršavanje petlji). Neki programeri uvijek stavljaju naredbu “**return**” neposredno prije kraja funkcije koja ne vraća vrijednost, iako to u principu nije neophodno, jer će nailazak na kraj funkcije svakako uzrokovati povratak na mjesto poziva. Ako se naredba “**return**” upotrebí unutar glavnog programa (tj. unutar funkcije “**main**”), ona dovodi do prekida izvršavanja programa (odnosno povratka u operativni sistem, koji je zapravo pozvao funkciju “**main**”).

Primijetimo da je, u suštini, funkcija “**main**” funkcija koja *vraća vrijednost*. Njen tip povratne vrijednosti mora biti “**int**”, s obzirom da onaj koji je pozvao funkciju “**main**” (tj. operativni sistem) očekuje da ona vrati vrijednost tog tipa. Kako ne možemo utjecati na način kako se “**main**” funkcija poziva iz operativnog sistema, nemamo načina da utičemo na tip povratne vrijednosti koju funkcija “**main**” mora imati. Prirodno je zapitati se *da li funkcija “main” može imati parametre*. Odgovor je *potvrđan* (njihove vrijednosti se tada “**main**” funkciji proslijeduju iz samog operativnog sistema). Međutim, o ovoj mogućnosti nećemo govoriti, jer ona zahtijeva izvjesno poznавање rada samog operativnog sistema, što izlazi izvan okvira o kojima se ovdje govorи.

Bitno je napomenuti da nije moguće imati preklapanje po *tipu povratne vrijednosti*. Na primjer, kompjajler će prijaviti grešku ukoliko probate formirati preklopljene funkcije poput sljedećih:

```
int F(int x) {
    return 3 * x;
}
```

```

double F(int x) {
    return 3.14 * x;
}

```

Ovakvo preklapanje nije moguće iz jednostavnog razloga što u navedenom primjeru kompjuter zaista ne može da zna koju funkciju treba pozvati u slučaju poziva poput “`cout << F(5)`”.

Iz brojnih navedenih primjera funkcija *koje vraćaju vrijednost* ne treba pogrešno zaključiti da bi svaka funkcija trebala da vraća neku vrijednost (da je tako, mogućnost formiranja funkcija bez povratne vrijednosti ne bi uopće ni postojala). Tako, na primjer, funkcije poput funkcija “`IspisiPozdrav`”, “`StampajTablu`” i “`StampajTablicuMnozenja`” koje smo pisali u prethodnom poglavlju teško da mogu vraćati ikakvu smislenu vrijednost!

Treba napomenuti da se iz funkcije može vratiti *samo jedna vrijednost* (iako je vraćanje vrijednosti moguće izvršiti sa više od jednog mesta). Na primjer, nije moguće napraviti funkciju koja istovremeno vraća i zbir i razliku dva broja koji su proslijedeni kao argumenti (mada ćemo kasnije vidjeti da je moguće vratiti kao rezultat tzv. *agregat* koji u sebi sadrži zbir i razliku). Početnici koji ne razumiju smisao *operatora zarez* mogu doći u zabludu da pomisle da je ovako nešto moguće, s obzirom da je sljedeća funkcija, sintaksno posmatrano, sasvim ispravna:

```

int VratiZbirIRazliku(int a, int b) {
    return a + b, a - b;
}

```

Za one koji su shvatili smisao operatora zarez trebalo bi biti jasno da ova funkcija zapravo vraća kao rezultat samo *razliku*. Još više zabune može stvoriti činjenica da će, uz prepostavku da su promjenljive “*zbir*” i “*razlika*” propisno deklarisane, poziv poput sljedećeg

```

zbir, razlika = VratiZbirIRazliku(3, 2);

```

biti sasvim ispravno prihvaćen. Međutim, ovdje je ponovo operator zarez upotrijebljen na pogrešan način. Na osnovu značenja operatora zarez trebalo bi biti jasno da će u ovakovom pozivu vrijednost vraćena iz funkcije “`VratiZbirIRazliku`” biti dodijeljena samo promjenljivoj “*razlika*”, dok promjenljivoj “*zbir*” neće biti dodijeljeno *ništa*. U svakom slučaju treba zapamtiti da funkcija može vratiti samo jednu vrijednost, dok su svi eventualni primjeri koji izgledaju kao da vraćaju više vrijednosti samo *pogrešne interpretacije*, obično uzrokovane neshvatanjem smisla operatora zarez.

Kod potpunih početnika često se javlja podsvjesno mješanje pojmove “vraća vrijednost” i “ispisuje vrijednost”, uslijed kojeg može doći do prilično banalnih grešaka. Naime, kod početnika se često može sresti *posve bespotreban* ispis vrijednosti koje trebaju da budu vraćene kao rezultat iz funkcije, umjesto da ispis bude prepušten onome ko je pozvao funkciju. Na primjer, ukoliko se funkcija “`Kub`” definira na sljedeći način:

```

double Kub(double x) {
    cout << x * x * x;
    return x * x * x;
}

```

tada će, pored činjenice da ova funkcija vraća kao rezultat kub vrijednosti koja je proslijedena kao parametar, svaki poziv funkcije “`kub`” izvršiti ispis ovog rezultata i kada treba i kada ne treba. Tako će, na primjer, sljedeća naredba, pored toga što će promjenljivoj “*c*” dodijeliti vrijednost 125, također *ispisati* broj 125 na ekran, iako se to od ne *ne očekuje*:

```

c = Kub(5);

```

Isto tako, lako možemo vidjeti zbog čega će naredba poput

```
cout << "Kub broja 5 je " << Kub(5);
```

sa ovako definiranom funkcijom “kub” proizvesti besmislen ispis

Kub broja 5 je 125125

Slična greška je korištenje ispisa *umjesto* upotrebe naredbe “**return**”, kao u sljedećoj funkciji:

```
double Kub(double x) {
    cout << x * x * x;
}
```

U ovom slučaju kompjajler može zaključiti da nešto nije regularno, i prijaviti grešku. Nažalost, izvjesni kompjajleri ne prijavljuju grešku u slučaju izostavljanja naredbe “**return**”. U tom slučaju, vrijednost koja će biti vraćena iz funkcije nije predvidljiva, što može dovesti do grešaka analognih greškama koje nastaju zbog upotrebe neinicijaliziranih promjenljivih.

Iz svih navedenih primjera ne treba pogrešno zaključiti da funkcije koje vraćaju vrijednost nikada ne treba da ispisuju ništa na ekran, niti da se ove funkcije uvijek moraju upotrebiti unutar nekog izraza. Čest je slučaj da potprogram treba da obavi određeni posao, pri čemu pozivaoc programa treba da primi povratnu informaciju vezanu za određeni posao. Tu informaciju je najbolje proslijediti putem povratne vrijednosti. Razmotrimo, na primjer, sljedeći potprogram. Ovaj potprogram nalazi i ispisuje rješenja kvadratne jednačine sa koeficijentima koji se prosljeđuju kao parametri. Međutim, pored toga, ovaj program *vraća informaciju* o tome da li je razmatrana jednačina imala realna rješenja ili ne:

```
bool KvadratnaJednacina(double a, double b, double c) {
    double d = b * b - 4 * a * c;
    if(d >= 0) {
        cout << "x1 = " << (-b - sqrt(d)) / (2 * a)
            << "\nx2 = " << (-b + sqrt(d)) / (2 * a) << endl;
        return true;
    }
    double re = -b / (2 * a), im = abs(sqrt(-d)) / (2 * a));
    cout << "x1 = (" << re << "," << -im << ") \n"
        "x2 = (" << re << "," << im << ") \n";
    return false;
}
```

U slučaju da je razmatrana kvadratna jednačina imala realna rješenja, funkcija kao rezultat vraća vrijednost “**false**”, a u suprotnom vraća vrijednost “**true**” (ne treba ove vraćene vrijednosti brkati sa rješenjima kvadratne jednačine, koje ova funkcija ne *vraća*, nego samo *ispisuje* na ekran, stoga bi prikladnije ime ovog potprograma bilo “*IspisiRjesenjaKvadratneJednacine*”, što nismo koristili zbog dužine). Prepostavimo da je pozivaoc ove funkcije pored ispisa rješenja, potrebna i informacija o prirodi rješenja (realna ili kompleksna). U tom slučaju, poziv je moguće izvršiti kao u sljedećoj sekvenci naredbi:

```
double a, b, c;
cout << "Unesi koeficijente: ";
cin >> a >> b >> c;
bool rjesenja_su_realna = KvadratnaJednacina(a, b, c);
if(rjesenja_su_realna) cout << "Rješenja su realna\n";
```

```
else cout << "Rješenja su kompleksna\n";
```

U ovom primjeru, vrijednost vraćena iz funkcije "KvadratnaJednacina" dodijeljena je logičkoj promjenljivoj "rjesenja_su_realna", što nam omogućava da kasnije ispitamo status vraćen iz funkcije. Principijelno je dozvoljena i konstrukcija poput sljedeće:

```
if(KvadratnaJednacina(a, b, c)) cout << "Rješenja su realna\n";
else cout << "Rješenja su kompleksna\n";
```

Ovakva konstrukcija se ne preporučuje, jer je prilično nejasna (iz načina pozivanja teško je zaključiti da će poziv funkcije "KvadratnaJednacina" imati i propratni efekat ispisa rješenja, pogotovo što nismo koristili preporučeno mnogo prikladnije, ali nažalost predugačko ime). Međutim, pretpostavimo da nas zanimaju samo rješenja kvadratne jednačine, ali ne i informacija o prirodi njenih rješenja. U tom slučaju, funkciju "KvadratnaJednacina" možemo pozvati kao da se radi o funkciji koja ne vraća vrijednost, odnosno pozivom poput

```
KvadratnaJednacina(a, b, c);
```

U ovom primjeru imamo funkciju koja *vraća vrijednost*, čiju povratnu vrijednost *ignoriramo*, jer nam nije potrebna. Upravo je ovakva mogućnost osnovni razlog zbog kojeg jezik C++ omogućava ignoriranje povratne vrijednosti. Naime, ponekad su nam potrebni samo sporedni efekti koje proizvodi poziv funkcije, ali ne i sama vraćena vrijednost. Mnoge funkcije u standardnim bibliotekama koje čine sastavni dio jezika C++, pored toga što vraćaju vrijednosti, proizvode i neke sporedne efekte koji su često bitniji od same vraćene vrijednosti. Na primjer, razmotrimo često korištenu funkciju "getch" iz (nestandardne) biblioteke "conio" koju, iako je nestandardna, podržavaju gotovo svi kompjajleri za PC računare. Ova funkcija čeka na pritisak tastera i *vraća kao rezultat* ASCII šifru pritisnutog tastera. Stoga je sasvim moguće napisati sekvencu naredbi poput sljedeće:

```
char znak = getch();
cout << "Upravo ste pritisnuli taster " << znak;
```

Međutim, često se povratna vrijednost funkcije "getch" ignorira, nego se samo koristi njen *propratni efekat* (čekanje na pritisak tastera). Ovu funkciju smo do sada koristili isključivo na taj način.

Može li se desiti da funkcija *nema parametre* a da *vraća vrijednost*? Zašto da ne!? Funkcija "getch" koju smo maločas spomenuli upravo je takva. Slijedi još jedan primjer takve funkcije:

```
int OcitajBroj() {
    int x;
    cout << "Unesite neki broj: ";
    cin >> x;
    return x;
}
```

Ovu funkciju možemo pozvati na sljedeći način (uz pretpostavku da je deklarirana promjenljiva "broj" tipa "**int**":

```
broj = OcitajBroj();
```

Neko bi mogao postaviti pitanje kakva je korist od ovakve funkcije, s obzirom da se isti efekat može postići prostim izrazom "cin >> broj". Postoji više razloga zbog kojeg je ovakva funkcija korisna. Na prvom mjestu, ovakva funkcija dozvoljava pridruživanje vrijednosti unesene sa tastature odmah pri deklaraciji promjenljive, kao u sljedećoj deklaraciji:

```
int broj = OcitajBroj();
```

Ovakav stil je više “u duhu jezika C++” nego klasični unos putem operatora “`>>`”, jer je u jeziku C++ preporučljivo sve promjenljive obavezno inicijalizirati odmah prilikom njihovog definiranja. Međutim, mnogo je važniji razlog činjenica da je u funkciju “`OcitajBroj`” moguće unijeti sve provjere ispravnosti unosa podataka (npr. provjeru da li je zaista unesen broj) i tražiti ponovni unos vrijednosti u slučaju da ona nije ispravna *prije nego što se izvrši povratak iz funkcije*. Na primjer, ukoliko je potrebno unijeti tri vrijednosti “`a`”, “`b`” i “`c`” sa tastature uz kontrolu ispravnosti unosa (uz pretpostavku da su odgovarajuće promjenljive prethodno deklarirane), možemo samo pisati

```
a = OcitajBroj(); b = OcitajBroj(); c = OcitajBroj();
```

Na ovaj način ćemo izbjegići potrebu da postupak kontrole unosa pišemo iznova svaki put pri unosu nove promjenljive sa tastature.

Bitno je napomenuti da ukoliko se u nekom izrazu javlja više poziva funkcija koje vraćaju vrijednost, standard jezika C++ ne propisuje *kojim će se redoslijedom* te funkcije pozivati (isto kao što nije propisan redoslijed izračunavanja stvarnih parametara). Na primjer, neka su “`F`” i “`G`” dvije funkcije koje primaju cjelobrojni argument, a “`x`” neka cjelobrojna promjenljiva. Prilikom izvršavanja izraza

```
x = F(2) + G(3);
```

nije propisano da li će prvo biti pozvana funkcija “`F`” ili funkcija “`G`”. Zapravo, ista stvar vrijedi i prilikom izvršavanja izraza

```
cout << F(2) << " " << G(3);
```

U posljednjem slučaju jedino se garantira da će prvo biti ispisana rezultat funkcije “`F`” a zatim rezultat funkcije “`G`”, ali koji će od ta dva rezultata *biti prije izračunat* nije propisano standardom. U većini slučajeva to zapravo i nije važno, međutim može biti značajno u slučajevima kada funkcija posjeduje propratne efekte. Da bismo ovo ilustrirali, pretpostavimo da imamo sljedeće funkcije (koje ne rade ništa pametno, ali kao propratni efekat ispisuju činjenicu da su pozvane):

```
int F(int x) {
    cout << "Pozvana je funkcija F\n";
    return x + 1;
}

int G(int x) {
    cout << "Pozvana je funkcija G\n";
    return x + 1;
}
```

Ukoliko probamo prethodne izraze u kojima se javljaju pozivi funkcija “`F`” i “`G`” sa ovako definiranim funkcijama “`f`” i “`g`”, nije definirano da li će prvo biti ispisana tekst “Pozvana je funkcija F” ili tekst “Pozvana je funkcija G”, s obzirom da nije definirano koja će funkcija biti prvo pozvana! Slično kao pri redoslijedu izračunavanja argumenata, kompjuter ima puno pravo da organizira redoslijed poziva funkcija unutar istog izraza po vlastitom nahođenju. Zbog toga je potreban priličan oprez prilikom upotrebe funkcija koje posjeduju bočne efekte unutar izraza. Do problema neće doći ukoliko poštujemo ranije navedeno pravilo po kojem je veoma rizično unutar jednog izraza imati više bočnih efekata.

Problemi sa bočnim efektima mogu također nastati pri nepažljivoj upotrebi funkcija koje posjeduju

bočne efekte u izrazima koji sadrže operatore “`&&`” i “`||`”. Naime, izrazi koji sadrže ove operatore izračunavaju se na specifičan način, o kojem smo govorili u poglavlju o logičkim operatorima. Da bismo ilustrirali ove probleme, pretpostavimo da imamo logički izraz poput “`F(x) && G(x)`” u kojem funkcije “`F`” i “`G`” posjeduju bočne efekte. Pretpostavimo dalje da programeru nije bitno da li će prvo biti pozvana funkcija “`F`” ili “`G`”, ali mu je bitno da obje funkcije budu *zaista pozvane* (odnosno da naredbe u njihovom tijelu budu zaista izvršene). Međutim, u navedenom izrazu prvo će biti pozvana funkcija “`F`” (ovo je garantirano), a ukoliko njen rezultat bude vrijednost “`false`” ili nula, funkcija “`G`” *uopće neće biti pozvana!* Naime, kao što je već ranije objašnjeno, u izrazu oblika “`x && y`” podizraz “`y`” se uopće ne izračunava ukoliko se ustanovi da je podizraz “`x`” netačan (za tu svrhu, podizraz “`x`” se mora izračunati prvi). Sličnu situaciju imamo u izrazu oblika “`x || y`” u kojem se podizraz “`y`” ne izračunava ukoliko se ustanovi da je podizraz “`x`” tačan (tj. ima vrijednost “`true`” ili različitu od nule).

Poseban oprez je potreban u slučaju funkcija koje *pamte stanje svog izvršavanja* (što je također jedna specijalna vrsta bočnog efekta). Na primjer, neka je potrebno napraviti funkciju nazvanu “`KumulativnaSuma`” sa jednim parametrom koja vraća kao rezultat ukupnu (kumulativnu) sumu svih dotada zadanih vrijednosti njenih stvarnih argumenata. Na primjer, sljedeća sekvenca naredbi

```
cout << KumulativnaSuma(3) << endl;
cout << KumulativnaSuma(5) << endl;
cout << KumulativnaSuma(2) << endl;
cout << KumulativnaSuma(1) << endl;
cout << KumulativnaSuma(7) << endl;
```

trebala bi da ispiše niz brojeva 3, 8, 10, 11 i 18 ($3+5=8$, $8+2=10$, $10+1=11$, $11+7=18$). Ovakvu funkciju nije teško napraviti tako što ćemo definirati promjenljivu koja pamti sumu dotada zadanih vrijednosti argumenata. Ta promjenljiva naravno mora biti statička, jer treba da se inicijalizira samo pri prvom pozivu funkcije, i da “preživi” njen završetak. Tako dobijamo definiciju poput sljedeće (pomalo neobična konstrukcija “`return suma += n;`” služi kao zamjena za dvije naredbe “`suma += n;`” i “`return suma;`”):

```
int KumulativnaSuma(int n) {
    static int suma(0);
    return suma += n;
}
```

Ovakva funkcija sasvim ispravno radi ukoliko se upotrijebi u sekvenci naredbi poput maloprije navedene sekvence. Međutim, ukoliko umjesto toga napišemo naredbu

```
cout << KumulativnaSuma(3) << endl << KumulativnaSuma(5) << endl
<< KumulativnaSuma(2) << endl << KumulativnaSuma(1) << endl
<< KumulativnaSuma(7) << endl;
```

koja *izgleda* ekvivalentna prethodnoj sekvenci, vjerovatno ćemo dobiti posve neočekivane rezultate. Naime, ova naredba je, u suštini, *jedan izraz* u kojem se pet puta poziva funkcija “`KumulativnaSuma`”. Mada je redoslijed ispisa jasno definiran (slijeva nadesno), redoslijed *poziva* ove funkcije (unutar istog izraza) *nije definiran*, stoga su rezultati nepredvidljivi. Na primjer, ukoliko redoslijed poziva ove funkcije bude izvršen zdesna nalijevo, umjesto očekivane sekvence brojeva dobićemo ispis sekvenice 7, 8, 10, 15 i 18 ($7+1=8$, $8+2=10$, $10+5=15$, $15+3=18$). Ovdje se ponovo radi o upotrebi više bočnih efekata u istom izrazu, mada se ovakva situacija lako može da previdi (mnogi programeri nemaju naviku da naredbu za ispis posmatraju kao izraz)!

17. Reference i prenos parametara po referenci

Mehanizam prenošenja parametara u funkcije koji smo do sada upoznali naziva se *prenos parametara po vrijednosti* (engl. *passing by value*). Ovaj mehanizam omogućava prenošenje vrijednosti sa mesta poziva funkcije u samu funkciju. Međutim, pri tome ne postoji nikakav način da funkcija promijeni vrijednost nekog od stvarnih parametara koji je korišten u pozivu funkcije, s obzirom da funkcija manipulira samo sa formalnim parametrima koji su posve neovisni objekti od stvarnih parametara (mada su inicijalizirani tako da im na početku izvršavanja funkcije vrijednosti budu jednake vrijednosti stvarnih parametara). Slijedi da se putem ovakvog prenosa parametara ne može prenijeti nikakva informacija iz funkcije nazad na mjesto poziva funkcije. Informacija se doduše može prenijeti iz funkcije na mjesto poziva putem povratne vrijednosti, ali postoje situacije gdje to nije dovoljno.

Ograničenja prenosa po vrijednosti lako možemo vidjeti iz sljedećeg razmatranja. Zamislimo da želimo da definiramo funkciju "Povecaj" koja uvećava vrijednost cjelobrojne promjenljive kojoj je proslijeđena kao parametar, tako da sljedeća sekvenca naredbi

```
int a = 5;
Povecaj (a);
cout << a;
```

ispiše broj "6". Postavljeni problem je zapravo principijelne prirode, jer je jasno da nam za nešto ovakvo uopće nije potrebna nikakva funkcija (dovoljno je samo izvršiti izraz "a++"). Međutim, činjenica je da ovaku funkciju *nije moguće napisati* koristeći samo do sada izložene koncepte. Naime, ako napišemo nešto poput

```
void Povecaj (int x) {
    x++;
}
```

nismo postigli ništa korisno. Naime, vrijednost promjenljive "a" će prilikom poziva funkcije "Povecaj" biti prenesena u formalni parametar "x", koji će nakon toga zaista biti povećan za 1, ali se to neće odraziti na sadržaj promjenljive "a". Naime, formalni parametar "x" je posve neovisan objekat od promjenljive "a", koji pored toga biva uništen prilikom završetka funkcije, odnosno prilikom povratka iz funkcije na mjesto poziva. Naravno, ne bi ništa pomoglo ni da formalni parametar ima isto ime kao stvarni parametar, s obzirom da su formalni i stvarni parametri uvijek različiti objekti, bez obzira na to kako se zovu. Ni sljedeća definicija ne nudi mnogo veću korist:

```
int Povecaj (int x) {
    return x + 1;
}
```

Ovako definiranu funkciju bismo doduše mogli iskoristiti za postizanje željenog efekta (povećanje promjenljive "a") upotrebom konstrukcije poput

```
a = Povecaj (a);
```

ali to nije forma poziva kakvu smo tražili. Naime, nama je potreban mehanizam koji omogućava da funkcija *promijeni* svoj *stvarni parametar*.

Izloženi problem se rješava upotrebom tzv. *referenci* (ili *upućivača*, kako se ovaj termin ponekad

prevodi). Reference su specijalni objekti koje je najlakše zamisliti kao *alternativna imena* (engl. *alias names*) za druge objekte. Reference se deklariraju kao i obične promjenljive, samo se prilikom deklaracije ispred njihovog imena stavlja oznaka “&”. Reference se obavezno moraju *inicijalizirati* prilikom deklaracije, bilo upotrebom znaka “=”, bilo upotrebom konstruktorske sintakse (navođenjem inicijalizatora unutar zagrada). Međutim, za razliku od običnih promjenljivih, reference se ne mogu inicijalizirati proizvoljnim izrazom, već samo nekom *drugom promjenljivom* potpuno istog tipa (dakle, konverzije tipova poput konverzije iz tipa “**int**” u tip “**double**” *nisu dozvoljene*) ili, općenitije, nekom *l-vrijednošću* istog tipa (mada su, za sada, promjenljive i individualni elementi niza jedine l-vrijednosti koje poznajemo). Pri tome, referenca postaje *vezana* (engl. *tied*) za promjenljivu (odnosno l-vrijednost) kojom je inicijalizirana u smislu koji ćemo uskoro razjasniti. Na primjer, ukoliko je “**a**” neka cijelobrojna promjenljiva, referencu “**b**” vezanu za promjenljivu “**a**” možemo deklarirati na jedan od sljedeća dva ekvivalentna načina:

```
int &b = a;  
int &b(a);
```

Kada bi “**b**” bila obična promjenljiva a ne referenca, efekat ovakve deklaracije bio bi da bi se promjenljiva “**b**” inicijalizirala trenutnom vrijednošću promjenljive “**a**”, nakon čega bi se ponašala kao posve neovisan objekat od promjenljive “**a**”, odnosno eventualna promjena sadržaja promjenljive “**b**” ni na kakav način ne bi uticala na promjenljivu “**a**”. Međutim, referenca se *potpuno poistovjećuju* sa objektom na koji su vezane. Drugim riječima, “**b**” se ponaša kao *alternativno* ime za promjenljivu “**a**”, odnosno svaka manipulacija sa objektom “**b**” odražava se na identičan način na objekat “**a**” (kaže se da je “**b**” referenca na promjenljivu “**a**”). To možemo vidjeti iz sljedećeg primjera:

```
b = 7;  
cout << a;
```

Ovaj primjer će ispisati broj “7”, bez obzira na eventualni prethodni sadržaj promjenljive “**a**”, odnosno dodjela vrijednosti “7” promjenljivoj “**b**” zapravo je promijenila sadržaj promjenljive “**a**”. Objekti “**a**” i “**b**” ponašaju se kao da se radi o istom objektu! Potrebno je napomenuti da nakon što se referenca jednom veže za neki objekat, ne postoji način da se ona preusmjeri na neki drugi objekat. Zaista, ukoliko je npr. “**c**” također neka cijelobrojna promjenljiva, tada naredba dodjele poput

```
b = c;
```

neće preusmjeriti referencu “**b**” na promjenljivu “**c**”, nego će promjenljivoj “**a**” dodijeliti vrijednost promjenljive “**c**”, s obzirom da je referenca “**b**” i dalje vezana za promjenljivu “**a**”. Svaka referenca čitavo vrijeme svog života uvijek upućuje na objekat za koji je vezana prilikom inicijalizacije. Kako referenca uvijek mora biti vezana za neki objekat, to deklaracija poput

```
int &b;
```

nema nikakvog smisla.

S obzirom da se reference uvijek vežu za konkretan objekat, i ponašaju se kao alternativno ime za drugi objekat, postavlja se pitanje da li su one uopće posebni objekti, ili predstavljaju isti objekat kao i objekat na koji su vezani. Da bi se u potpunosti shvatio mehanizam prenosa parametara putem referenci, koji ćemo uskoro objasniti, moramo reći da je, tehnički gledano, odgovor potvrđan. Reference jesu posebni objekti, koji zauzimaju mjesto u memoriji neovisno od objekta na koji su vezani. Reference zapravo u sebi sadrže informaciju o tome *gdje se u memoriji nalazi objekat za koje su vezane* (tj. adresu objekta za koji su vezane), tako da svaki pristup referenci koristi ovu pohranjenu informaciju da

indirektno pristupi objektu na koji referenca upućuje. U tom smislu, reference su slične tzv. *pokazivačima*, sa kojima ćemo se upoznati kasnije. Međutim, za razliku od pokazivača, koji se ponašaju bitno drugačije od objekata na koji upućuju, reference se ponašaju *istovjetno* kao i objekat sa kojim su vezani. Drugim riječima, ne postoji nikakav način kojim bi program mogao utvrditi da se referenca i po čemu razlikuje od objekta za koji je vezana, odnosno da ona *nije* objekat za koji je vezana. Čak i neke od najdelikatnijih operacija koje bi se mogle primijeniti na referencu obaviće se nad objektom za koji je referenca vezana. Na primjer, operator “**sizeof**” primijenjen na referencu neće vratiti kao rezultat veličinu same reference, nego veličinu objekta na koji referenca upućuje. Dakle, referenca i objekat za koji je referenca vezana tehnički posmatrano *nisu isti objekat*, ali program nema način da to sazna. Sa aspekta izvršavanja programa, referenca i objekat za koji je ona vezana predstavljaju *potpuno isti objekat!* Ipak, potrebno je naglasiti da tip nekog objekta i tip reference na taj objekat *nisu isti*. Dok je, u prethodnom primjeru, promjenljiva “a” tipa *cijeli broj* (tj. tipa “**int**”) dотле je promjenljiva “b” tipa *referenca na cijeli broj* (ovo je tip bez posebnog imena, koji se često obilježava kao “**int &**”). Tip reference je moguće posebno imenovati “**typedef**” naredbom. Na primjer, sljedeća sekvenca naredbi prvo uvodi tip “Referenca” koji predstavlja referencu na cijeli broj, a zatim novouvedeni tip koristi za deklaraciju reference “b” koja upućuje na promjenljivu “a”:

```
typedef int &Referenca;
Referenca b = a;
```

Početnik se ovom razlikom u tipu između reference i objekta na koji referenca upućuje ne treba da zamara, s obzirom da je ona uglavnom nebitna, osim u nekim vrlo specifičnim slučajevima, na koje ćemo ukazati onog trenutka kada se pojave. U suštini, nije loše znati da ova razlika *postoji*, s obzirom da je u jeziku C++ pojam tipa izuzetno važan. Bez obzira na ovu razliku u tipu, čak ni ona se ne može iskoristiti da program sazna da neka promjenljiva predstavlja referencu na neki objekat, a ne sam objekat. Reference u jeziku C++ su zaista izuzetno dobro “zamaskirane”.

Mada reference na prvi pogled izgledaju dosta čudno, one u jeziku C++ imaju mnogobrojne primjene. Jedna od najočiglednijih primjena je tzv. *prenos parametara po referenci* (engl. *passing by reference*) koji omogućava rješenje problema postavljenog na početku ovog poglavlja. Naime, da bismo postigli da funkcija promjeni vrijednost svog stvarnog parametra, *dovoljno je da odgovarajući formalni parametar bude referencia*. Na primjer, funkciju “*Povecaj*” trebalo bi modificirati na sljedeći način:

```
void Povecaj (int &x) {
    x++;
}
```

Da bismo vidjeli šta se ovdje zapravo dešava, pretpostavimo da je ova funkcija pozvana na sljedeći način (“a” je neka cijelobrojna promjenljiva):

```
Povecaj (a);
```

Prilikom ovog poziva, formalni parametar “x” koji je *referenca* biće inicijaliziran stvarnim parametrom “a”. Međutim, prilikom inicijalizacije referenci, one se *vezuju* za objekat kojim su inicijalizirane, tako da se formalni parametar “x” vezuje za promjenljivu “a”. Stoga će se svaka promjena sadržaja formalnog parametra “x” zapravo odraziti na promjenu stvarnog parametra “a”, odnosno za vrijeme izvršavanja funkcije “*Povecaj*”, promjenljiva “x” se ponaša kao da ona u stvari *jeste* promjenljiva “a”. Ovdje je iskorištena osobina referenci da se one ponašaju tako kao da su one upravo objekat za koji su vezane. Po završetku funkcije, referenca “x” se *uništava*, kao i svaka druga lokalna promjenljiva na kraju bloka kojem pripada. Međutim, za vrijeme njenog života, ona je iskorištena da promijeni sadržaj stvarnog parametra “a”, koji razumljivo ostaje takav i nakon što referenca “x” prestane “živjeti”!

U slučaju kada je neki formalni parametar referenca, odgovarajući stvarni parametar *mora biti l-vrijednost* (tipično neka promjenljiva), jer se reference mogu vezati samo za l-vrijednosti. Drugim riječima, odgovarajući parametar ne smije biti broj, ili proizvoljan izraz. Stoga, pozivi poput sljedećih nisu dozvoljeni:

```
Povecaj(7);  
Povecaj(2 * a - 3);
```

U suštini, ako malo bolje razmislimo, vidjećemo da ovakvi pozivi zapravo i nemaju smisla. Funkcija "Povecaj" je dizajnirana sa ciljem da promijeni vrijednost svog stvarnog argumenta, što je nemoguće ukoliko je on, na primjer, broj. Broj ima svoju vrijednost koju nije moguće mijenjati!

Kako su individualni elementi niza također l-vrijednosti, kao stvarni parametar koji se prenosi po referenci može se upotrijebiti i individualni element niza. Tako, na primjer, ukoliko imamo deklaraciju

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

tada se sljedeći poziv sasvim legalno može iskoristiti za uvećavanje elementa niza sa indeksom 2:

```
Povecaj(niz[2]);
```

Ovo je posljedica činjenice da se referencia može vezati za bilo koju l-vrijednost, pa tako i za individualni element niza. Drugim riječima, deklaracija

```
int &element = niz[2];
```

sasvim je legalna, i nakon nje ime "element" postaje alternativno ime za element niza "niz" sa indeksom 2. Drugim riječima, naredba

```
element = 5;
```

imaće isti efekat kao i naredba

```
niz[2] = 5;
```

Moguće je čak definirati i *reference na čitav niz*, ali se ova mogućnost prilično rijetko koristi. Na primjer, deklaracija

```
int (&klon)[10] = niz;
```

deklarira referencu "klon" koja je vezana za niz "niz" (od 10 cijelih brojeva), i koja se, prema tome, može koristiti kao njegovo alternativno ime (tj. kao alternativno ime čitavog niza, a ne nekog njegovog individualnog elementa). Primijetimo pomalo čudnu sintaksu u kojoj se koriste i obične zagrade. Ukoliko bismo izostavili ove zagrade, smatralo bi se da želimo deklarirati *niz od 10 referenci na cijele brojeve* (a ne *referencu na niz od 10 cijelih brojeva*). Ovo bi dovelo do prijave greške, jer jezik C++ ne dozvoljava kreiranje nizova referenci. Ukoliko bi ovo bilo moguće, postojao bi trik koji bi omogućavao preusmjeravanje referenci (zasnovan na činjenici da se imena nizova upotrijebljena sama za sebe konvertiraju u adresu prvog elementa niza), a tvorci jezika C++ su željeli da takvu mogućnost spriječe po svaku cijenu.

Treba naglasiti da su reference u jeziku C++ mnogo fleksibilnije nego u većini drugih programskih jezika. Na primjer, u jeziku Pascal samo formalni parametri mogu biti reference, odnosno referencia kao pojam ne postoji izvan konteksta formalnih parametara, tako da se u Pascalu pojam reference (kao neovisnog objekta) uopće ne uvode, nego se samo govori o prenosu po referenci. S druge strane, u jeziku

C++ referenca može postojati kao objekat *posve neovisan o formalnim parametrima*, a prenos po referenci se prosto ostvaruje tako što se odgovarajući formalni parametar deklarira kao referenca.

Sljedeći primjer ilustrira realističniju situaciju u kojoj se javlja potreba za prenosom parametara po referenci nego što je bilo demonstrirano u trivijalnoj funkciji "Povecaj". Zamislimo da želimo da napravimo funkciju "UnesiMjesec" koja sa tastature prihvata redni broj mjeseca u opsegu od 1 do 12, pri čemu ponavlja unos sve dok korisnik ne unese broj u ispravnom opsegu, a zatim smješta unesenu vrijednost u promjenljivu koja je navedena kao argument. Na primjer, ako je izvršen poziv poput

```
UnesiMjesec(mjesec);
```

i ako korisnik unese vrijednost 4, vrijednost promjenljive "mjesec" treba da postane 4. Jasno je da moramo koristiti prenos *po referenci*, jer funkcija "UnesiMjesec" treba da *promjeni* vrijednost promjenljive "mjesec". Funkcija "UnesiMjesec" mogla bi izgledati na primjer ovako (podsetimo se da konstrukcija "**for**(;;)", slično kao i "**while(true)**" ili "**while(1)**" označava beskonačnu petlju iz koje se može izaći samo nasilnim putem):

```
void UnesiMjesec(int &m) {
    cout << "Unesi mjesec u opsegu od 1 do 12: ";
    for(;;) {
        cin >> m;
        if(cin && m >= 1 && m <= 12) return;
        cout << "Neispravan unos!\n";
        if(!cin) cin.clear();
        cin.ignore(10000, '\n');
    }
}
```

Šta bi se dogodilo da smo zaboravili deklarirati formalni parametar "m" kao referencu? U tom slučaju bi formalni argument "m" i stvarni argument "mjesec" prilikom poziva funkcije "UnesiMjesec" predstavljali striktno različite objekte. Vrijednost promjenljive "mjesec" bila bi, naravno, prenesena u parametar "m" (u ovom trenutku je potpuno nebitno kakva je ta vrijednost, niti da li je ta promjenljiva uopće imala neku konkretnu definiranu vrijednost). Nakon toga, sve manipulacije unutar funkcije sa promjenljivom "m" (uključujući i unos sa tastature) ne bi imale nikakav utjecaj na promjenljivu "mjesec". Po završetku funkcije "UnesiMjesec" formalni parametar "m" bio bi *uništen*, čime bi vrijednost koju smo unijeli sa tastature bila *izgubljena*, a vrijednost promjenljive "mjesec" bi na kraju bila *ista kao što je bila i prije poziva funkcije!*

Prema načinu na koji se koriste, odnosno prema *smjeru* toka informacija koji se putem njih prenose, parametre možemo podijeliti na *ulazne*, *izlazne* i *ulazno-izlazne*. Parametri koji se prenose po vrijednosti su uvijek *ulazni*, s obzirom da putem njih informacija može samo *ući* u funkciju, ali ne može iz nje *izaći*. Parametri koji se prenose po referenci u načelu također mogu biti striktno ulazni, ali se oni mnogo češće koriste kao *izlazni* odnosno *ulazno-izlazni*. Na primjer, parametar funkcije "UnesiMjesec" je izlazni parametar. Putem njega informacija o unesenom mjesecu *izlazi* iz funkcije, i smješta se u stvarni argument koji je upotrijebljen u pozivu funkcije. Pri tome je za funkciju posve nebitno kakva je bila vrijednost stvarnog argumenta prilikom poziva funkcije (ona će svakako biti prebrisana). Stoga je parametar ove funkcije striktno izlazni, s obzirom da funkcija putem njega ne prima nikakvu informaciju od onoga ko je poziva (bolje rečeno, *prima* ali je ne *koristi*). S druge strane, parametar funkcije "Povecaj" je tipičan primjer ulazno-izlaznog parametra. Preko njega funkcija dobija informaciju o vrijednosti stvarnog parametra, izmijeni ovu vrijednost, i vrati izmijenjenu vrijednost nazad u stvarni parametar. Drugim riječima, protok informacija se odvija u oba smjera.

Razmotrimo sada malo detaljnije opisane mehanizme prenosa parametara. Kod prenosa parametara *po*

vrijednosti, vrijednosti stvarnih parametara se *kopiraju* u formalne parametre. U ovom slučaju, funkcija samo dobija informaciju o vrijednosti nekog stvarnog parametra, ali nema nikakvu informaciju o tome gdje se on nalazi u memoriji, pa ga ne može ni promijeniti. S druge strane, kod prenosa *po referenci*, koji se u jeziku C++ realizira tako što se formalni parametar deklarira kao referenca, u funkciju se prenosi *informacija o mjestu u memoriji računara (adresi)* gdje se odgovarajući stvarni parametar čuva. Referenca na taj način saznaje gdje se nalazi objekat koji treba da predstavlja, i može preuzeti potpunu kontrolu nad njim. Nekada se ovaj način u malo slobodnijoj interpretaciji naziva i *prenos po imenu* (engl. *passing by name*) s obzirom da, na izvjestan način, formalni parametar pri pozivu funkcije saznaje *ime* objekta s kojim treba da se poistovijeti. U suštini, kod prenosa po vrijednosti, formalni i stvarni parametar uvijek predstavljaju različite objekte čak i kada imaju isto ime. S druge strane, možemo smatrati da kod prenosa po referenci formalni i stvarni parametar predstavljaju iste objekte čak i kada imaju različito ime (ova tvrdnja je tačna sa aspekta ponašanja mada, kao što smo već opisali, sa tehničkog aspekta nije posve precizna). Na ovu činjenicu (koja u početku može djelovati zbunjujuće) treba dobro obratiti pažnju, jer je ona jedan od najčešćih uzročnika grešaka u programima koje koriste funkcije!

Neki početnici često pogrešno shvataju da kod parametara koji se prenose po referenci dolazi do *dvostrukog kopiranja*, odnosno da se prilikom poziva funkcije vrijednosti stvarnih parametara kopiraju u formalne parametre, a da se na završetku funkcije vrijednosti formalnih parametara kopiraju nazad u stvarne parametre. Mada sa aspekta funkcioniranja izgleda da je tako, ovo nije ono što se zaista dešava, nego se formalni i stvarni parametri *poistovjećuju* i ni do kakvog kopiranja uopće ne dolazi (već samo do prenosa adrese). Kopiranje parametara može biti vremenski zahtjevna operacija u slučaju kada su parametri masivni objekti (tj. objekti koji zauzimaju puno memorijskog prostora) sa kakvim ćemo se susretati kasnije. Stoga je sa aspekta efikasnosti važno znati da kod prenosa po referenci ne dolazi ni do kakvog kopiranja parametara, a pogotovo ne do dvostrukog kopiranja.

Funkcije koje ne vraćaju vrijednost, a kod kojih se jedan parametar prenosi po referenci nisu pretjerano korisne, s obzirom da se načelno isti efekat, samo uz drugačiji način pozivanja, može ostvariti i korištenjem funkcija koje vraćaju vrijednost. Tako smo, na primjer, funkciju "UnesiMjesec" mogli realizirati kao funkciju bez parametara a koja *vraća kao rezultat* uneseni mjesec:

```
int UnesiMjesec() {
    cout << "Unesi mjesec u opsegu od 1 do 12: ";
    for(;;) {
        int m;
        cin >> m;
        if(cin && m >= 1 && m <= 12) return m;
        cout << "Neispravan unos!\n";
        if(!cin) cin.clear();
        cin.ignore(10000, '\n');
    }
}
```

Naravno, ovaku funkciju bismo morali pozivati na nešto drugačiji način, pozivom poput

```
mjesec = UnesiMjesec();
```

U ovom slučaju, funkcija ne *smješta* traženu vrijednost u svoj parametar, nego je jednostavno *vraća* kao povratnu vrijednost, koju prostom dodjelom smještamo u željenu promjenljivu "mjesec". Drugim riječima, imamo način da postignemo isti efekat, samo uz neznatno izmijenjenu sintaksu. Čak se smatra i da je ovaj drugi način (tj. korištenje povratne vrijednosti) više "u duhu" jezika C++. Međutim, prenos po referenci dolazi do punog izražaja kada je potrebno prenijeti *više od jedne vrijednosti* iz funkcije nazad na mjesto njenog poziva. Pošto funkcija ne može vratiti kao rezultat više vrijednosti, kao rezultat se nameće korištenje izlaznih parametara putem prenosa po referenci, pri čemu će funkcija koristeći reference prosto

smjestiti tražene vrijednosti u odgovarajuće stvarne parametre koji su joj proslijedeni. Ova tehnika je ilustrirana u sljedećoj verziji programa za proračun obima i površine kruga, u kojoj funkcija "ProracunajKrug" smješta izračunate vrijednosti obima i površine kruga sa poluprečnikom koji joj je proslijeden kao prvi parametar u promjenljive koje su proslijedene kao drugi i treći parametar. Ovo smještanje se ostvaruje tako što su drugi i treći formalni parametar ove funkcije ("O" i "P") deklarirani kao reference, koje se za vrijeme izvršavanja funkcije vezuju za odgovarajuće stvarne argumente (uspit, nije posebno mudro promjenljivu nazvati "O", s obzirom da se slovo "O" lako može pobrkati sa oznakom "0" koja predstavlja nulu):

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

void ProracunajKrug(double r, double &O, double &P) {
    const double PI(3.141592654);
    O = 2 * PI * r;
    P = PI * r * r;
}

int main() {
    double poluprecnik, obim, povrsina;
    cin >> poluprecnik;
    ProracunajKrug(poluprecnik, obim, povrsina);
    cout << "Obim: " << obim << endl
        << "Površina: " << povrsina << endl;
    return 0;
}
```

Sličnu situaciju imamo i u sljedećem programu u kojem je definirana funkcija "RastaviSekunde", čiji je prvi parametar broj sekundi, a koja kroz drugi, treći i četvrti parametar prenosi na mjesto poziva informaciju o broju sati, minuta i sekundi koji odgovaraju zadanim broju sekundi:

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

void RastaviSekunde(int br_sek, int &sati, int &minute, int &sekunde) {
    sati = br_sek / 3600;
    minute = (br_sek % 3600) / 60;
    sekunde = br_sek % 60;
}

int main() {
    int sek, h, m, s;
    cout << "Unesi broj sekundi: ";
    cin >> sek;
    RastaviSekunde(sek, h, m, s);
    cout << "h = " << h << " m = " << m << " s = " << s << endl;
    return 0;
}
```

Kao što je već rečeno, stvarni parametri koji se prenosi po vrijednosti mogu bez ikakvih problema biti konstante ili izrazi, dok stvarni parametri koji se prenose po referenci *moraju* biti l-vrijednosti (obično promjenljive). Tako su uz funkciju "RastaviSekunde" iz prethodnog primjera sasvim legalni pozivi

```
RastaviSekunde(73 * sek + 13, h, m, s);
RastaviSekunde(12322, h, m, s);
```

dok sljedeći pozivi nisu legalni, jer odgovarajući stvarni parametri nisu l-vrijednosti:

```

RastaviSekunde(sek, 3 * h + 2, m, s);
RastaviSekunde(sek, h, 17, s);
RastaviSekunde(1, 2, 3, 4);
RastaviSekunde(sek + 2, sek - 2, m, s);

```

Navedimo još jedan primjer u kojem se koriste reference kao parametri. U Britaniji i Sjevernoj Americi se pored metričkog sistema još uvijek koriste stare jedinice za dužinu, stope i inči (1 stopa ima 12 inča, a u jednom metru ima $1250/381 \approx 3.280839895$ stopa). Ove mjerne jedinice (kao i neke druge podjednako nestandardne jedinice za težinu i zapreminu) poznate su pod nazivom "kraljevske" jedinice. Ovdje je dat program u kojem je upotrijebljena funkcija koja pretvara dužinu u metrima proslijedenu kao prvi parametar u dužinu izraženu u stopama i inčima (zaokruženo na najbliži cijeli broj inča), pri čemu se odgovarajući iznosi u stopama i inčima smještaju u promjenljive proslijedene kao drugi i treći parametar u funkciju:

```

#include <iostream>
using namespace std;

// Pretvara broj metara predstavljen parametrom "metri" u broj stopa
// i inča, i vraća ih kroz parametre "stope" i "inci"

void PretvoriUKraljevske(double metri, int &stope, int &inci) {
    const double StopaPoMetru(1250/381);
    double pomocna = metri * StopaPoMetru;
    stope = int(pomocna);
    inci = int((pomocna - stope) * 12);
}

int main() {
    double broj_metara;
    int broj_stopa, broj_inca;
    cout << "Unesi dužinu u metrima: ";
    cin >> broj_metara;
    PretvoriUKraljevske(broj_metara, broj_stopa, broj_inca);
    cout << "Dužina u kraljevskim jedinicama je: "
        << broj_stopa << " stopa i " << broj_inca << " inča\n";
    return 0;
}

```

U svim dosadašnjim primjerima (osim u trivijalnom slučaju funkcije "Povecaj") parametri koji su se prenosili po referenci korišteni su isključivo kao izlazni parametri, odnosno nikakva informacija se putem njih nije prenosila sa mjesta poziva funkcije u samu funkciju. U sljedećem primjeru definirana je interesantna funkcija "Razmijeni" koja razmjenjuje sadržaj dvije realne promjenljive (odnosno dvije l-vrijednosti) koje joj se proslijeduju kao parametri:

```

void Razmijeni(double &p, double &q) {
    double pomocna = p;
    p = q; q = pomocna;
}

```

Na primjer, ukoliko je vrijednost promjenljive "a" bila 2.13 a sadržaj promjenljive "b" 3.6, nakon izvršenja naredbe

```
Razmijeni(a, b);
```

vrijednost promjenljive “`a`” će biti 3.6, a vrijednost promjenljive “`a`” biće 2.13. Nije teško uvidjeti kako ova funkcija radi: njeni formalni parametri “`p`” i “`q`”, koji su reference, vežu se za navedene stvarne parametre. Funkcija pokušava da razmijeni dvije reference, ali će se razmijena obaviti nad promjenljivim za koje su ove reference vezane. Jasno je da se ovakav efekat može ostvariti samo putem prenosa po referenci, jer u suprotnom funkcija “`Razmijeni`” ne bi mogla imati utjecaj na svoje stvarne parametre.

Kako su individualni elementi niza također l-vrijednosti, funkcija “`Razmijeni`” se lijepo može iskoristiti za razmjenu dva elementa niza. Na primjer, ukoliko je “`niz`” neki niz realnih brojeva (sa barem 6 elemenata), tada će naredba

```
Razmijeni(niz[2], niz[5]);
```

razmjeniti elemente niza sa indeksima 2 i 5 (tj. treći i šesti element).

Bitno je naglasiti da se kod prenosa po referenci formalni i stvarni parametri *moraju u potpunosti slagati po tipu*, jer se reference mogu vezati samo za promjenljive odgovarajućeg tipa (osim u nekim rijetkim izuzecima, sa kojima ćemo se susresti kasnije). Na primjer, ukoliko funkcija ima formalni parametar koji je referencia na tip “`double`”, odgovarajući stvarni parametar ne može biti npr. tipa “`int`”. Razlog za ovo ograničenje nije teško uvidjeti. Razmotrimo, na primjer, sljedeću funkciju:

```
void Problem(double &x) {
    x = 3.25;
}
```

Pretpostavimo da se funkcija “`Problem`” može pozvati navodeći neku cijelobrojnu promjenljivu kao stvarni argument. Formalni parametar “`x`” trebao bi se nakon toga vezati i poistovjetiti sa tom promjenljivom. Međutim, funkcija smješta u “`x`” vrijednost koja nije cijelobrojna. U skladu sa djelovanjem referenci, ova vrijednost trebala bi da se zapravo smjesti u promjenljivu za koju je ova referencia vezana, što je očigledno nemoguće s obzirom da se radi o cijelobrojnoj promjenljivoj. Dakle, smisao načina na koji se reference ponašaju ne može biti ostvaren, što je ujedno i razlog zbog kojeg se formalni i stvarni parametar u slučaju prenosa po referenci moraju u potpunosti slagati po tipu.

Posljedica činjenice da se parametri koji se prenose po referenci moraju u potpunosti slagati po tipu je da se maloprije napisana funkcija “`Razmijeni`” ne može primijeniti za razmjenu dvije promjenljive koje nisu tipa “`double`”, npr. dvije promjenljive tipa “`int`” (pa čak ni promjenljive tipa “`float`”), bez obzira što sama funkcija ne radi ništa što bi suštinski trebalo ovisiti od tipa promjenljive. Kao rješenje ovog problema možemo koristiti *preklapanje funkcija po tipu*. Naime, sasvim je dozvoljeno napraviti još jednu verziju funkcije “`Razmijeni`” koja razmjenjuje dvije cijelobrojne promjenljive:

```
void Razmijeni(int &p, int &q) {
    int pomocna = p;
    p = q; q = pomocna;
}
```

U ovom slučaju, prilikom pokušaja razmjene dvije promjenljive pozivom funkcije “`Razmijeni`” pozvaće se odgovarajuća verzija ovisno o tome da li su promjenljive tipa “`double`” ili “`int`”. Međutim, razmjena opet neće raditi za neki treći tip (npr. “`char`”, “`float`”, “`bool`” ili neki pobrojani tip). Naravno, principijelno je moguće napraviti verziju funkcije “`Razmijeni`” za svaki tip za koji nam treba razmjena, ali ovo dovodi do potrebe za pisanjem velikog broja gotovo identičnih funkcija istih imena. Kasnije ćemo vidjeti kako se ovaj problem može elegantnije riješiti upotrebom tzv. *šablonu i generičkih funkcija*.

Na ovom mjestu nije loše ukazati na jednu činjenicu koja često može zbuniti početnika. Razmotrimo šta će ispisati sljedeći program:

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

void Potprogram(int &a, int &b) {
    cout << a << b;
    a += 3; b *= 2;
    cout << a << b;
}

int main() {
    int a(2), b(5);
    cout << a << b;
    potprogram(b, a);
    cout << a << b;
    return 0;
}
```

Da su za imenovanje formalnih parametara upotrijebljena neka druga slova a ne "a" i "b" (npr. "p" i "q"), vjerovatno biste bez ikakvih problema odgovorili da će biti ispisana sekvenca "25528448". Naravno, i u ovom slučaju biće ispisana ista sekvenca, s obzirom da rad programa ne može ovisiti od toga kako smo imenovali formalne parametre. Međutim, ukoliko probate neposredno pratiti tok ovog programa *ovako kako je napisan*, veoma se lako možete "zapetljati". Naime, pri pozivu funkcije "Potprogram", njen formalni parametar sa imenom "a" (koji je referenca) vezuje se za promjenljivu "b" iz glavnog programa, dok se formalni parametar sa imenom "b" vezuje za promjenljivu "a" iz glavnog programa. Stoga, sve ono što se obavlja sa promjenljivom (referencom) "a" u potprogramu, zapravo se obavlja sa promjenljivom "b" u glavnom programu, a sve što se obavlja sa promjenljivom "b" u potprogramu djeluje na promjenljivu "a" u glavnom programu. Izgleda kao da su u potprogramu promjenljive "a" i "b" razmijenile svoja imena u odnosu na glavni program! Ovakve zavrzlame ne dešavaju se često u praktičnim situacijama, ali je potrebno da shvatite šta se ovdje tačno dešava da biste u potpunosti ovladali mehanizmom prenosa parametara.

Znanja koja smo do sada stekli o prenosu parametara iskoristićemo da učinimo *modularnim* program koji računa dan u sedmici za proizvoljan datum u okviru 2000. godine, koji je ranije napisan na nemodularan način:

```
#include <iostream>
using namespace std;
const int mjeseci[12] = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int main() {
    void UnesiDatum(int &, int &); // Prototipovi korištenih funkcija
    int BrojDanaDoZadanog(int, int);
    void IspisiDan(int);
    int dan, mjesec;
    UnesiMjesec(dan, mjesec);
    IspisiDan(BrojDanaDoZadanog(dan, mjesec) % 7);
    return 0;
}
// Unosi datum, uz provjeru ispravnosti unosa
```

```

void UnesiDatum(int &dan, int &mjesec) {
    for(;;) {
        cout << "Unesite datum u obliku DD MM: ";
        cin >> dan >> mjesec;
        if(cin && mjesec >= 1 && mjesec <= 12
            && dan < 1 && dan > mjeseci[mjesec]) return;
        if(!cin) cin.clear();
        cout << "Unijeli ste besmislen datum!\n";
        cin.ignore(10000, '\n');
    }
}

// Određuje broj dana od 1. I 2000. do zadanog datuma
int BrojDanaDoZadanog(int dan, int mjesec) {
    int suma(0);
    for(int i = 0; i < mjesec - 1; i++) suma += mjeseci[i];
    return suma + dan - 1;
}

// Ispisuje ime dana prema ostatku koji je proslijeden kao parametar
void IspisiDan(int ostatak) {
    switch(ostatak) {
        case 0: cout << "Subota"; break;
        case 1: cout << "Nedjelja"; break;
        case 2: cout << "Ponedjeljak"; break;
        case 3: cout << "Utorak"; break;
        case 4: cout << "Srijeda"; break;
        case 5: cout << "Četvrtak"; break;
        case 6: cout << "Petak";
    }
}

```

Obratimo pažnju na prototip potprograma “UnesiDatum”. U njemu su izostavljena imena formalnih parametara, ali se oznaka za referencu *ne smije izostaviti*. Formalni parametri ovog potprograma nisu tipa “**int**”, nego su tipa *reference na “int”* (odnosno “**int &**”), što mora biti jasno naznačeno, bez obzira što su imena parametara izostavljena. Također, primjetimo da smo niz “mjeseci” deklarirali kao globalni, s obzirom da nam je njegov sadržaj potreban kako u potprogramu “UnesiDatum”, tako i u potprogramu “BrojDanaDoZadanog”. Kao alternativa bi se ovaj niz također mogao prenosi kao parametar u potprograme (prenos nizova kao parametara biće objašnjen u sljedećem poglavlju), mada je ovakvo rješenje sasvim zadovoljavajuće, jer potprogrami samo čitaju sadržaj ovog niza, bez mijenjanja njegovog sadržaja, čime je izbjegnuto nehotično stvaranje zavisnosti između potprograma putem mijenjanja sadržaja niza koji oba zajednički koriste. Pomoću ključne riječi “**const**” jasno je istaknuta namjera da sadržaj niza “mjeseci” neće i ne smije biti promijenjen. Kao što je već istaknuto, konstantni objekti tipično nisu problematični čak i ukoliko im je vidljivost globalna. Ipak, bez velikog razloga ne treba koristiti čak ni konstantne objekte sa globalnom vidljivošću, s obzirom da dobar dizajn programa nalaže da oni dijelovi programa koji ne treba da koriste neki resurs ne treba ni da ga vide. Na taj način se dizajn održava “čistijim” i manje je podložan greškama.

Funkcije koje koriste prenos parametara po referenci su najčešće funkcije koje *ne vraćaju vrijednost*, iako ne postoji pravilo koje bi nalagalo da mora biti tako. Naime, ukoliko bi funkcija istovremeno koristila prenos po referenci i imala povratnu vrijednost, tada bi informacije iz funkcije “izlazile” na dva suštinski različita načina, što može stvoriti konfuziju. Na primjer, funkcija

```

int MinIMax(int a, int b, int &max) {
    if(a > b) {

```

```

        max = a; return b;
    }
else {
    max = b; return a;
}
}

```

kao rezultat vraća manji od svoja prva dva parametra, a smješta u treći parametar veći od svoja prva dva parametra. Ovo je svakako konfuzno, tako da je bolje izbjegći povratnu vrijednost, i umjesto toga uvesti i četvrti parametar, tako da funkcija smješta kako minimum tako i maksimum u parametre proslijedene po referenci, kao u sljedećoj funkciji:

```

void NadjiMinIMax(int a, int b, int &min, int &max) {
    if(a > b) {
        max = a; min = b;
    }
    else {
        max = b; min = a;
    }
}

```

Još jedan razlog zbog kojeg treba izbjegavati funkcije koje vraćaju vrijednost i koriste prenos po referenci je činjenica da takve funkcije istovremeno mogu biti upotrijebljene unutar izraza i mijenjati vrijednosti svojih stvarnih argumenata, čime se zapravo ostvaruje bočni efekat nad argumentima. Pogledajmo, na primjer, gore napisanu funkciju "MinIMax". Ona se, u principu, može pozvati na sljedeći način (uz pretpostavku da su "x" i "y" cjelobrojne promjenljive):

```
x = MinIMax(3, 5, y) + 5;
```

Međutim, iz ovog poziva nimalo nije očigledno da će on dovesti do promjene sadržaja promjenljive "y". Što je još gore, moguće je kreirati izraze nedefiniranog dejstva, poput sljedećeg:

```
x = MinIMax(3, 5, y) * y;
```

Naime, ovdje nije definirano da li će se kao drugi operand operacije množenja upotrijebiti vrijednost promjenljive "y" prije izmjene ili poslije izmjene (tj. vrijednost po izlasku i funkcije "MinIMax"), jer standardom nije definiran redoslijed izračunavanja operanada (tj. da li će prvo biti izračunat lijevi ili desni operand operacije množenja). Ovdje zapravo imamo upotrebu promjenljive "y", nad kojom je obavljen bočni efekat, na dva mjesta u izrazu što, kao što već znamo, vodi ka nedefiniranom ponašanju. Kao još jedan primjer, posmatrajmo sljedeći logički izraz (uvjet):

```
x >= 0 && MinIMax(a, b, y) > 3
```

Programer koji bi se oslonio na to da će, zbog poziva funkcije "MinIMax" promjenljiva "y" sigurno dobiti vrijednost veće od promjenljivih "a" i "b" došao bi u gadnu zabludu, zbog specifičnosti operatora "&&". Naime, ukoliko je vrijednost promjenljive "x" manja od nule, drugi operand operatora "&&" uopće se ne izračunava, tako da funkcija "MinIMax" uopće neće biti pozvana. Izloženi primjeri ukazuju na to da funkcije koje istovremeno vraćaju vrijednost i koriste prenos parametara po referenci treba izbjegavati, a ukoliko ih već definiramo, treba ih koristiti sa izuzetnim oprezom.

Jedan od slučaja u kojem se može smatrati opravdanim definirati funkciju koja vraća vrijednost i koristi prenos po referenci je slučaj u kojem funkcija kao rezultat vraća neku *statusnu informaciju* o obavljenom poslu, koji može uključivati i smještanje informacija u parametre prenesene po referenci. Na primjer, u sljedećem programu definirana je funkcija "KvJed", koja je principijelno dosta slična funkciji

“KvadratnaJednacina” iz jednog od ranijih poglavlja, samo što ne vrši nikakav ispis na ekran (prikladnije ime ove funkcije bilo bi “NadjiRjesenjaKvadratneJednacine”, koje ovdje nećemo koristiti radi dužine). Ova funkcija nalazi rješenja kvadratne jednačine čiji se koeficijenti zadaju preko prva tri parametra. Za slučaj da su rješenja realna, funkcija ih *prenosi* nazad na mjesto poziva funkcije kroz četvrti i peti parametar i *vraća* vrijednost “**true**” kao rezultat. Za slučaj kada rješenja nisu realna, funkcija vraća “**false**” kao rezultat, a četvrti i peti parametar ostaju netaknuti. Obratite pažnju na način kako je ova funkcija upotrijebljena u glavnom programu:

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

bool KvJed(double a, double b, double c, double &x1, double &x2) {
    double d = b * b - 4 * a * c;
    if(d < 0) return false;
    x1 = (-b - sqrt(d)) / (2*a);
    x2 = (-b + sqrt(d)) / (2*a);
    return true;
}

int main() {
    double a, b, c;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    cout << "c = ";
    cin >> c;
    double x1, x2;
    if(KvJed(a, b, c, x1, x2))
        cout << "x1 = " << x1 << "\nx2 = " << x2;
    else cout << "Jednačina nema realnih rješenja!";
    return 0;
}
```

Iz svega što je do sada rečeno može se zaključiti da izlazne i ulazno-izlazne parametre treba koristiti sa oprezom, i samo onda kada je zaista neophodno. Pošto se iz načina pozivanja funkcije ne može zaključiti da će funkcija eventualno izmijeniti svoj stvarni argument, treba to učiniti jasnim davanjem odgovarajućeg imena funkciji, tako da ta činjenica bude očigledna (npr. davanjem naziva poput “UnesiMjesec”, ili “NadjiMinIMax” umjesto samo “MinIMax”). Posebno su problematični čisto izlazni parametri, jer se njihovom upotreboru ne može ostvariti preporka da svaka promjenljiva treba po mogućnosti biti smisleno inicijalizirana odmah pri deklaraciji. Naime, pri korištenju izlaznih parametara, stvarni argumenti dobijaju smislene vrijednosti tek nakon poziva funkcije.

Sljedeći primjer ilustrira upotrebu pobrojanih tipova kao parametara. Neka je definiran pobrojani tip “Boje” koji definira četiri pobrojane konstante: “Crvena”, “Plava”, “Zelena” i “Bijela”. Funkcija “IspisiBoju” ima jedan parametar tipa “Boje” i ona ispisuje naziv odgovarajuće boje na ekranu. Na primjer, poziv funkcije

```
IspisiBoju(Crvena);
```

ispisaće riječ “Crvena” na ekranu. Funkcija “UnesiBoju” također ima jedan parametar tipa “Boje”. Ova funkcija zahtijeva od korisnika da unese sa tastature boju predstavljenu jednim slovom ‘C’, ‘P’, ‘Z’ ili ‘B’ (respektivno za crvenu, plavu, zelenu i bijelu boju), i dodjeljuje odgovarajuću pobrojanu vrijednost promjenljivoj zadanoj kao stvarni parametar funkcije. Npr. ako prepostavimo da je “moja_boja” promjenljiva tipa “Boja”, tada će naredba

```
UnesiBoju(moja_boja);
```

smjestiti u promjenljivu "moja_boja" vrijednost "Crvena", "Plava", "Zelena" ili "Bijela" ovisno o tome da li je korisnik unijeo 'C', 'P', 'Z' ili 'B'. Funkcija pored toga vodi računa o ispravnosti unesenih podataka. Obje funkcije su demonstrirane u kratkom testnom programu. Obratite pažnju da se u funkciji "IspisiBoju" parametar prenosi po vrijednosti, a u funkciji "UnesiBoju" po referenci:

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

enum Boje {Crvena, Plava, Zelena, Bijela};

void IspisiBoju(Boje b) {
    switch(b) {
        case Crvena: cout << "Crvena"; break;
        case Plava: cout << "Plava"; break;
        case Zelena: cout << "Zelena"; break;
        case Bijela: cout << "Bijela";
    }
}

void UnesiBoju(Boje &b) {
    for(;;) {
        char znak;
        cin >> znak;
        switch(znak) {
            case 'C' : b = Crvena; return;
            case 'P' : b = Plava; return;
            case 'Z' : b = Zelena; return;
            case 'B' : b = Bijela; return;
        }
        cout << "Neispravan unos!\n";
        cin.ignore(10000, '\n');
    }
}

int main() {
    Boje moja_boja;
    cout << "Unesi slovo koje predstavlja boju: ";
    UnesiBoju(moja_boja);
    cout << "Unijeli ste boju: ";
    IspisiBoju(moja_boja);
    return 0;
}
```

Do sada smo vidjeli da formalni parametri funkcija mogu biti reference. Međutim, interesantno je da rezultat koji vraća funkcija *također može biti referencia* (nekada se ovo naziva *vraćanje vrijednosti po referenci*). Ova mogućnost se ne koristi prečesto, ali postoje situacije kada je ovakva mogućnost od neprocjenjive važnosti. Naime, kasnije ćemo vidjeti da se neki problemi u objektno-orientiranom pristupu programiranja ne bi mogli riješiti da ne postoji ovakva mogućnost. U slučaju funkcija čiji je rezultat referenca, nakon završetka izvršavanja funkcije ne vrši se *prosta zamjena* poziva funkcije sa vraćenom *vrijednošću* (kao u slučaju kada rezultat nije referenca), nego se poziv funkcije *potpuno poistovjećuje* sa vraćenim *objektom* (koji u ovom slučaju mora biti promjenljiva, ili općenitije l-vrijednost). Ovo poistovjećivanje ide dotle da poziv funkcije postaje l-vrijednost, tako da se poziv funkcije čak može upotrijebiti sa lijeve strane operatora dodjele, ili prenijeti u funkciju čiji je formalni parametar referenca

(obje ove radnje zahtijevaju l-vrijednost).

Sve ovo može na prvi pogled djelovati dosta nejasno. Stoga ćemo vraćanje reference kao rezultata ilustrirati konkretnim primjerom. Posmatrajmo sljedeću funkciju, koja obavlja posve jednostavan zadatak (vraća kao rezultat veći od svoja dva parametra):

```
int VeciOd(int x, int y) {
    if(x > y) return x;
    else return y;
}
```

Pretpostavimo sada da imamo sljedeći poziv:

```
int a(5), b(8);
cout << VeciOd(a, b);
```

Ova sekvenca naredbi će, naravno, ispisati broj "8", s obzirom da će poziv funkcije "VeciOd(a, b)" po povratku iz funkcije biti zamijenjen izračunatom vrijednošću (koja očigledno iznosi 8, kao veća od dvije vrijednosti 5 i 8). Modificirajmo sada funkciju "VeciOd" tako da joj formalni parametri postanu reference:

```
int VeciOd(int &x, int &y) {
    if(x > y) return x;
    else return y;
}
```

Prethodna sekvenca naredbi koja poziva funkciju "VeciOd" i dalje radi ispravno, samo što se sada vrijednosti stvarnih parametara "a" i "b" ne *kopiraju* u formalne parametre "x" i "y", nego se formalni parametri "x" i "y" *poistovjećuju* sa stvarnim parametrima "a" i "b". U ovom konkretnom slučaju, krajnji efekat je isti. Ipak, ovom izmjenom smo ograničili upotrebu funkcije, jer pozivi poput

```
cout << VeciOd(5, 7);
```

više nisu mogući, s obzirom da se reference ne mogu vezati za brojeve. Međutim, ova izmjena predstavlja korak do posljednje izmjene koju ćemo učiniti: modificiraćemo funkciju tako da kao rezultat *vraća referencu*:

```
int &VeciOd(int &x, int &y) {
    if(x > y) return x;
    else return y;
}
```

Ovako modificirana funkcija vraća kao rezultat *referencu na veći od svoja dva parametra*, odnosno sam poziv funkcije ponaša se kao da je on upravo vraćeni objekat. Na primjer, pri pozivu "VeciOd(a, b)" formalni parametar "x" se poistovjećuje sa promjenljivom "a", a formalni parametar "y" sa promjenljivom "b". Uz pretpostavku da je vrijednost promjenljive "b" veća od promjenljive "a" (kao u prethodnim primjerima poziva), funkcija će vratiti referencu na formalni parametar "y". Kako reference na reference ne postoje, biće zapravo vraćena referenca na onu promjenljivu koju referenca "y" predstavlja, odnosno promjenljivu "b". Kad kažemo da reference na reference ne postoje, mislimo na sljedeće: ukoliko imamo deklaracije poput

```
int &q = p;
int &r = q;
```

tada “*x*” ne predstavlja referencu na referencu “*q*”, već referencu na promjenljivu za koju je referenca “*q*” vezana, odnosno na promjenljivu “*p*” (odnosno “*x*” je također referenca na “*p*”). Dakle, funkcija je vratila referencu na promjenljivu “*b*”, što znači da će se poziv “*VeciOd(a, b)*” ponašati upravo kao promjenljiva “*b*”. To znači da postaje moguća ne samo upotreba ove funkcije kao obične vrijednosti, već je moguća njena upotreba u bilo kojem kontekstu u kojem se očekuje neka promjenljiva (ili l-vrijednost). Tako su, na primjer, sasvim legalne konstrukcije poput sljedećih (ovdje su “*Povecaj*” i “*Razmijeni*” funkcije iz ranijih primjera, a “*c*” neka cjelobrojna promjenljiva):

```
VeciOd(a, b) = 10;
VeciOd(a, b)++;
VeciOd(a, b) += 3;
Povecaj(VeciOd(a, b));
Razmijeni(VeciOd(a, b), c);
```

Posljednji primjer razmjenjuje onu od promjenljivih “*a*” i “*b*” čija je vrijednost veća sa promjenljivom “*c*”. Drugim riječima, posljednja napisana verzija funkcije “*VeciOd*” ima tu osobinu da se poziv poput “*VeciOd(a, b)*” ponaša ne samo kao *vrijednost* veće od dvije promjenljive “*a*” i “*b*”, nego se ponaša kao da je ovaj poziv *upravo ona promjenljiva* od ove dvije čija je vrijednost veća! Usput, upravo smo naučili šta još može biti l-vrijednost osim obične promjenljive ili elementa niza: l-vrijednost može biti i *poziv funkcije koji vraća referencu* (kasnije ćemo upoznati još izraza koji mogu biti l-vrijednosti). Kako je poziv funkcije koja vraća referencu l-vrijednost, za njega se može vezati i referenca (da nije tako, ne bi bili dozvoljeni gore navedeni pozivi u kojima je poziv funkcije “*VeciOd*” iskorišten kao stvarni argument u funkcijama kod kojih je formalni parametar referenca). Stoga je sasvim dozvoljena deklaracija poput sljedeće:

```
int &v = VeciOd(a, b);
```

Nakon ove deklaracije, referenca “*v*” ponaša se kao ona od promjenljivih “*a*” i “*b*” čija je vrijednost veća (preciznije, kao ona od promjenljivih “*a*” i “*b*” čija je vrijednost *bila veća u trenutku deklariranja ove reference*).

Vraćanje referenci kao rezultata funkcije treba koristiti samo u izuzetnim prilikama, i to sa dosta opreza, jer ova tehnika podliježe brojnim ograničenjima. Na prvom mjestu, jasno je da se prilikom vraćanja referenci kao rezultata iza naredbe “**return**” mora naći isključivo neka promjenljiva ili l-vrijednost, s obzirom da se reference mogu vezati samo za l-vrijednosti. Kompajler će prijaviti grešku ukoliko ne ispoštujemo ovo ograničenje. Mnogo opasniji problem je ukoliko vratimo kao rezultat referencu na neki objekat koji *prestaje živjeti nakon prestanka funkcije* (npr. na neku lokalnu promjenljivu, uključujući i formalne parametre funkcije koji nisu reference). Na primjer, zamislimo da smo funkciju “*VeciOd*” napisali ovako:

```
int &VeciOd(int x, int y) {
    if(x > y) return x;
    else return y;
}
```

Ovdje funkcija i dalje vraća referencu na veći od parametara “*x*” i “*y*”, međutim ovaj put ovi parametri nisu reference (tj. ne predstavljaju neke objekte koji postoje izvan ove funkcije) nego samostalni objekti koji imaju smisao samo unutar funkcije, i koji se uništavaju nakon završetka funkcije. Drugim riječima, funkcija će vratiti referencu na *objekat koji je prestao postojati*, odnosno prostor u memoriji koji je objekat zauzimao raspoloživ je za druge potrebe. Ovakva referenca naziva se *viseća referencia* (engl. *dangling reference*). Ukoliko za rezultat ove funkcije vežemo neku referencu, ona će se vezati za objekat

koji zapravo ne postoji! Posljedice ovakvih akcija su potpuno nepredvidljive i često se završavaju potpunim krahom programa ili operativnog sistema. Dobri kompjajleri mogu uočiti većinu situacija u kojima ste eventualno napravili viseću referencu i prijaviti grešku prilikom prevođenja, ali postoje i situacije koje ostaju nedetektirane od strane kompjajlera, tako da dodatni oprez nije na odmet.

Pored običnih referenci postoje i *reference na konstante* (ili *reference na konstante objekte*). One se deklariraju isto kao i obične reference, uz dodatak ključne riječi “**const**” na početku. Reference na konstante se također vezuju za objekat kojim su inicijalizirane, ali se nakon toga ponašaju kao konstantni objekti, odnosno pomoću njih nije moguće mijenjati vrijednost vezanog objekta. Na primjer, neka su date sljedeće deklaracije:

```
int a = 5;
const int &b = a;
```

Referenca “**b**” će se vezati za promjenljivu “**a**”, tako da će pokušaj ispisu promjenljive “**b**” ispisati vrijednost “5”, ali pokušaj promjene vezanog objekta pomoću naredbe

```
b = 6;
```

doveće do prijave greške. Naravno, promjenljiva “**a**” nije time postala konstanta: ona se i dalje može direktno mijenjati (ali ne i indirektno, preko reference “**b**”). Treba uočiti da se prethodne deklaracije bitno razlikuju od sljedećih deklaracija, u kojoj je “**b**” obična konstanta, a ne referenca na konstantni objekat:

```
int a = 5;
const int b = a;
```

Naime, u posljednjoj deklaraciji, ukoliko promjenljiva “**a**” promijeni vrijednost recimo na vrijednost “6”, vrijednost konstante “**b**” i dalje ostaje “5”. Sasvim drugačiju situaciju imamo ukoliko je “**b**” referenca na konstantu: promjena vrijednosti promjenljive “**a**” ostavlja identičan efekat na referencu “**b**”, s obzirom da je ona vezana na objekat “**a**”. Dakle, svaka promjena promjenljive “**a**” odražava se na referencu “**b**”, mada se ona, tehnički gledano, ponaša kao konstantan objekat!

Postoji još jedna bitna razlika između prethodnih deklaracija. U slučaju kada “**b**” nije referenca, u nju se prilikom inicijalizacije kopira *čitav sadržaj* promjenljive “**a**”, dok se u slučaju kada je “**b**” referenca, u nju kopira *samo adresu* mjesta u memoriji gdje se vrijednost promjenljive “**a**” čuva, tako da se pristup vrijednosti promjenljive “**a**” putem reference “**b**” obavlja *indirektno*. U slučaju da promjenljiva “**a**” nije nekog jednostavnog tipa poput “**int**”, nego nekog masivnog tipa koji zauzima veliku količinu memorije (takve tipove ćemo upoznati u kasnijim poglavljima), kopiranje čitavog sadržaja može biti zahtjevno, tako da upotreba referenci može znatno povećati efikasnost.

Činjenica da se reference na konstante ne mogu iskoristiti za promjenu objekta za koji su vezane omogućava da se reference na konstante mogu vezati i za konstante, brojeve, pa i proizvoljne izraze. Tako su, na primjer, sljedeća deklaracije sasvim legalne:

```
int a = 5;
const int &b = 3 * a + 1;
const int &c = 10;
```

Također, reference na konstante mogu se vezati za objekat koji nije istog tipa, ali za koji postoji automatska pretvorba u tip koji odgovara referenci. Na primjer:

```
int p = 5;
const double &b = p;
```

Interesantna stvar nastaje ukoliko referencu na konstantu upotrijebimo kao *formalni parametar funkcije*. Na primjer, pogledajmo sljedeću funkciju:

```
double Kvadrat(const double &x) {  
    return x * x;  
}
```

Kako se referenca na konstantu može vezati za proizvoljan izraz (a ne samo za l-vrijednosti), sa ovako napisanom funkcijom pozivi poput

```
cout << Kvadrat(3 * a + 2);
```

postaju potpuno legalni. Praktički, funkcija "Kvadrat" može se koristiti na posve isti način kao da se koristi prenos parametara po vrijednosti (jedina formalna razlika je u činjenici da bi eventualni pokušaj promjene vrijednosti parametra "x" unutar funkcije "Kvadrat" doveo do prijave greške, zbog činjenice da je "x" referenca na konstantu, što također znači i da ovakva funkcija ne može promijeniti vrijednost svog stvarnog parametra). Ipak, postoji suštinska tehnička razlika u tome šta se zaista internu dešava u slučaju kada se ova funkcija pozove. U slučaju da kao stvarni parametar upotrijebimo neku l-vrijednost (npr. promjenljivu) kao u pozivu

```
cout << Kvadrat(a);
```

dešavaju se iste stvari kao pri klasičnom prenosu po referenci: formalni parametar "x" se poistovjećuje sa promjenljivom "a", što se ostvaruje prenosom adrese. Dakle, ne dolazi ni do kakvog kopiranja vrijednosti. U slučaju kada stvarni parametar nije l-vrijednost (što nije dozvoljeno kod običnog prenosa po referenci), automatski se kreira privremena promjenljiva koja se inicijalizira stvarnim parametrom, koja se zatim klasično prenosi po referenci, i uništava čim se poziv funkcije završi (činjenica da će ona biti uništена ne smeta, jer njena vrijednost svakako nije mogla biti promijenjena putem reference na konstantu). Drugim riječima, poziv

```
cout << Kvadrat(3 * a + 2);
```

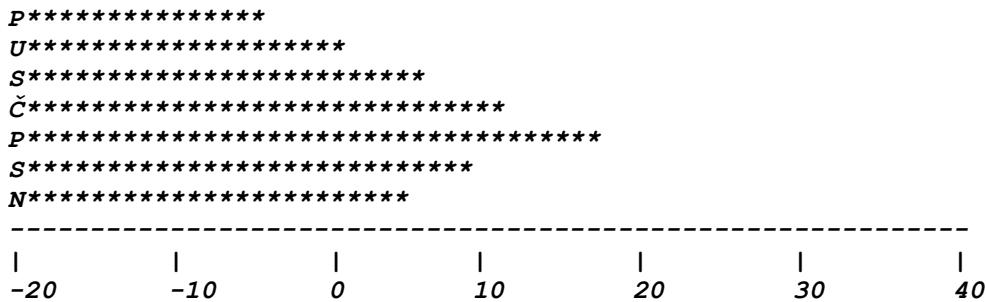
načelno je ekvivalentan pozivu

```
{  
    const double _privremena_ = 3 * a + 2;  
    cout << Kvadrat(_privremena_);  
}
```

Formalni parametri koji su reference na konstantu koriste se uglavnom kod rada sa parametrima masivnih tipova, što ćemo obilato koristiti u kasnijim poglavlјima. Naime, prilikom prenosa po vrijednosti uvijek dolazi do kopiranja stvarnog parametra u formalne, što je neefikasno za slučaj masivnih objekata. Kod prenosa po referenci do ovog kopiranja ne dolazi, a upotreba reference na konstantu dozvoljava da kao stvarni argument upotrijebimo proizvoljan izraz (slično kao pri prenosa po vrijednosti). Stoga, u slučaju masivnih objekata, prenos po vrijednosti treba koristiti samo ukoliko zaista želimo da formalni parametar bude kopija stvarnog parametra (npr. ukoliko je unutar funkcije neophodno mijenjati vrijednost formalnog parametra, a ne želimo da se to odrazi na vrijednost stvarnog parametra).

18. Nizovi i vektori kao parametri

Nizovi se također mogu koristiti kao parametri funkcija (ovdje ne mislimo na *individualne elemente nizova*, nego na *čitave nizove*). Ipak, prilikom prenosa nizova u funkcije dolazi do izvjesnih specifičnosti zbog kojih je ovoj temi posvećeno posebno poglavlje. Prije nego što detaljno razmotrimo te specifičnosti, prikazaćemo prvo jedan primjer koji ilustrira upotrebu nizova kao parametara. Pretpostavimo da neka meteorološka stanica svaki dan registrira srednju vrijednost temperature za taj dan, zaokruženu na najbliži cijeli broj. Potprogram "CrtajDijagram" u sljedećem programu prima kao ulaz parametar "temperature", koji sadrži srednje vrijednosti temperature u toku jedne sedmice, i prikazuje te podatke u obliku jednostavnog linijskog dijagrama tako što štampa niz zvjezdica na ekran. Ovaj parametar je tipa *niz od sedam cijelih brojeva*. Opseg dozvoljenih temperatura je od -20 do +40. U programu je napisana glavna funkcija koja traži od korisnika unos podataka sa tastature, a zatim ih proslijeđuje potprogramu. Na primjer, ako se unesu podaci -5, 0, 5, 10, 16, 8 i 4, ispis na ekran izgledaće ovako:



U programu treba malo pripaziti na razmake, da bi ispis izgledao tačno onako kao što je traženo:

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

using namespace std;

void CrtajDijagram(int temperature[7]) {
    const char imena_dana[7] = {'P', 'U', 'S', 'Č', 'P', 'S', 'N'};
    for(int i = 0; i < 7; i++)
        cout << imena_dana[i] << setfill('*')
            << setw(temperature[i] + 20) << " " << endl;
    cout << setfill('-') << setw(61) << " " << endl;
    for(int i = 1; i <= 7; i++) cout << "| " ; // 9 razmaka
    cout << "\n-20      -10      0      " 
        "10      20      30      40\n";
}

int main() {
    int temp[7];
    for(int i = 0; i < 7; i++) {
        cout << "Unesi temperaturu za " << i + 1 << ". dan: ";
        cin >> temp[i];
    }
    cout << endl;
    CrtajDijagram(temp);
    return 0;
}
```

U ovom primjeru, formalni parametar "temperature" deklariran je kao niz od sedam cijelih brojeva.

Međutim, interesantno je napomenuti da je dimenzija navedena u deklaraciji formalnog parametra *potpuno nebitna*, i kompjajler je zapravo ignorira. Dimenzija formalnog parametra uvijek će biti jednaka dimenziji odgovarajućeg stvarnog parametra, bez obzira na način kako je formalni parametar deklariran. Stoga bi isti potprogram "CrtajDijagram" posve ispravno radio čak i da mu je zaglavlje izgledalo na primjer ovako:

```
void CrtajDijagram(int temperature[3])
```

S obzirom da se dimenzija navedena u deklaraciji formalnog parametra ignorira, nju je moguće potpuno izostaviti (ali ne i uglaste zgrade, jer one označavaju da je formalni parametar niz). Stoga je zaglavlje potprograma "CrtajDijagram" moglo izgledati i ovako:

```
void CrtajDijagram(int temperature[])
```

Ovdje prosto kažemo da je parametar "temperatura" tipa *niz cijelih brojeva, neodređene dimenzije*. U slučaju kada se kao stvarni parametri potprogramu uvijek zadaju nizovi istih dimenzija, tada je dobro istu dimenziju navesti i u formalnom parametru, da bi se jasnije naznačilo da potprogram očekuje nizove baš takve dimenzije (ovim se poboljšava jasnoća i čitljivost programa). S druge strane, dimenzija se u deklaraciji formalnog parametra tipično izostavlja ukoliko se potprogram poziva navodeći kao stvarne parametre nizove *različitih dimenzija*, o čemu ćemo govoriti nešto kasnije.

Na ovom mjestu je neophodno naglasiti da početnici često mijesaju pozive poput

```
CrtajDijagram(temp);
```

i

```
CrtajDijagram(temp[i]);
```

U prvom slučaju, stvarni parametar je *niz*, a u drugom slučaju stvarni parametar je *jedan konkretan element niza*, tj. *cijeli broj*. Kako je u funkciji "CrtajDijagram" formalni parametar tipa *niz*, slijedi da je samo prvi poziv legalan. Isto tako, neispravan je i poziv

```
CrtajDijagram(temp[]);
```

jer se prazne uglaste zgrade mogu upotrijebiti samo u *deklaraciji formalnog parametra*, a ne i u pozivu funkcije (tj. u stvarnom parametru). S druge strane, kao što smo već vidjeli ranije, stvarni parametar oblika "temp[i]" može se legalno upotrijebiti u bilo kojoj funkciji koja ima *cijeli broj kao formalni parametar*.

Prilikom prenosa nizova u funkcije karakteristično je da se oni ponašaju *kao da se prenose po referenci*, bez obzira što odgovarajući formalni parametar nije deklariran kao referenca. Da bismo ovo demonstrirali, razmotrimo sljedeći program:

```
#include <iostream>
using namespace std;
void Potprogram(int a[]) {
    a[0] = 20;
}
int main() {
    int niz[5];
    niz[0] = 10;
    cout << niz[0] << endl;
```

```

Potprogram(niz);
cout << niz[0] << endl;
return 0;
}

```

Ovaj program će ispisati brojeve 10 i 20, iz čega zaključujemo da se promjena elementa “`a[0]`” formalnog parametra “`a`” odrazila na promjeni odgovarajućeg elementa “`niz[0]`” stvarnog parametra “`niz`”. Odavde izgleda da je stvarni parametar “`niz`” zapravo prenesen *po referenci*. Možemo prihvati da se nizovi uvijek prenose po referenci, mada je, tehnički gledano, ovo samo iluzija. Naime, u poglavljiju koje govori o pokazivačima vidjećemo da je ovakvo ponašanje zapravo posljedica činjenice da se ime niza upotrijebljeno samo za sebe automatski konvertira u pokazivač na prvi element niza. Tako se, u pozivu “`Potprogram(niz)`”, ime niza “`niz`” upotrijebljeno samo za sebe automatski konvertira u adresu njegovog prvog elementa, tako da funkcija zapravo samo dobija adresu, kao da je formalni parametar referenca. Ako zanemarimo ove tehničke detalje, možemo smatrati da se nizovi u funkcije *uvijek prenose po referenci*, mada ovo nije potpuno tačno (slučajevi u kojima je neophodno znati šta se tačno dešava toliko su specifični da se njima ovdje nećemo baviti). Principijelno je moguće formalni parametar deklarirati i kao *referencu na niz*, odnosno zaglavje funkcije “`CrtajDijagram`” moglo je izgledati i ovako:

```
void CrtajDijagram(int (&temperature) [7])
```

U ovom slučaju, dimenzija se *ne smije izostaviti*. Jedina vidljiva razlika u ovom slučaju bila bi što bi se ovakva funkcija mogla pozvati samo sa stvarnim parametrom koji je *niz od sedam elemenata* (a ne neke druge dimenzije), s obzirom da se reference mogu vezati samo za objekat potpuno identičnog tipa. Ovakva deklaracija se ipak prilično rijetko koristi (sa tehničkog aspekta, ona uvodi dodatni nivo indirekcije, što donekle smanjuje efikasnost).

Pošto se nizovi prilikom prenosa u funkcije ponašaju kao da su preneseni po referenci, funkcija može promijeniti njihov sadržaj. Ukoliko funkcija ne treba da mijenja sadržaj niza koji joj je prenesen kao parametar, odgovarajući formalni parametar *treba deklarirati sa prefiksom “const”*. Ovim postižemo tri efekta. Prvo, onome ko analizira program biće jasno da funkcija neće (i ne može) promijeniti sadržaj proslijedenog niza, što poboljšava čitljivost i razumljivost programa. Drugo, svaki nehotični pokušaj promjene sadržaja niza unutar funkcije doveće do prijave greške od strane kompjajlera. Treće, ovakva funkcija će se moći pozvati i ukoliko je stvarni parametar *konstantni niz*. Ovo ne bi bilo moguće da formalni parametar nije deklariran sa prefiksom “`const`”, jer u suprotnom ne bi postojale garancije da funkcija neće promijeniti sadržaj stvarnog parametra (što nije dozvoljeno ukoliko je stvarni parametar konstantni niz). Stoga, preporučeni oblik zaglavja potprograma “`CrtajDijagram`” izgleda ovako:

```
void CrtajDijagram(const int temperature[7])
```

Već je rečeno da je u potprograme moguće prenositi nizove različite dužine. Međutim, tom prilikom se javlja jedan praktičan problem. Pretpostavimo da želimo napisati potprogram koji ispisuje na ekran elemente nekog niza cijelih brojeva, razdvojene razmacima. Ukoliko niz uvijek ima isti broj elemenata (npr. 10), odgovarajuća funkcija bi mogla izgledati ovako:

```

void IspisiNiz(const int niz[]) {
    for(int i = 0; i < 10; i++) cout << niz[i] << " ";
}

```

Ovakvoj funkciji moguće je kao stvarni parametar navesti niz sa proizvoljnim brojem elemenata. Međutim, “`for`” petlja je napisana sa fiksnom gornjom granicom 10, tako da u slučaju da kao parametar upotrijebimo niz sa više od 10 elemenata, biće ispisano samo prvih deset elemenata (u slučaju da niz ima

manje od 10 elemenata, biće ispisano "smeće", s obzirom da će indeks niza izaći izvan dozvoljenog opsega). Sljedeća verzija funkcije "IspisiNiz" predstavlja *pokušaj* da ovaj problem riješimo tako što ćemo pokušati primijeniti trik sa "**sizeof**" operatorom za utvrđivanje broja elemenata niza:

```
void IspisiNiz(const int niz[]) {
    const int br_elemenata = sizeof niz / sizeof niz[0];
    for(int i = 0; i < br_elemenata; i++) cout << niz[i] << " ";
```

Međutim, ovako napisana funkcija *ne radi ispravno*. Naime, "**sizeof**" operator ne daje ispravan rezultat ukoliko se primijeni na formalni parametar tipa niza. Ovo je također posljedica činjenice da se pri pozivu funkcije ime niza upotrijebljenog kao stvarni parametar automatski konvertira u pokazivač na prvi element niza. Pri ovoj konverziji gubi se informacija o veličini niza koji je stvarni parametar, tako da funkcija uopće ne dobija ovu informaciju. Jedno moguće rješenje je umjesto nizova koristiti *vektore*, kod kojih se pomenuta informacija ne gubi. Ovakvo rješenje demonstriraćemo kasnije. U slučaju da moramo koristiti nizove, jedino rješenje je prenijeti informaciju o broju elemenata niza kao *dodatni parametar*. Takva funkcija mogla bi izgledati ovako:

```
void IspisiNiz(const int niz[], int br_elemenata) {
    for(int i = 0; i < br_elemenata; i++) cout << niz[i] << " ";
```

Tako, na primjer, ukoliko imamo dva niza "a" i "b", od kojih prvi ima 10 a drugi 20 elemenata, za njihov ispis na ekran možemo koristiti sljedeće pozive:

```
IspisiNiz(a, 10);
IspisiNiz(b, 20);
```

Alternativno, mogući su i pozivi poput sljedećih, što može biti korisno ukoliko prilikom razvoja programa često mijenjamo veličinu nizova:

```
IspisiNiz(a, sizeof a / sizeof a[0]);
IspisiNiz(b, sizeof b / sizeof b[0]);
```

Opisano rješenje, u kojem se veličina niza prenosi kao dodatni parametar, uopće nije toliko nelegantno koliko bi se moglo pomisliti na prvi pogled. Naime, upotreba dopunskog parametra nudi i dodatnu prednost da je moguće zadati broj elemenata nad kojim funkcija treba obaviti svoj zadatak, koji ne mora nužno biti jednak dimenziji niza. Na primjer, ukoliko želimo ispisati samo prvih pet elemenata niza "a", možemo koristiti poziv

```
IspisiNiz(a, 5);
```

bez obzira što niz "a" ima 10 elemenata. Možemo dodati i preklopljenu verziju funkcije "IspisiNiz" kojoj je moguće zadati i element od kojeg počinje ispis:

```
void IspisiNiz(const int niz[], int odakle, int br_elemenata) {
    for(int i = odakle; i < odakle + br_elemenata; i++)
        cout << niz[i] << " ";
```

Tako, ukoliko želimo ispisati 5 elemenata niza "a" počev od elementa sa indeksom 3, koristićemo poziv

```
IspisiNiz(a, 3, 5);
```

Primijetimo da smo u ovom slučaju morali koristiti preklapanje, a ne parametre sa podrazumijevanom vrijednošću, pošto se verzija funkcije sa dva parametra ponaša kao da je u njoj izostavljen *drugi*, a ne *treći* parametar (sa podrazumijevanom vrijednošću 0). Naravno, parametar sa podrazumijevanom vrijednošću mogao bi se koristiti ukoliko bismo se odlučili da parametar kojim se zadaje indeks početnog elementa bude *posljednji* a ne drugi parametar.

Sljedeći primjer također ilustrira činjenicu da je često poželjno imati dodatni parametar kojim se određuje broj elemenata nad kojim treba obaviti određenu operaciju. U priloženom programu definirana je funkcija "HarmonijskaSredina" koja računa harmonijsku sredinu brojeva iz niza koji je zadan kao prvi parametar, pri čemu je drugi parametar broj elemenata u nizu nad kojim treba izračunati harmonijsku sredinu. Podsjetimo se da se harmonijska sredina definira kao recipročna vrijednost aritmetičke sredine recipročnih vrijednosti niza brojeva, odnosno

$$H(a_1, a_2, \dots a_N) = \frac{N}{\frac{1}{a_1} + \frac{1}{a_2} + \dots + \frac{1}{a_N}}$$

Broj elemenata niza, kao i sami elementi niza unose se sa tastature. S obzirom da broj elemenata niza nije unaprijed poznat, niz u glavnom programu je deklariran sa dimenzijom "MaxBroj", gdje je "MaxBroj" konstanta za koju se pretpostavlja da je veća od maksimalnog očekivanog broja elemenata niza. U ovom primjeru se vidi neophodnost prenosa informacije o broju elemenata za koje treba izračunati harmonijsku sredinu u funkciju. Naime, čak i kada bi funkcija nekako sama uspjela da odredi informaciju o dimenziji niza, ta dimenzija bi svakako bila *deklarirana dimenzija* "MaxBroj", a ne broj elemenata koji *zaista koristimo*, što funkcija nikako ne može saznati ukoliko joj tu informaciju ne proslijedimo eksplicitno:

```
#include <iostream>

const int MaxBroj(50); // Maksimalan broj brojeva

double HarmonijskaSredina(const double a[], int n) {
    double suma(0);
    for(int i = 0; i < n; i++) suma += 1 / a[i];
    return n / suma;
}

int main() {
    int n;
    cout << "Koliko elemenata ima niz (max. " << MaxBroj << ")? ";
    cin >> n;
    cout << "Unesite elemente niza:\n";
    double niz[MaxBroj];
    for(int i = 0; i < n; i++) cin >> niz[i];
    cout << "Harmonijska sredina iznosi "
        << HarmonijskaSredina(niz, n) << endl;
    return 0;
}
```

Prenos nizova u funkcije putem parametara podliježe mnogo jačim ograničenjima nego prenos prostih tipova podataka. Na primjer, dok će funkcija koja očekuje formalni parametar tipa "**double**" bez problema prihvati stvarni parametar tipa "**int**" (osim ukoliko je prenos po referenci), s obzirom da postoji automatska konverzija iz tipa "**int**" u tip "**double**", funkcija koja očekuje formalni parametar čiji je tip niz elemenata tipa "**double**" neće prihvati kao stvarni parametar niz elemenata tipa "**int**". Drugim riječima, poziv poput

```
HarmonijskaSredina(niz, n)
```

neće biti prihvaćen ukoliko je niz “niz” niz cijelih brojeva (niti bilo kakav niz čiji elementi nisu tipa “**double**”). Ovo ograničenje nije vezano samo za prenos parametara: niz elemenata nekog tipa “tip1” nikada se ne konvertira automatski u niz elemenata tipa “tip2”. Ne samo da nije podržana *automatska konverzija*, već se konverzija ne može izvršiti ni *eksplicitno* primjenom operatora za konverziju tipa. Na primjer, poziv poput sljedećeg također neće biti prihvaćen:

```
HarmonijskaSredina ((double [])niz, n)
```

Postoje veoma jaki razlozi zbog čega nisu podržane konverzije nizova sa jednim tipom elemenata u niz sa drugim tipom elemenata. Naime, da su podržane, takve konverzije bile bi izuzetno neefikasne. Na primjer, relativno je lako pretvoriti podatak tipa “**int**” u tip “**double**”: dovoljno je binarnu reprezentaciju jednog cijelog broja u memoriji preorganizirati u binarni zapis odgovarajućeg realnog broja u sistemu pokretnog zareza. S druge strane, da bi se niz cijelih brojeva konvertirao u niz realnih brojeva, opisanu konverziju trebalo bi primijeniti *element po element* na sve elemente niza. Jasno je da, u slučaju da niz ima npr. 10000 elemenata, ovakva konverzija može biti dugotrajna. Međutim, ovo je tek manji dio problema. Različiti tipovi zauzimaju različitu količinu memorije (npr. kod većine današnjih kompjajlera podatak tipa “**int**” zauzima 4 bajta, a podatak tipa “**double**” 8 bajtova). Kako se elementi nizova u memoriju smještaju kontinualno, jedan za drugim, to bi konverzija svakog elementa niza u tip drugačije veličine zahtijevala *pomjeranje u memoriji svih elemenata niza koji slijede iza njega*, npr. da bi se dobio prostor za konvertirani element koji zauzima veći prostor. Zbog svih navedenih razloga, konverzija nizova sa jednim tipom elemenata u niz sa drugim tipom elemenata ne može se izvesti efikasno, te su konstruktori jezika C++ odlučili da je uopće ne podrže.

Ukoliko nam zaista treba da neku funkciju pozivamo prenoseći kao parametre nizove različitih tipova, jedno moguće rješenje je korištenje preklapanja funkcija po tipu. Tako je, na primjer, ukoliko funkcija “HarmonijskaSredina” treba raditi kako sa nizovima realnih brojeva tako i sa nizovima cijelih brojeva, moguće je napraviti još jednu preklopljenu verziju iste funkcije koja kao parametar prima niz cijelih brojeva. Tijelo ove funkcije biće praktično identično kao u slučaju napisane funkcije koja prima niz relanih brojeva (osim što bismo cijelobrojnu konstantu “1” trebali zamijeniti sa realnom konstantom “1.” da izbjegnemo cijelobrojno dijeljenje). Nedostatak ovog pristupa je što bismo morali pisati po jednu preklopljenu funkciju za svaki tip elemenata niza koji bismo eventualno htjeli koristiti. Mechanizam generičkih funkcija, koji ćemo opisati u narednim poglavljima, nudi elegantnije rješenje za ovaj problem.

Prenos nizova kao parametara ilustriraćemo na još nekoliko korisnih primjera. Česta situacija je da u nekom nizu treba pronaći *na kojoj poziciji* u nizu se nalazi neki zadani element (ovo je primjer elementarnog problema iz kategorije tzv. *problema pretrage*). U sljedećem programu definirana je funkcija “NadjElement” koja pretražuje niz koji je zadan kao prvi parametar u potrazi za elementom čija se vrijednost zadaje kao drugi parametar. Treći parametar je broj elemenata u nizu. U slučaju da se element pronađe, funkcija vraća njegov indeks kao rezultat (odnosno indeks *prvog pojavljivanja zadanog elementa* u slučaju da se on javlja više puta), a u suprotom vraća -1 kao rezultat (ova konvencija je usvojena s obzirom na činjenicu da -1 ne može biti legalan indeks niti jednog elementa koji postoji u nizu). Radi preglednosti, u programu je definirana konstanta sa imenom “**NEMA_GA**”, čija je vrijednost upravo -1. Naime, mnogo je čitljivije u programu koristiti simboličko ime “**NEMA_GA**” kao indikator da element nije nađen, nego brojčanu vrijednost “-1”, čije značenje nije jasno na prvi pogled, i koja se može koristiti i za potrebe nevezane od samog problema pretrage. Glavna funkcija u programu ilustrira kako se definirana funkcija “NadjElement” može koristiti:

```
#include <iostream>
const int NEMA_GA(-1);
```

```

int NadjiElement(const int lista[], int element, int n) {
    for(int i = 0; i < n; i++)
        if(lista[i] == element) return i;
    return NEMA_GA;
}

int main() {
    int test[] = {2, 5, 3, 7, 4, 5, 1, 11, 8, 4};
    const int broj_elemenata = sizeof test / sizeof test[0];
    cout << "Unesite element koji se traži: ";
    cin >> element;
    int gdje_je = NadjiElement(test, element, broj_elemenata);
    if(gdje_je == NEMA_GA) cout << "Traženog elementa nema u nizu!";
    else cout << "Traženi element se nalazi na poziciji << gdje_je";
    return 0;
}

```

Princip rada funkcije “NadjiElement” je očigledan: elementi niza “lista” pretražuju se jedan po jedan, i ukoliko se ustanovi jednakost vrijednosti elementa niza sa traženom vrijednošću, funkcija se *odmah prekida*, uz vraćanje indeksa tekućeg elementa kao rezultat. Vrijednost “NEMA_GA” vraća se ukoliko su svi elementi niza ispitani, a traženi element nije nađen.

Mnoge od dosada izloženih koncepcata ilustrira i sljedeći primjer, u kojem je definirana funkcija “DaLiSuJednaki” koja određuje da li su dva niza koji su joj proslijedeni kao parametri jednak (tj. da li su im svi odgovarajući elementi sa istim indeksima jednak) ili nisu, pri čemu treći parametar predstavlja broj elemenata nizova (za koji pretpostavljamo da je isti za oba niza). Možemo primijetiti da se izvršavanje funkcije prekida čim se pronađe prvi par odgovarajućih elemenata koji su različiti, jer u tom slučaju pouzdano znamo da nizovi nisu jednak, bez potrebe da ispitujemo ostatak elemenata. S druge strane, eventualnu jednakost nizova možemo utvrditi *tek nakon što ispitamo sve elemente*:

```

#include <iostream>
using namespace std;

bool DaLiSuJednaki(const int a[], const int b[], int n) {
    for(int i = 0; i < n; i++)
        if(a[i] != b[i]) return false;
    return true;
}

int main() {
    const int Max(1000);
    int a[Max], b[Max], n;
    cout << "Koliko elemenata imaju nizovi (max. " << Max << ")? ";
    cin >> n;
    cout << "Unesite prvi niz:\n";
    for(int i = 0; i < n; i++) cin >> a[i];
    cout << "Unesite drugi niz:\n";
    for(int i = 0; i < n; i++) cin >> b[i];
    if(DaLiSuJednaki(a, b, n)) cout << "Nizovi su jednak\n";
    else cout << "Nizovi su različiti\n";
    return 0;
}

```

Naročito grubu grešku, koja se veoma često može susresti kod početnika, predstavlja pisanje funkcija poput sljedeće:

```
bool DaLiSuJednaki(const int a[], const int b[], int n) {
```

```

    for(int i = 0; i < n; i++)
        if(a[i] != b[i]) return false;
        else return true;
}

```

Ovdje je greška u tome što se u svakom slučaju izvršavanje funkcije *prekida već nakon upoređivanja prvih elemenata niza*. Naime, bilo da je uvjet ispunjen bilo da nije ispunjen, nailazi se na naredbu “**return**” koja prekida izvršavanje funkcije, tako da se petlja uopće neće nastaviti!

Činjenica da se nizovi prilikom prenosa putem parametara u funkcije ponašaju kao da se prenose po referenci omogućava funkcijama da promijene sadržaj nizova koji su im proslijedeni kao parametri. Ova činjenica se često može korisno upotrijebiti. Na primjer, u sljedećem programu definirana je funkcija “NadjiProsteBrojeve”, koja puni niz koji joj je proslijeden kao prvi parametar sa prvih n prostih brojeva, gdje je n drugi parametar. Dakle, ovdje je prvi parametar tipični primjer *izlaznog parametra*: njegov sadržaj prije poziva funkcije je nebitan, a nakon poziva funkcije on sadrži korisne informacije. Funkcija je zasnovana na sljedećem algoritmu. Broj 2 je prost broj, pa odmah kao prvi element niza upisujemo 2. Nakon toga krećemo od broja 3 i analiziramo svaki neparan broj. Pri ispitivanju da li je broj prost, dovoljno je ispitati da li je broj djeljiv samo sa prostim brojevima manjim od njega, a kako brojeve ispitujemo po redu, svi prosti brojevi manji od njega *već se nalaze u nizu*. Pri svakom pronađenom prostom broju povećavamo brojač prostih brojeva za 1, i ponavljamo postupak dok brojač ne dostigne željeni broj prostih brojeva n :

```

#include <iostream>
using namespace std;

void NadjiProsteBrojeve(int niz[], int n) {
    int brojac(1), tekuci(3);
    niz[0] = 2;
    while(brojac < n) {
        bool da_li_je_prost(true);
        for(int i = 0; i < brojac; i++)
            if(tekuci % niz[i] == 0) {
                da_li_je_prost = false;
                break;
            }
        if(da_li_je_prost) niz[brojac++] = tekuci;
        tekuci += 2;
    }
}

int main() {
    const int Max(1000);
    int n, a[Max];
    cout << "Koliko zelite prostih brojeva (max. " << Max << ")? ";
    cin >> n;
    NadjiProsteBrojeve(a, n);
    for(int i = 0; i < n; i++) cout << a[i] << " ";
    return 0;
}

```

Ako vas zbujuje tipična C++ konstrukcija

```
niz[brojac++] = tekuci;
```

zamijenite je sa sljedećom grupom naredbi (ne zaboravite pri tom vitičaste zagrade, da objedinite ove naredbe u blok):

```
niz[brojac] = tekuci;
brojac++
```

Također je interesantno da se, zahvaljujući fleksibilnosti “**for**” petlje u jeziku C++, “**while**” petlja u funkciji “NadjiProsteBrojeve” može reformulirati kao “**for**” petlja, čime funkcija postaje neznatno kraća, ali i nešto teža za razumijevanje:

```
void NadjiProsteBrojeve(int niz[], int n) {
    niz[0] = 2;
    for(int brojac = 1, tekuci = 3; brojac < n; tekuci += 2) {
        bool da_li_je_prost(true);
        for(int i = 0; i < brojac; i++)
            if(tekuci % niz[i] == 0) {
                da_li_je_prost = false;
                break;
            }
        if(da_li_je_prost) niz[brojac++] = tekuci;
    }
}
```

U sljedećem primjeru demonstrirana je još jedna funkcija kod koje je upotrijebljen parametar nizovnog tipa kao čisto izlazni parametar. Naime, u programu koji slijedi, definirana je funkcija “NadjiFibonacijeveBrojeve” koja puni niz koji joj je proslijeden kao prvi parametar sa prvih n Fibonačijevih brojeva, gdje je n drugi parametar (podsjetimo se da su Fibonačijevi brojevi F_k definirani relacijama $F_0=F_1=1$, $F_k=F_{k-1}+F_{k-2}$ za $k>1$). Primijetimo da je ovaj program mnogo razumljiviji nego sličan program koji smo pisali prije uvođenja pojma *nizova* (u poglavlju o petljama), jer direktno koristi formulu kojom se definiraju Fibonačijevi brojevi. S druge strane, ovaj program je znatno *neracionalniji*, jer nepotrebno troši memoriju na rezerviranje prostora za niz:

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

void NadjiFibonacijeveBrojeve(int f[], int n) {
    f[0] = f[1] = 1;
    for(int i = 2; i <= n; i++) f[i] = f[i-1] + f[i-2];
}

int main() {
    const int Max(1000);
    int n, fib[Max];
    cout << "Koliko zelite brojeva (max, " << Max << ")? ";
    cin >> n;
    NadjiFibonacijeveBrojeve(fib, n);
    for(int i = 0; i < n; i++) cout << fib[i] << " ";
    return 0;
}
```

Sljedeći primjer koristi funkciju koja posjeduje bočni efekat koji se manifestira *ispisom određenih podataka na ekran*, a koja pored toga vraća *vrijednost*. Ovaj primjer je veoma interesantan, između ostalog i zbog algoritma koji je korišten za realizaciju funkcije. Funkcija “StampajBezDuplikata”, definirana u narednom programu, štampa sve elemente iz niza koji je zadan kao prvi parametar (drugi parametar je broj elemenata niza), izostavljajući elemente koji se ponavljaju, i vraća *kao rezultat* broj

elemenata koji se ponavljaju. Na primjer, ako je niz sadržavao elemente 6, 20, -3, 6, 6, 20, 3, 14, 101, 20 i 21, ova funkcija će *odštampati* sekvencu brojeva “6 20 -3 3 14 101 21”, i *vratiti* kao rezultat 2, jer se dva broja u nizu ponavljaju (to su brojevi 6 i 20):

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

int StampajBezDuplikata(const int niz[], int broj_elemenata) {
    int broj_duplikata(0);
    for(int i = 0; i < broj_elemenata; i++) {
        int broj_ponavljanja(0);
        for(int j = 0; j < i; j++)
            if(niz[i] == niz[j]) broj_ponavljanja++;
        if(broj_ponavljanja == 0) cout << niz[i] << " ";
        if(broj_ponavljanja == 1) broj_duplikata++;
    }
    cout << endl;
    return broj_duplikata;
}

int main() {
    const int Max(1000);
    int n, a[Max];
    cout << "Unesite broj elemenata niza (max. " << Max << "): ";
    cin >> n;
    cout << "Unesite elemente niza:\n";
    for(int i = 0; i < n; i++) cin >> a[i];
    cout << "Niz bez elemenata koji se ponavljaju glasi: ";
    int broj_duplikata = StampajBezDuplikata(a, n);
    cout << "Broj elemenata koji se ponavljaju je " << broj_duplikata;
    return 0;
}
```

Razmotrimo prvo kako radi funkcija “StampajBezDuplikata”. Spoljašnja “**for**” petlja prolazi kroz sve elemente niza. Unutar nje se nalazi druga “**for**” petlja koja za svaki element niza broji koliko se on puta već pojavio ranije, upoređujući ga sa prethodnim elementima i uvećavajući pri tom promjenljivu “*broj_ponavljanja*” za 1 svaki put kada se ustanovi podudarnost (ona se inicijalizira na nulu svaki put na početku petlje). Ako ustanovimo da se element pojavio prvi put (tj. ako je vrijednost promjenljive “*broj_ponavljanja*” jednaka nuli), ispisujemo ga. Ako se element javljao ranije, uvećavamo brojač duplikata za 1, ali samo kada prvi put uočimo element koji se ponavlja (tj. ukoliko je vrijednost promjenljive “*broj_ponavljanja*” jednak jedinici) – u suprotnom bi se u nizu 3, 3, 3, 3, 3, 3 pogrešno registriralo da se ponavlja 5 elemenata iako se ponavlja samo jedan.

Obratimo sada pažnju na glavni program, u kojem se nalazi naredba

```
int broj_duplikata = StampajBezDuplikata(a, n);
```

Ova naredba radi *dvije stvari*. Prvo, ona *ispisuje* niz bez elemenata koji se ponavljaju (iako to nije očigledno iz same forme naredbe, mada je jasno iz njenog imena – ovo je tipičan primjer kako bočni efekti mogu učiniti poziv funkcije nejasnim u slučaju loše odabranog imena). Drugo, ona *dodjeljuje* promjenljivoj “*broj_duplikata*” vrijednost koju je *vratila* funkcija “StampajBezDuplikata”. U sljedećoj naredbi vrijednost ove promjenljive će biti ispisana, sa ciljem da ispišemo broj elemenata koji se ponavljaju. Zamislimo sada da smo željeli samo da ispišemo niz bez elemenata koji se ponavljaju, a da nas njihov broj *uopće ne zanima*. Tada bismo mogli napisati samo:

```
StampajBezDuplikata(a, n);
```

Ovdje imamo očigledan primjer u kojem *ignoriramo* povratnu vrijednost funkcije, a koristimo samo njen bočni efekat.

Kao i kod svih funkcija sa bočnim efektima, trebamo biti jako oprezni u kakvom kontekstu pozivamo ovu funkciju, jer efekti često mogu da ne budu u skladu sa očekivanjima. Razmislite sami zbog čega naredba

```
cout << "Broj elemenata koji se ponavljaju je "
    << StampajBezDuplikata(a, n);
```

može proizvesti sasvim neočekivane efekte.

U sljedećem primjeru, formalni parametar funkcije “*OdstraniParne*” nizovnog tipa iskorišten je kao *ulazno-izlazni parametar*. Ova funkcija *odstranjuje* iz niza koji je proslijeden kao prvi parametar (drugi parametar predstavlja broj elemenata niza) sve elemente koji su parni brojevi, i *vraća kao rezultat* broj elemenata niza nakon obavljenog odstranjivanja. Drugim riječima, nakon poziva ove funkcije, sadržaj niza proslijedenog kao parametar biće *izmijenjen*. Funkcija radi tako što svaki put kada nađe na element koji je paran broj, smanjuje broj elemenata niza za 1, i premješta sve elemente niza koji slijede iza elementa koji treba odstraniti za jedno mjesto unazad, čime se efektivno briše taj element i popunjava praznina koja bi inače nastala njegovim brisanjem. Napisana funkcija je demonstrirana u kratkom testnom programu. Ovaj primjer je veoma značajan, stoga zahtijeva pažljiviju analizu:

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

int OdstraniParne(int niz[], int n) {
    for(int i = 0; i < n; i++)
        while(niz[i] % 2 == 0 && i < n) {
            n--;
            for(int j = i; j < n; j++) niz[j] = niz[j + 1];
        }
    return n;
}

int main() {
    const int Max(1000);
    int n, a[Max];
    cout << "Unesite broj elemenata niza (max. " << Max << "): ";
    cin >> n;
    cout << "Unesite elemente niza:\n";
    for(int i = 0; i < n; i++) cin >> a[i];
    int novi_n = OdstraniParne(a, n);
    cout << "Nakon odstranjivanja parnih brojeva, niz glasi:\n";
    for(int i=0; i < novi_n; i++) cout << a[i] << endl;
    return 0;
}
```

Obratimo pažnju da u ovom programu, naredba

```
int novi_n = OdstraniParne(a, n);
```

također sadrži bočni efekat: ona ostavlja trajno dejstvo na stvarni parametar “*a*”, što nije posve očigledno

iz načina kako je funkcija pozvana. Već smo ranije naglasili da funkcije koje istovremeno vraćaju vrijednost i koriste izlazne ili ulazno-izlazne parametre (što je upravo slučaj u ovdje napisanoj funkciji) treba izbjegavati ukoliko je ikako moguće.

Jedno od ograničenja prilikom vraćanja rezultata iz funkcije je što funkcije *ne mogu vratiti niz kao rezultat*. Osnovni razlog za ovo ograničenje je činjenica da se nizovi ne mogu međusobno dodjeljivati primjenom operatora “`=`”, tako da ne bi postojao način da se prihvati niz vraćen iz funkcije (tj. ne bi bilo moguće niz vraćen iz funkcije dodijeliti nekoj nizovnoj promjenljivoj). Kao alternativno rješenje može se koristiti *smještanje* niza koji treba proslijediti nazad kao rezultat u neki od dodatnih parametara funkcije. Tako je u sljedećem primjeru definiran novi tip “*Vektor*” (pomoću naredbe “`typedef`”) koji predstavlja trodimenzionalni vektor (koji je, zapravo, niz od tri realna elementa), a zatim je definirana funkcija “*VektorskiProizvod*” koja računa vektorski proizvod prva dva argumenta, i smješta rezultat u treći argument. Prvi i drugi argument pri tome ostaju *neizmjenjeni*. Podsjetimo se da se vektorski proizvod dva vektora zadana preko koordinata računa po formuli

$$\{a_0, a_1, a_2\} \times \{b_0, b_1, b_2\} = \{a_1b_2 - a_2b_1, a_2b_0 - a_0b_2, a_0b_1 - a_1b_0\}$$

Program je dovoljno jednostavan da ne zahtijeva nikakva posebna objašnjenja:

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

typedef double Vektor[3];

void VektorskiProizvod(const Vektor a, const Vektor b, Vektor c) {
    c[0] = a[1] * b[2] - a[2] * b[1];
    c[1] = a[2] * b[0] - a[0] * b[2];
    c[2] = a[0] * b[1] - a[1] * b[0];
}

int main() {
    Vektor a, b, c;
    cout << "Unesi prvi vektor: ";
    cin >> a[0] >> a[1] >> a[2];
    cout << "Unesi drugi vektor: ";
    cin >> b[0] >> b[1] >> b[2];
    VektorskiProizvod(a, b, c);
    cout << "Njihov vektorski proizvod je: "
        << c[0] << " " << c[1] << " " << c[2] << endl;
    return 0;
}
```

Šteta je što se nizovi ne mogu vraćati kao rezultati, jer bi inače bilo moguće formirati funkcije koje bi, sa matematičkog aspekta, imale mnogo elegantniju formu. Na primjer, bilo bi moguće rješenje poput sljedećeg (koje, da naglasimo, *nije legalno*):

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

typedef double Vektor[3];

// Ovo ne radi!
Vektor VektorskiProizvod(const Vektor a, const Vektor b) {
    vektor c;
    c[0] = a[1] * b[2] - a[2] * b[1];
    c[1] = a[2] * b[0] - a[0] * b[2];
```

```

    c[2] = a[0] * b[1] - a[1] * b[0];
    return c;
}

int main() {
    Vektor a, b;
    cout << "Unesi prvi vektor: ";
    cin >> a[0] >> a[1] >> a[2];
    cout << "Unesi drugi vektor: ";
    cin >> b[0] >> b[1] >> b[2];
    Vektor c = VektorskiProizvod(a, b);
    cout << "Vektorski proizvod je: "
        << c[0] << " " << c[1] << " " << c[2] << endl;
    return 0;
}

```

Matematički elegantnije rješenje, slično gore navedenom (koje ne radi) ipak je moguće napraviti koristeći neke druge strukture podataka (kao što su npr. *strukture* i *klase* koje ćemo upoznati kasnije) koje je moguće vraćati kao rezultate iz funkcija. Također je umjesto nizova moguće koristiti *vektore*, koji se, za razliku od nizova, *mogu* vraćati kao rezultati iz funkcija. Jedno takvo rješenje biće demonstrirano kasnije u ovom poglavlju.

U nekim slučajevima je korisno napraviti funkciju koja kao rezultat vraća *referencu na neki element niza*. Da bismo ilustrirali eventualnu primjenu takvih funkcija, razmotrimo prvo primjer sljedeće funkcije, koja kao rezultat vraća najveći element niza realnih brojeva koji je prenesen kao parametar:

```

double MaxElement(const double niz[], int br_elemenata) {
    double max = niz[0];
    for(int i = 1; i < br_elemenata; i++)
        if(niz[i] > max) max = niz[i];
    return max;
}

```

Princip rada ove funkcije je očigledan. Međutim, ista funkcija se mogla napisati i na nešto manje očigledan način, u kojem se prvo pronalazi *indeks najvećeg elementa*, a zatim se nađeni indeks koristi za vraćanje najvećeg elementa:

```

double MaxElement(const double niz[], int br_elemenata) {
    int indeks(0);
    for(int i = 1; i < br_elemenata; i++)
        if(niz[i] > niz[indeks]) indeks = i;
    return niz[indeks];
}

```

Jasno je na koji način se napisane funkcije mogu koristiti (obje verzije imaju isto dejstvo). Međutim, modificiraćemo drugu verziju funkcije tako da umjesto *vrijednosti* najvećeg elementa niza vraća kao rezultat *referencu* na najveći element niza:

```

double &Max_element(const double niz[], int br_elemenata) {
    int indeks(0);
    for(int i = 1; i < br_elemenata; i++)
        if(niz[i] > niz[indeks]) indeks = i;
    return niz[indeks];
}

```

Šta smo postigli ovom modifikacijom? Funkcija “MaxElement” i dalje se može koristiti na iste načine kao i prije ove modifikacije. Međutim, vraćanjem reference, poziv funkcije postaje l-vrijednost, pa

se može koristiti i u kontekstima u kojima se mogu koristiti samo l-vrijednosti. Na primjer, ukoliko je “*a*” neki niz od 10 realnih elemenata, tada će naredba

```
MaxElement(a, 10) = 1000;
```

pronaći najveći element u nizu “*a*”, i *dodijeliti* mu vrijednost “1000” (s obzirom da će se poziv funkcije “*MaxElement(a, 10)*” *poistovijetiti* sa nađenim najvećim elementom. Bitno je naglasiti da slična modifikacija *nije bila moguća* u prvoj verziji funkcije “*MaxElement*” koju smo napisali. Naime, da smo samo promjenili njeno zaglavlje tako da se kao rezultat vraća referenca, kao rezultat bismo zapravo vratili *referencu na lokalnu promjenljivu “max”* koja prestaje postojati po završetku funkcije, odnosno vratili bismo *viseću referencu!* Ovakav pokušaj najvjerovatnije bi bio sankcioniran (prijavom greške) od strane boljeg kompjajlera.

Nakon što smo razmotrili upotrebu nizova kao parametara, recimo na kraju nešto i o upotrebi *vektora* kao parametara. S obzirom da su vektori izvedeni tip koji je tek nedavno ušao u standard jezika C++, i koji je uveden sa ciljem da se prevaziđu izvjesna ograničenja i nedostaci tipova, prirodno je očekivati da upotreba vektora kao parametara rješava neke od problema koje smo uočili prilikom razmatranja upotrebe nizova kao parametara. Zaista, upotrebo vektora kao parametara ostvaruju se sljedeća poboljšanja:

- Za razliku od nizova za koje izgleda da se uvijek prenose po referenci, vektori se mogu prenositi kako po *vrijednosti* (u tom slučaju je formalni parametar kopija vektora proslijedenog kao stvarni parametar, potpuno neovisna od stvarnog parametra), tako i po *referenci*. Prenos po referenci se, kao i obično, zahtijeva deklariranjem formalnog parametra kao reference na vektor. Tipovi elemenata vektora u formalnom parametru i stvarnom parametru se i dalje moraju u potpunosti slagati, kao i u slučaju prenošenja nizova.
- Funkcija može primjenom operacije “*size*” nad formalnim parametrom tipa vektor saznati kolika je bila dimenzija vektora koji joj je proslijeden kao stvarni parametar, tako da nije neophodno prenošenje informacije o dimenziji vektora kroz dodatni parametar.
- Funkcije mogu bez ikakvih problema vraćati vektore kao rezultat.

Kao primjer, napisaćemo funkciju koja ispisuje sve elemente nekog vektora cijelih brojeva na ekran (razdvojene po jednim razmakom), bez korištenja dodatnog parametra koji predstavlja broj elemenata vektora (obratimo pažnju da se prilikom deklaracije vektora kao formalnog parametra nikakve dimenzije niti zgrade u kojima se inače navode dimenzije ne navode):

```
void IspisiVektor(vector<int> v) {
    for(int i = 0; i < v.size(); i++) cout << v[i] << " ";
}
```

Prilikom poziva ove funkcije, stvarni parametar će biti u funkciju prenesen *po vrijednosti*. To zapravo znači da će, prilikom poziva funkcije, čitav vektor koji predstavlja stvarni parametar biti *kopiran* u formalni parametar “*v*”, nakon čega eventualna promjena sadržaja elemenata vektora “*v*” neće imati nikakav utjecaj na stvarni parametar (kao što je i inače situacija kod prenosa po vrijednosti). Međutim, kopiranje čitavog vektora pri svakom pozivanju funkcije može biti vremenski vrlo zahtjevna operacija. Stoga je bolje vektore prenosići *po referenci*. Naravno, na taj način se funkciji ostavlja mogućnost da *promjeni* sadržaj svog stvarnog parametra, ali ova funkcija to svakako neće uraditi. Ipak, ukoliko koristimo prenos po referenci, kao stvarni parametar u pozivu funkcije nećemo moći upotrijebiti proizvoljan izraz tipa vektor (npr. poziv neke druge funkcije koja vraća vektor kao rezultat), već samo l-vrijednost. Ovaj problem se može riješiti da formalni parametar deklariramo kao *referencu na konstantu* (koja se može vezati za proizvoljan izraz):

```

void IspisiVektor(const vector<int> &v) {
    for(int i = 0; i < v.size(); i++) cout << v[i] << " ";
}

```

Na ovaj način također omogućavamo kompjajleru prijavu greške ukoliko funkcija pokuša nehotično da izvrši promjenu sadržaja svog parametra.

Generalno, radi izbjegavanja neefikasnosti koja može nastati uslijed kopiranja masivnih parametara (a vektori su upravo takvi), parametre tipa vektor u funkcije uvijek treba prenositi po referenci (ukoliko funkcija treba da mijenja vrijednost stvarnog parametra), ili po referenci na konstantu (ukoliko funkcija ne treba da mijenja vrijednost stvarnog parametra). Prenos po vrijednosti treba koristiti samo ukoliko funkcija mijenja vrijednost svog formalnog parametra, a ta promjena *ne treba da se odrazi na vrijednost stvarnog parametra*. U tom slučaju ne možemo koristiti niti prenos po referenci (jer bi u tom slučaju stvarni parametar bio promijenjen), niti po referenci na konstantu (jer u tom slučaju ne bi bila dozvoljena promjena vrijednosti formalnog parametra).

Kao ilustraciju prenosa vektora kao parametara, slijedi modifikacija programa koji računa harmonijsku sredinu niza brojeva, u kojem se umjesto nizova koriste vektori:

```

#include <iostream>
#include <vector>

double HarmonijskaSredina(const vector<double> &a) {
    double suma(0);
    for(int i = 0; i < a.size(); i++) suma += 1 / a[i];
    return a.size() / suma;
}

int main() {
    int n;
    cout << "Koliko elemenata ima niz? ";
    cin >> n;
    vector<double> niz(n);
    cout << "Unesite elemente niza:\n";
    for(int i = 0; i < n; i++) cin >> niz[i];
    cout << "Harmonijska sredina iznosi "
        << HarmonijskaSredina(niz) << endl;
    return 0;
}

```

Posljednji primjer predstavlja prepravljenu verziju programa u kojem je definirana funkcija za računanje vektorskog proizvoda. Ovaj primjer ilustrira i kako se vektori mogu vratiti kao rezultat iz funkcije:

```

#include <iostream>
#include <vector>

using namespace std;

vector<double> VektorskiProizvod(const vector<double> &a,
                                         const vector<double> &b) {
    vector<double> c(3);
    c[0] = a[1] * b[2] - a[2] * b[1];
    c[1] = a[2] * b[0] - a[0] * b[2];
    c[2] = a[0] * b[1] - a[1] * b[0];
    return c;
}

int main() {

```

```
vector<double> a(3), b(3);
cout << "Unesi prvi vektor: ";
cin >> a[0] >> a[1] >> a[2];
cout << "Unesi drugi vektor: ";
cin >> b[0] >> b[1] >> b[2];
vector<double> c = VektorskiProizvod(a, b);
cout << "Vektorski proizvod je: "
     << c[0] << " " << c[1] << " " << c[2] << endl;
return 0;
}
```

Ni rješenje upotrijebljeno u ovom primjeru nije savršeno. Funkcija “VektorskiProizvod” se sada zaista poziva na matematički konzistentniji način, međutim njoj se kao parametri mogu proslijediti vektori proizvoljne dimenzije, a ne samo vektori dimenzije 3, kakve bi funkcija trebala da očekuje. Potpuno korektno rješenje ovog problema moguće je postići samo kreiranjem korisničkih tipova podataka, kod kojih je moguće u potpunosti odrediti njihovo ponašanje. Ovom temom ćemo se kasnije detaljno baviti.

19. Umetnute i generičke funkcije

Umetnute funkcije (engl. *inline functions*) predstavljaju interesantno svojstvo jezika C++ koje često omogućavaju poboljšanje modularnosti programa bez gubitaka na efikasnosti, odnosno poboljšanje efikasnosti programa koji intenzivno koriste i pozivaju kratke funkcije. Umetnute funkcije su gotovo u potpunosti istisnuli upotrebu tzv. *makroa*, koji su se intenzivno koristili u jeziku C, a koje ovdje nećemo obrađivati, s obzirom na brojne neželjene posljedice koje mogu nastati prilikom njihove nepažljive upotrebe.

Da bismo vidjeli potrebu za umetnutim funkcijama, moramo razmotriti šta se tačno dešava u programima koji koriste obične (neumetnute) funkcije. Prilikom nailaska na definiciju funkcije, kompjajler generira izvršni mašinski kôd koji odgovara naredbama koje čine tijelo funkcije. Generirani kôd za svaku od funkcija nalazi se u prevedenom programu *na samo jednom mjestu*, bez obzira koliko puta se ta funkcija u programu poziva. Prilikom svakog poziva funkcije, kompjajler generira kôd koji kopira stvarne argumente funkcije u formalne (pri prenosu po referenci kopiraju se samo adrese), pamti negdje u memoriji mjesto odakle je poziv izvršen, a zatim vrši skok na početak kôda koji odgovara funkciji. Po završetku funkcije, rezultat funkcije se kopira u traženo odredište (za slučaj funkcija koje vraćaju vrijednost), nakon čega se na osnovu informacije o mjestu sa kojeg je izvršen poziv funkcije vrši skok nazad na instrukcije koje slijede neposredno iza mjesta poziva. Očigledno se prilikom svakog poziva funkcije gubi izvjesno vrijeme na razne pomoćne radnje, poput kopiranja parametara i rezultata, pamćenja informacija o mjestu na koje treba izvršiti povratak, i izvršavanja samih skokova sa jednog mesta u programu na drugo (koji također troše vrijeme).

Za razliku od običnih funkcija, prilikom nailaska na definiciju umetnute funkcije, kompjajler *ne generira ništa*. Tek kada se funkcija pozove, kompjajler na mjestu poziva funkcije *generira i umeće* kompletan mašinski kôd koji odgovara naredbama unutar tijela funkcije. Dakle, ovdje se nikakvi skokovi ne vrše: tijelo funkcije se praktično umeće na mjesto poziva. Ovo umetanje se vrši na svakom mjestu gdje se funkcija poziva, što znači da se generirani kôd multiplicira pri svakom pozivu funkcije, što u slučaju da je tijelo funkcije veliko, može osjetno produžiti veličinu generiranog izvršnog kôda (pogotovo ukoliko se funkcija poziva sa mnogo mjesta). Što se tiče kopiranja parametara i rezultata funkcije, ono je ponekad neizbjježno čak i u slučaju umetnutih funkcija, mada u mnogim slučajevima kompjajler može pomoći raznih optimizacijskih trikova u potpunosti izbjegći potrebu za kopiranjem parametara (neznatnom izmjenom generiranog kôda u ovisnosti od vrijednosti stvarnih parametara pri različitim pozivima).

Umetnute funkcije se definiraju na isti način kao i obične funkcije, samo što se na početku njihovog zaglavlja dodaje ključna riječ “**inline**” (kada koristimo prototipove, ovu ključnu riječ treba staviti i u prototip). Sa aspekta izvršavanja programa, rezultat izvršavanja će uvijek biti isti, bez obzira da li je funkcija umetnuta ili ne. Međutim, u slučaju kratkih funkcija, koje se pozivaju često, efikasnost programa može se osjetno poboljšati ukoliko funkcije definiramo kao umetnute, jer se ne gubi vrijeme na propratne radnje koje su uvijek vezane za poziv običnih funkcija. Izvršni program koji koristi umetnute funkcije obično je duži nego u slučaju da smo koristili obične funkcije. Međutim, u slučaju da su funkcije koje definiramo kao umetnute vrlo kratke, proglašavanje funkcije za umetnuto funkcionu može čak skratiti izvršni program. Naime, nekada je isplatnije umetnuti čitav kôd koji odgovara tijelu funkcije na mjesto poziva nego generirati kôd za poziv funkcije, pogotovo ukoliko se umetanje može izvesti tako da kompjajler izbjegne potrebu za kopiranjem parametara. Stoga su najbolji kandidati za proglašavanje ua umetnute funkcije upravo vrlo kratke funkcije, čije se tijelo sastoji npr. samo od naredbe “**return**” iza koje slijedi neki izraz, ili od niza posve jednostavnih naredbi, poput naredbi pridruživanja. Slijede neki primjeri funkcija, koje bi uvijek trebalo definirati kao umetnute funkcije:

```

inline void Razmjeni(double &x, double &y) {
    double pomocna = x;
    x = y; y = pomocna;
}
inline double Kub(double x) {
    return x * x * x;
}

inline bool DaLiJePrestupna(int godina) {
    return (godina % 4 == 0) && (godina % 100 != 0 || godina % 400 == 0);
}

inline double Povrsina(double r) {
    const double PI(3.1415924);
    return PI * r * r;
}

inline int VeciOd(int x, int y) {
    if(x > y) return x;
    else return y;
}

```

Objasnimo na koji način umetnute funkcije mogu poboljšati modularnost programa bez gubitka na efikasnosti. Naime, mnogi programeri se ustručavaju da pišu kratke funkcije za obavljanje često obavljanih radnji, tvrdeći da se njihovim čestim pozivanjem gubi na efikasnosti. Na primer, prilikom problema sortiranja nizova (kojim ćemo se kasnije baviti) intenzivno se javlja potreba za razmjenom elemenata niza, koja se izvršava unutar petlje veliki broj puta (broj razmjena u slučaju velikih nizova može biti i više desetina pa i stotina hiljada). Stoga će mnogi programeri radije na mjesto gdje treba razmijeniti dva elementa niza direktno napisati sekvencu naredbi za razmjenu dva elementa nego pozvati funkciju poput funkcije “Razmjeni” koja vrši razmjenu svoja dva parametra, da ne bi došlo do gubitka u efikasnosti koji nastaje prilikom mnogobrojnih poziva funkcije. Do gubitka efikasnosti zaista dolazi ukoliko se funkcija “Razmjeni” realizira kao *obična* funkcija. Međutim, ukoliko se ova funkcija realizira kao *umetnuta* funkcija, efikasnost je potpuno ista kao da je na mjestu poziva funkcije prosto umetnuta sekvenca naredbi koja vrši razmjenu. Dakle, efikasnost nije smanjena, a program postaje razumljiviji, jer je operacija razmjene izdvojena u posebnu cjelinu!

Treba voditi računa da ključna riječ “**inline**” samo predstavlja *preporuku* kompjajleru da funkciju realizira kao umetnutu, ali ne i *naređenje*. Drugim riječima, kompjajler ima puno pravo da preporuku za definiranjem umetnute funkcije ignorira, *bez ikakvog upozorenja* (funkcija se tada realizira kao obična funkcija). Tako, na primjer, kompjajler može odbiti zahtjev za realizaciju umetnute funkcije ukoliko zaključi da je ona *prevelika* (mada je pojam “prevelik” posve relativan). Jedan od slučajeva u kojem će kompjajler gotovo sigurno ignorirati ključnu riječ “**inline**” je slučaj kada funkcija sadrži bilo koju vrstu *petlji*. Naime, vrijeme trajanja izvršavanja funkcija koje sadrže petlje je obično samo po sebi dovoljno dugo da je ušteda u vremenu koja bi se dobila izbjegavanjem pomoćnih radnji koje prate poziv funkcije posve zanemarljiva.

Za razliku od umetnutih funkcija, koje predstavljaju samo koristan “začin” za poboljšanje modularnosti i efikasnosti programa, *generičke funkcije* predstavljaju jedno od najkorisnijih svojstava jezika C++, koje su znatno doprinijeli popularnosti ovog jezika. Generičke funkcije su relativno skoro uvedene u jezik C++, i njihove mogućnosti se neprestano proširuju iz verzije u verziju jezika C++. Najjednostavnije rečeno, generičke funkcije, na izvjestan način, predstavljaju funkcije kod kojih stvarni tip parametara, povratne vrijednosti ili neke od lokalnih promjenljivih *nije poznat sve do trenutka poziva funkcije*, i može se *razlikovati od poziva do poziva*. Naime, ranije smo vidjeli da je pozivanje klasičnih funkcija kod kojih se tip formalnih i stvarnih argumenata ne slaže ili potpuno nemoguće (recimo u slučaju

prenosa parametara po referenci ili parametara nizovnih tipova), ili je praćeno automatskim konverzijama tipa koje ne samo da mogu znatno ugroziti efikasnost, nego mogu izazvati i neželjene efekte. Na primjer, umetnuta funkcija "Razmjeni" koju smo ranije napisali može se primijeniti samo na dvije promjenljive tipa "double" (ništa se ne bi promijenilo i da funkcija nije umetnuta). Funkcija "VeciOd", koju smo također imali kao primjer u ovom poglavlju, može se načelno primijeniti i na stvarne parametre koji nisu tipa "int", ali pri tome dolazi do konverzije u tip "int". Stoga, upotrijebimo li ovu funkciju sa stvarnim parametrima tipa "double", doći će do posve neželjenog odsjecanja decimala! Da smo u deklaraciji funkcije umjesto tipa "int" upotrijebili tip "double", funkcija bi se mogla pozvati sa stvarnim parametrima tipa "int", ali bi tom prilikom dolazilo do neefikasnih konverzija iz tipa "int" u tip "double". Dalje, sljedeća funkcija

```
void IspisiNiz(double niz[], int broj_elemenata) {
    for(int i = 0; i < broj_elemenata; i++) cout << niz[i] << " ";
}
```

koja je zamišljena za ispis elemenata nekog niza, radi samo sa nizovima čiji su elementi tipa "double". Ranije smo vidjeli da se problemi ove vrste mogu riješiti preklapanjem funkcija po tipu za svaki od tipova stvarnih parametara za koje želimo koristiti funkciju. Međutim, na taj način moramo stalno prepisivati praktično isto tijelo funkcije u svakoj verziji funkcije koju trebamo napraviti. Ovaj problem je posebno izražen u slučajevima kada ne možemo unaprijed predvidjeti koji će se tipovi stvarnih parametara koristiti prilikom poziva funkcije (npr. ukoliko želimo da razvijemo svoju biblioteku funkcija koju će koristiti drugi programeri, nemoguće je predvidjeti sa kakvim tipovima parametara će oni pozivati funkcije).

Generičke funkcije su uvedene sa ciljem da se riješe upravo opisani problemi. Generičke funkcije se u jeziku C++ realiziraju uz pomoć tzv. *predložaka* ili *šablonu* (engl. *templates*), koji se koriste još i za realizaciju *generičkih klasa*, o kojima ćemo govoriti kasnije. Pogledajmo, na primjer, kako bi izgledala definicija generičke verzije funkcije "VeciOd" (ključnu riječ "inline" koja je označavala da se radi o umetnutoj funkciji smo radi jednostavnosti izbacili, mada smo je mogli i zadržati, bez utjecaja na smisao primjera koji razmatramo):

```
template <typename NekiTip>
NekiTip VeciOd(NekiTip x, NekiTip y) {
    if(x > y) return x;
    else return y;
}
```

Sama definicija funkcije, bez prvog reda, podsjeća na klasičnu definiciju, samo što se u njoj umjesto imena tipa "int" javlja ime "NekiTip". Ovo ime je *proizvoljan identifikator*, koji je upotrijebljen da označi neki tip, čija tačna priroda u trenutku definiranja funkcije nije poznata. Naravno, da smo napisali ovaku definiciju bez prvog reda koji najavljuje da se radi o šablonu, kompjuter bi prijavio grešku (osim ukoliko smo pomoću "typedef" naredbe dali neki smisao identifikatoru "NekiTip"). Ovako, ključna riječ "**template**" govori da se radi o šablonu u koji je uklapljena funkcija "VeciOd". Iza ključne riječi "**template**" unutar šiljastih zagrada ("<>") navode se *parametri šablonu* (engl. *template parameters*). Parametri šablonu mogu biti različitih vrsta, ali u ovom trenutku nas zanimaju samo parametri šablonu koji predstavljaju ime nekog tipa koji nije poznat u trenutku definiranja funkcije (ovakvi parametri nazivaju se i *metatipovi*). Takvi parametri šablonu deklariraju se pomoću ključne riječi "**typename**" iza koje slijedi ime parametra šablonu (u ovom primjeru "NekiTip"). Umjesto ključne riječi "**typename**" može se koristiti i ključna riječ "**class**", mada se takva praksa ne preporučuje, s obzirom da se ključna riječ "**class**" također koristi i za svrhu koja nema nikakve veze sa šablonima, o čemu ćemo govoriti kasnije. Takva praksa, koja se može susresti i u mnogim knjigama, nasljeđe je prošlosti, s obzirom da je ključna riječ "**typename**" uvedena u standard jezika C++ tek nedavno.

Prilikom nailaska na šablon, kompjajler ne generira nikakav izvršni kôd, s obzirom da ne zna koji zaista tip predstavlja parametar šablona. Tek prilikom nailaska na poziv generičke funkcije definirane šablonom, kompjajler na osnovu zadanih stvarnih parametara pokušava da zaključi koji zaista tip odgovara parametru šablona, i po potrebi izvrši generiranje odgovarajuće verzije funkcije. Ovaj postupak zaključivanja naziva se *dedukcija tipa* (engl. *type deduction*). Na primjer, pretpostavimo da imamo sljedeću sekvencu naredbi:

```
int a(3), b(5);
cout << VeciOd(a, b);
```

Prilikom nailaska na poziv funkcije “VeciOd” sa stvarnim parametrima koji su tipa “**int**”, kompjajler će pokušati da utvrdi može li se ovakav poziv *uklopiti u navedeni šablon*, davanjem konkretnog značenja parametru šablona “NekiTip”. Kompajler će lako zaključiti da se uklapanje može izvesti, ukoliko se pretpostavi da parametar šablona “NekiTip” predstavlja tip “**int**”. Stoga će kompjajler generirati verziju funkcije “VeciOd”, uzimajući da je “NekiTip” sinonim za tip “**int**”, odnosno kao da je izvršena deklaracija poput

```
typedef int NekiTip;
```

Ovakvo naknadno generiranje funkcije, do kojeg dolazi tek kada kompjajler na osnovu tipa stvarnih parametara zaključi *kakvu verziju funkcije* “VeciOd” treba generirati, naziva se *instantacija generičke funkcije*. Kažemo da je pri nailasku na navedeni poziv stvoren jedan konkretni *primjerak (instanca)* generičke funkcije “VeciOd”. Ukoliko se kasnije nađe ponovo na poziv funkcije “VeciOd” pri čemu su parametri ponovo tipa “**int**”, neće doći ni do kakve nove instantacije, nego će prosto biti *pozvana* već generirana verzija funkcije “VeciOd” koja prihvata parametre tipa “**int**”. Međutim, ukoliko se kasnije nađe na poziv funkcije “VeciOd” sa *drugačijim tipom* stvarnih parametara (npr. tipom “**double**”), biće instancirana *nova verzija* funkcije “VeciOd”, koja prihvata parametre tipa “**double**” (već instancirana verzija koja prima parametre tipa “**int**” i dalje će postojati, i biće prosto pozvana u slučaju da se ponovo nađe na poziv funkcije “VeciOd” sa cijelobrojnim parametrima), s obzirom da će se u postupku dedukcije tipa zaključiti da se poziv može uklopiti u šablon samo ukoliko se parametru šablona “NekiTip” da smisao tipa “**double**”. Tako, svaki nailazak na poziv funkcije “VeciOd” sa tipom stvarnih parametara kakav nije korišten u ranijim pozivima, dovodi do generiranja nove verzije funkcije.

Formalno posmatrano, napisanu generičku funkciju “VeciOd” možemo shvatiti kao funkciju koja prihvata parametre *bilo kojeg tipa* (pod uvjetom da oba parametra imaju *isti tip*, što ćemo uskoro vidjeti). Mada je ovakvo posmatranje korisno sa aspekta razumijevanja smisla generičkih funkcija, kreiranje funkcija koje bi zaista primale parametre proizvoljnih tipova je *tehnički neizvodljivo*. Stoga, mehanizam šablona na kojem se zasniva rad generičkih funkcija zapravo predstavlja *automatizirano preklapanje po tipu*. Ukoliko u programu na pet mesta pozovemo funkciju “VeciOd” sa pet različitih tipova stvarnih argumenata, u generiranom izvršnom kôdu nalaziće se *pet verzija* funkcije “VeciOd” (svaka za po jedan tip parametara), kao da smo ručno napisali pet preklopljenih verzija funkcije “VeciOd” za različite tipove parametara, a ne jedna hipotetička funkcija “VeciOd” koja može primati različite tipove parametara. Ipak, ovo preklapanje je potpuno automatizirano i skriveno od programera koji ne mora o njemu da se brine.

Prilikom pozivanja generičke funkcije, moguće je zaobići mehanizam dedukcije tipova i *eksplicitno specificirati* šta predstavljaju parametri šablona. Na primjer, u sljedećoj sekvenci naredbi

```
int a(3), b(5);
cout << veci<b>(a, b);
```

biće instancirana (ukoliko nije bila prethodno instancirana) i pozvana verzija funkcije “VeciOd” u kojoj

parametar šablona “NekiTip” ima značenja tipa “**double**”, bez obzira što bi postupak dedukcije tipa doveo do zaključka da “NekiTip” treba imati značenje tipa “**int**”. Napomenimo da se razmak između imena funkcije “VeciOd” i znaka “<” ne smije pisati, jer bi znak “<” mogao biti pogrešno interpretiran kao operator poređenja (najbolje je posmatrati čitavu frazu “veci<**double**>” kao jednu riječ, odnosno kao *ime specijalne verzije* generičke funkcije “VeciOd” kod koje parametar šablona “NekiTip” predstavlja tip “**double**”.

Ako pažljivo razmotrimo kako je parametar šablona “NekiTip” upotrijebljen u deklaraciji generičke funkcije “VeciOd”, lako možemo zaključiti da je pretpostavljeno da će ova parametra funkcije biti *istog tipa* (s obzirom da je za imenovanje njihovog tipa upotrijebljen *isti identifikator*), i da će istog tipa biti i *povratna vrijednost funkcije*. Ukoliko ove pretpostavke ne ispunimo prilikom poziva funkcije, dedukcija tipa će biti onemogućena, i kompjajler će prijaviti grešku, kao u slučaju sljedeće sekvene naredbi:

```
double realni(5.27);
int cijeli(3);
cout << VeciOd(realni, cijeli);
```

Naime, ovdje kompjajler ne može dedukcijom zaključiti da li parametar šablona “NekiTip” treba predstavljati tip “**double**” ili “**int**”. Isti problem nastaje i u naizgled mnogo bezazlenijem pozivu

```
cout << VeciOd(5.27, 3);
```

s obzirom da “5.27” i “3” nisu istog tipa. Naravno, problem bismo mogli riješiti eksplicitnom specifikacijom parametara šablona, odnosno upotreboru nekog od sljedećih poziva (prvi poziv je svrshodniji, jer će u drugom pozivu doći do odsjecanja decimala):

```
cout << VeciOd<double>(realni, cijeli);
cout << VeciOd<int>(realni, cijeli);
```

Jasno, u slučaju kada su parametri funkcije *brojevi*, jednostavnije rješenje bi bilo napisati

```
cout << VeciOd(5.27, 3.);
```

s obzirom da parametri sada jesu *istog tipa*, pa je automatska dedukcija tipa moguća.

Činjenica da se izvršni kôd za generičku funkciju ne može generirati dok se pri pozivu funkcije ne odredi stvarno značenje parametara šablona ima neobične posljedice na strategiju prijavljivanja grešaka pri prevodenju od strane kompjajlera. Razmotrimo sljedeći primjer (koji pretpostavlja da smo uključili biblioteku “complex” u program):

```
complex<double> c1(3,5), c2(2,8);
cout << VeciOd(c1, c2);
```

Na prvi pogled, sve protiče u redu: u postupku dedukcije tipa zaključuje se da parametar šablona “NekiTip” treba imati značenje tipa “complex<**double**>”, nakon čega može započeti instantacija verzije funkcije “VeciOd” za ovaj tip. Međutim, ovdje nastaje problem: funkcija se oslanja na operator poređenja “<” koji nije definiran za kompleksne brojeve, odnosno za tip “complex<**double**>”. Jasno je da kompjajler treba da prijavi grešku. Sada se nameće pitanje *gdje greška treba da bude prijavljena*: da li na *mjestu poziva funkcije*, ili na *mjestu gdje je upotrijebljen operator <?* Oba rješenja imaju svoje nedostatke. Ukoliko se greška pojavi na mjestu poziva, programer će pomisliti da nešto nije u redu sa načinom kako poziva funkciju. To doduše jeste tačno (poziva je sa nedozvoljenom vrstom argumenata), ali ukoliko je definicija generičke funkcije dugačka i složena, on neće dobiti nikakvu informaciju o tome

zašto poziv nije dobar, odnosno zašto poziv nije dozvoljen sa takvim argumentima. Ne zaboravimo da nije moguće samo reći da se tipovi ne slažu, jer je funkcija načelno dizajnirana da prima argumente *bilo kojeg tipa*. Očigledno nije problem u prenosu parametara, nego funkcija pokušava sa argumentima uraditi nešto *što se sa njima ne smije uraditi*, a što je to, ne možemo saznati ukoliko se greška prijavi na mjestu poziva. S druge strane, ukoliko se greška prijavi na mjestu nedozvoljene operacije (u našem slučaju operacije poređenja), programeru neće biti jasno što nije u redu sa tom operacijom, ukoliko nije svjestan da je funkcija pozvana sa takvima argumentima za koje ta operacija zaista nije definirana.

Opisani problem nije lako riješiti, stoga većina kompjajlera primjenjuje solomensko rješenje: greška će biti prijavljena i na mjestu poziva funkcije, i na mjestu ne dozvoljene operacije. Greške koje formira kompjajler u tom slučaju najčešće imaju formu koja načelno glasi ovako: “prilikom instantacije funkcije na mjestu *mjesto1* došlo je do *tog i tog* problema unutar funkcije na mjestu *mjesto2*”. Pri tome, *mjesto1* predstavlja mjesto odakle je funkcija pozvana, odnosno na kojem je pokušana (neuspješna) instantacija funkcije, dok *mjesto2* predstavlja mjesto unutar same definicije funkcije na kojem je uočen problem. Na taj način, kombinirajući ponudene informacije, programer može saznati zbog čega je zaista došlo do problema.

Prethodni primjer ilustrira da nakon uvođenja generičkih funkcija, sam pojam tipa nije dovoljan za potpunu specifikaciju ponašanja funkcije. Naime, napisana funkcija “*VeciOd*” načelno je napisana tako da prihvata argumente proizvoljnog tipa. Međutim, prethodni primjer demonstrira da tipovi argumenta ipak ne mogu biti posve proizvoljni: argumenti moraju biti *objekti koji se mogu upoređivati po veličini*. Ova činjenica nameće izvjesna ograničenja na tipove argumenta koji se mogu proslijediti funkciji “*VeciOd*”. Skup ograničenja koje mora ispunjavati neki objekat da bi se mogao proslijediti kao parametar nekoj generičkoj funkciji naziva se *koncept*. Tako kažemo da funkcija “*VeciOd*” prihvata samo argumente koji zadovoljavaju *koncept uporedivih objekata*. Bilo koji konkretni tip koji zadovoljava ograničenja koja postavlja neki koncept naziva se *model* tog koncepta. Tako su, na primjer, tipovi “**int**” i “**double**” modeli koncepta uporedivih objekata, dok tip “**complex<double>**” nije model ovog koncepta.

Pojmovi *koncept* i *model* su čisto filozofski pojmovi, i nisu dio jezika C++, u smislu da u jeziku C++ nije moguće deklarirati neki koncept, niti deklarirati da je neki tip model nekog koncepta. Na primjer, nemoguće je deklaracijom specifizirati da parametri funkcije “*VeciOd*” moraju biti uporedivi objekti, tako da kompjajler odmah pri pozivu funkcije “*VeciOd*” sa kompleksnim argumentima prijavi neku grešku tipa “argumenti nisu uporedivi objekti”. Najviše što možemo sami uraditi u cilju doprinosa poštovanja filozofije koncepcata i modela je da parametrima šablona damo takva imena koja će nas podsjećati na ograničenja koja moraju biti ispunjena prilikom njihovog korištenja. Tako je, na primjer, veoma mudro generičku funkciju “*vuciOd*” definirati pomoću šablona koji izgleda na primjer ovako:

```
template <typename UporediviObjekat>
UporediviObjekat veci(UporediviObjekat x, UporediviObjekat y) {
    if(x > y) return x;
    else return y;
}
```

Tako će svako ko pogleda zaglavje funkcije odmah znati da funkcija nameće ograničenje da njeni stvarni parametri moraju biti uporedivi objekti. Zanemarimo činjenicu da se u ovom primjeru to lako može vidjeti i na osnovu samog imena funkcije i njene definicije. Naime, to ne mora biti tako u slučaju složenijih funkcija!

Moguće je definirati šablove sa više parametara šablona. Pogledajmo sljedeću definiciju generičke funkcije “*VuciOd*” koja je definirana pomoću šablona sa *dva parametra*:

```
template <typename Tip1, typename Tip2>
Tip1 VuciOd(Tip1 x, Tip2 y) {
    if(x > y) return x;
```

```

    else return y;
}

```

U ovom slučaju, poziv poput

```
cout << VeciOd(5.27, 3);
```

postaje savršeno legalan, jer se dedukcijom tipova zaključuje da je uklapanje u šablon moguće uz pretpostavku da parametar šablona "Tip1" predstavlja tip "**double**", a parametar šablona "Tip2" tip "**int**". Naravno, i dalje bi se funkcija "VeciOd" mogla pozvati sa dva parametra istog tipa (dedukcija tipova bi dovela do zaključka da i "Tip1" i "Tip2" predstavljaju isti tip). Ipak, ovo rješenje posjeduje jedan ozbiljan nedostatak. Naime, u njemu je pretpostavljeno da je tip rezultata ujedno i tip prvog parametra. Stoga, ukoliko bismo napravili poziv

```
cout << VeciOd(3, 5.27);
```

došlo bi do odsjecanja decimala prilikom vraćanja rezultata iz funkcije, s obzirom da bi za tip povratne vrijednosti bio izabran isti tip kao i tip prvog parametra, a to je tip "**int**". Univerzalno rješenje za ovaj problem ne postoji: nemoguće je dati opće pravilo kakav bi trebao biti tip rezultata ako su operandi koji se porede različitog tipa. U ovom slučaju pomaže jedino eksplisitna specifikacija parametara šablona:

```
cout << VeciOd<double,double>(3, 5.27);
```

Podržano je i navođenje *nepotpune eksplisitne specifikacije* parametara šablona, kao u sljedećem pozivu:

```
cout << VeciOd<double>(3, 5.27);
```

U slučaju nepotpune specifikacije, za značenje parametra šablona se *slijeva nadesno* uzimaju specifizirana značenja, a preostali parametri koji nisu specifizirani pokušavaju se odrediti dedukcijom. Tako, u prethodnom primjeru, parametar šablona "Tip1" uzima specifizirano značenje "**double**", a smisao parametra šablona "Tip2" određuje se dedukcijom (i također će dobiti značenje "**double**").

Parametri šablona deklarirani sa ključnom riječju "**typename**" (metatipovi) mogu se upotrijebiti ne samo u deklaraciji formalnih parametara generičke funkcije, nego i bilo gdje gdje se može upotrijebiti ime nekog tipa. Međutim, treba voditi računa da se dedukcijom može saznati samo smisao onih metatipova koji su iskorišteni za deklaraciju formalnih parametara generičke funkcije. Na primjer, definicija koja načelno izgleda poput sljedeće

```

template <typename NekiTip>
int F(int n) {
    NekiTip x;
    ...
}
```

i definira generičku funkciju koja prima *cjelobrojni parametar*, vraća *cjelobrojni rezultat*, ali pri tome u svom tijelu koristi lokalnu promjenljivu "x" čiji tip *nije poznat na mjestu definicije funkcije*. Jasno je da je tip ove funkcije nemoguće zaključiti iz poziva funkcije, tako da je funkciju "F" moguće pozvati samo uz eksplisitnu specifikaciju značenja parametra šablona "NekiTip". Također, parametar šablona koji je iskorišten samo kao tip povratne vrijednosti ne može se zaključiti dedukcijom. Tako se, u sljedećoj verziji generičke funkcije "VeciOd"

```

template <typename TipRezultata, typename Tip1, typename Tip2>
TipRezultata VeciOd(Tip1 x, Tip2 y) {
    if(x > y) return x;
    else return y;
}

```

```
}
```

parametar šablona “TipRezultata” ne može odrediti dedukcijom, već se mora eksplisitno specificirati, kao u sljedećoj sekvenci naredbi:

```
double realni(5.27);
int cijeli(3);
cout << VeciOd<double>(realni, cijeli);
```

Ovdje se naravno radi o nepotpunoj specifikaciji: značenje parametara “Tip1” i “Tip2” određeno je dedukcijom. Naravno, mogli smo navesti i potpunu specifikaciju:

```
cout << VeciOd<double, double, int>(realni, cijeli);
```

Važno je istaći da je redoslijed kojim su navedeni parametri šablona *bitan*. Naime, da smo šablon definirali ovako:

```
template <typename Tip1, typename Tip2, typename TipRezultata>
TipRezultata VeciOd(Tip1 x, Tip2 y) {
    if(x > y) return x;
    else return y;
}
```

tada poziv

```
cout << VeciOd<double>(realni, cijeli);
```

ne bi bio ispravan: smatralo bi se da je značenje parametra “Tip1” specificirano na “**double**”, dok se parametri “Tip2” i “TipRezultata” trebaju odrediti dedukcijom, što nije moguće.

Mehanizam šablona, kao što smo već rekli, predstavlja izvjesnu automatizaciju postupka preklapanja po tipu, pri čemu se preklapanje vrši automatski uz pretpostavku da je za sve neophodne tipove postupak opisan u tijelu funkcije načelno isti. Međutim, pretpostavimo situaciju da želimo napisati funkciju koja se za sve moguće tipove argumenata izvodi na jedan način, ali se za neki specijalan tip (npr. tip “**int**”) izvodi na *drugačiji način* (u kasnijim poglavljima ćemo vidjeti i konkretne primjere ovakvih izuzetaka). Tada možemo posebno definirati *generičku funkciju* i istoimenu *običnu funkciju* koja prihvata željeni specijalni tip (tada za tu funkciju kažemo da predstavlja *specijalizaciju* generičke funkcije za konkretni tip). Na primjer:

```
template <typename NekiTip>
NekiTip F(NekiTip x) {
    ...
}
int F(int x) {
    ...
}
```

Prilikom poziva funkcije “F”, kompjuter će prvo potražiti postoji li *obična* funkcija “F” za koju se tip stvarnog argumenta slaže sa tipom formalnog argumenta. Ukoliko postoji, ona će biti pozvana. Tek ukoliko to nije slučaj, koristi se opisani mehanizam generičkih funkcija.

Generičke funkcije se također mogu preklapati po broju argumenata, pa čak i po tipu argumenata (uz dosta opreza), pod uvjetom da je iz poziva nedvosmisleno jasno koju verziju funkcije treba pozvati. Tako je, pored dosad napisane generičke funkcije “VeciOd” (odlučimo se za prvu napisanu varijantu u ovom

poglavlju), koja prima dva parametra, moguće dodati i verziju funkcije “`VeciOd`” koja prihvata *tri argumenta* istog tipa (pod uvjetom da zadovoljavaju koncept uporedivih objekata), i vraća kao rezultat *najveći od njih*:

```
template <typename NekiTip>
NekiTip VeciOd(NekiTip x, NekiTip y, NekiTip z) {
    if(x > y && x > z) return x;
    else if(y > x && y > z) return y;
    else return z;
}
```

Na osnovu izloženih razmatranja, sasvim je jasno kako možemo napraviti generičku funkciju koja razmjenjuje svoja dva argumenta *proizvoljnog tipa* (razumije se da im tipovi *moraju biti jednaki*). Ovakvu funkciju je poželjno realizirati kao *umetnutu*, tako da dobijamo sljedeću definiciju:

```
template <typename NekiTip>
inline void Razmijeni(NekiTip &x, NekiTip &y) {
    NekiTip pomocna = x;
    x = y; y = pomocna;
}
```

Ne moraju svi formalni parametri generičke funkcije biti neodređenog tipa. Na primjer, sljedeća definicija prikazuje generičku funkciju “`IspisiNiz`” koja ispisuje na ekran elemente niza čiji su elementi ma kakvog tipa, pod uvjetom da se oni *mogu ispisivati* (vidjećemo kasnije da postoje objekti koji se ne mogu ispisivati na ekran), u kojoj je drugi parametar (broj elemenata niza) običan cijeli broj. Možemo reći da funkcija posjeduje ograničenje da prvi parametar mora biti niz elemenata koji zadovoljavaju *koncept ispisivih elemenata*:

```
template <typename IspisiviObjekat>
void IspisiNiz(IspisiviObjekat niz[], int broj_elemlenata) {
    for(int i = 0; i < broj_elemlenata; i++) cout << niz[i] << " ";
}
```

Slijedi primjer upotrebe ove funkcije:

```
int a[10] = {3, 4, 0, 8, 1, -6, 1, 4, 2, -7};
IspisiNiz(a, 10);
double b[5] = {2.13, -3.4, 8, 1.232, 7.6};
IspisiNiz(b, 5);
```

U prvom pozivu, dedukcija tipa zaključuje da “`IspisiviObjekat`” predstavlja tip “`int`”, dok u drugom pozivu predstavlja tip “`double`”.

Treba obratiti pažnju da u ovoj funkciji imamo takozvanu *djelimičnu (parcijalnu) dedukciju tipa*. Naime, za formalni parametar “*niz*” se na osnovu njegove deklaracije (preciznije, na osnovu prisustva uglastih zagrada) zna da on nije u potpunosti *bilo kakav tip*: on mora biti *niz elemenata*, a tip elemenata će biti određen dedukcijom. To automatski znači da će poziv poput

```
IspisiNiz(5, 3);
```

biti odbijen i prije nego što se pokuša instantacija funkcije, jer se stvarni parametar “5” *ne može interpretirati kao niz*. Drugim riječima, ovakav stvarni parametar je *nemoguće uklopiti u šablon*. Ovo je, naravno, poželjno svojstvo, jer ovakav poziv i ne smije biti dozvoljen. Međutim, ovaj pristup posjeduje i ozbiljan nedostatak. Zamislimo da funkciju “`IspisiNiz`” želimo iskoristiti da ispišemo elemente nekog

vektora, kao u sljedećoj sekvenci naredbi:

```
vector<double> a(10);
...
IspisiNiz(a, 10);
```

Ovaj poziv će ponovo biti odbijen. Naime, vektor *liči na niz*, i ponaša se slično kao niz, ali vektor nije niz, dok generička funkcija “IspisiNiz” zahtijeva da prvi parametar bude niz. Ukoliko želimo da funkcija koja se zove isto i koja se poziva na isti način može raditi i sa vektorima, jedna mogućnost je da napišemo preklopljenu verziju generičke funkcije “IspisiNiz” koja bi kao prvi parametar primala vektore (čiji su elementi neodređenog tipa), koja bi mogla izgledati ovako:

```
template <typename IspisiviObjekat>
void IspisiNiz(vector<IspisiviObjekat> niz, int broj_elemenata) {
    for(int i = 0; i < broj_elemenata; i++) cout << niz[i] << " ";
```

Prethodni poziv sada ispravno radi, pri čemu dedukcija tipa zaključuje da je značenje parametra šablona “IspisiviObjekat” tip “double”, s obzirom da je stvarni parametar “a” tipa “vector<double>”.

Sada se prirodno nameće pitanje da li je moguće napraviti takvu generičku funkciju “IspisiNiz” koja bi radila i sa nizovima i sa vektorima, bez potrebe da ručno definiramo posebne preklopljene verzije za nizove i vektore. Odgovor je potvrđan ukoliko umjesto djelimične upotrijebimo potpunu dedukciju tipa. Naime, nećemo kompjleru dati nikakvu uputu šta bi prvi parametar trebao da predstavlja, nego ćemo pustiti da kompletnu informaciju izvuče sam prilikom poziva funkcije. Definicija bi, stoga, trebala izgledati poput sljedeće (primijetimo odsustvo uglastih zagrada pri deklaraciji parametra “niz”):

```
template <typename NekiTip>
void IspisiNiz(NekiTip niz, int broj_elemenata) {
    for(int i = 0; i < broj_elemenata; i++) cout << niz[i] << " ";
```

Razmotrimo sada kako izgleda dedukcija tipa u slučaju sekvence naredbi:

```
int a[10] = {3, 4, 0, 8, 1, -6, 1, 4, 2, -7};
IspisiNiz(a, 10);
```

Kompajler će zaključiti da se može uklopiti u šablon jedino ukoliko prepostavi da parametar šablona “NekiTip” predstavlja ne tip “int”, već tip *niza cijelih brojeva* (odnosno neki polu-anonimni tip koji smo označavali kao “int []”, a koji bi se mogao konkretno imenovati “typedef” naredbom), jer je jedino uz takvu pretpostavku moguće formalni parametar “niz” deklarirati kao niz, bez navođenja uglastih zagrada. Prema tome, kompjajler će ispravno zaključiti da “NekiTip” predstavlja nizovni tip, i ispravno će deklarirati formalni parametar “niz”. Slično, u sekvenci naredbi

```
vector<double> a(10);
...
IspisiNiz(a, 10);
```

kompajler zaključuje da parametar šablona “NekiTip” predstavlja vektor realnih brojeva, odnosno tip “vector<double>”. Slijedi da će formalni parametar “niz” ponovo biti deklariran ispravno, tako da će funkcija raditi ispravno kako sa nizovima, tako i sa vektorima.

Potpuna dedukcija tipa ipak ima i jedan neželjeni propratni efekat. Uz potpunu dedukciju tipa, u slučaju besmislenog poziva poput

```
IspisiNiz(5, 3);
```

kompajler će ipak pokušati instantaciju, jer se ovakav poziv može uklopiti u šablon, ukoliko se pretpostavi da parametar šablona "NekiTip" predstavlja tip "**int**". Formalni parametar "niz" će, prema tome, biti deklariran sa tipom "**int**". Greška će biti uočena kada se pokuša izvršiti *pristup nekom elementu* parametra "niz", jer on nije niz, pa se na njega ne može primijeniti indeksiranje. Drugim riječima, greška će biti uočena tek kada se sa parametrom pokuša izvesti operacija koja na njega nije primjenljiva (indeksiranje). Zbog toga, potpunu dedukciju treba koristiti samo u slučaju kada je zaista potrebna velika općenitost generičke funkcije. U slučaju posjedovanja dijela informacije o očekivanom tipu (npr. da je on *niz*), poželjno je koristiti djelimičnu dedukciju.

Posljednji primjer funkcije "IspisiNiz" će kao prvi parametar prihvati bilo koji objekat na koji se može primijeniti indeksiranje (za sada znamo da je to moguće za nizove i vektore). Možemo reći da ova funkcija kao prvi parametar zahtijeva objekat koji zadovoljava *koncept objekata koji se mogu indeksirati*. Kako se objekti koji se mogu indeksirati također nazivaju i *kontejneri sa direktnim pristupom*, prvi parametar ove funkcije mora zadovoljavati *koncept kontejnera sa direktnim pristupom*. Na primjer, nizovi i vektori predstavljaju *modele koncepta kontejnera sa direktnim pristupom*.

Treba napomenuti da se u slučaju generičkih funkcija ne vrši nikakva automatska konverzija tipova čak ni za one parametre funkcije koji su jasno određeni, tj. koji ne zavise od parametara šablona. Tako, prilikom poziva generičke funkcije "IspisiNiz", drugi parametar *mora biti cijeli broj*. Drugim riječima, ukoliko kao drugi parametar navedemo realni broj, biće prijavljena greška, dok bi u slučaju da je "IspisiNiz" obična funkcija, bila izvršena konverzija u tip "**int**" (odsječanjem decimala).

Mnogi primjeri koje smo razmatrali u prethodnom poglavlju mogu se učiniti fleksibilnijim pretvaranjem običnih funkcija u generičke funkcije. Razmotrimo, na primjer, funkciju "NadjiElement" koju smo demonstrirali u prethodnom poglavlju, koja pronalazi i vraća kao rezultat indeks traženog elementa u nizu, odnosno vrijednost konstante "NEMA_GA" (čija je vrijednost bila definirana kao -1) u slučaju da traženi element nije nađen. Ova funkcija je bila ograničena na traženje *cjelobrojnih vrijednosti* u nizu *cijelih brojeva*. Slijedi generička verzija ove funkcije, koja traži vrijednosti *proizvoljnog tipa* u nizu elemenata *istog takvog tipa* (s obzirom da je korištena djelimična dedukcija, funkcija radi *samo sa nizovima*):

```
template <typename NekiTip>
int NadjiElement(const NekiTip lista[], NekiTip element, int n) {
    for(int i = 0; i < n; i++)
        if(lista[i] == element) return i;
    return NEMA_GA;
}
```

Ne bi bilo teško (korištenjem potpune dedukcije) prepraviti ovu funkciju da radi sa proizvoljnim kontejnerom sa slučajnim pristupom. Sljedeći primjer demonstrira generičku funkciju koja vraća referencu na najveći element u nizu elemenata proizvoljnog tipa (odnosno na prvi najveći element u slučaju da takvih elemenata ima više), za koje se samo pretpostavlja da se mogu *porediti po veličini* (što isključuje, recimo, nizove kompleksnih brojeva):

```
template <typename UporediviTip>
UporediviTip &MaxElement(const UporediviTip niz[], int br_elemenata) {
    int indeks(0);
    for(int i = 1; i < br_elemenata; i++)
        if(niz[i] > niz[indeks]) indeks = i;
    return niz[indeks];
```

}

Generičke funkcije imaju najveću primjenu za potrebe pravljenja biblioteka funkcija za višestruku upotrebu u drugim programima, a koje mogu raditi sa različitim tipovima podataka, čija tačna priroda nije poznata prilikom pisanja same funkcije. Veliki broj funkcija koje se nalaze u bibliotekama koje po standardu čine integralni dio jezika C++ izvedene su upravo kao generičke funkcije. Sa nekim od tih funkcija ćemo se detaljnije upoznati kasnije.

20. Primjeri modularnog dizajna

Programiranje predstavlja jednu od naučnih disciplina koje se uče pretežno kroz praktičan rad. Stoga se programiranje, bez obzira na stepen usvojenog teorijskog znanja, ne može savladati bez samostalne izrade većeg broja programa, počev od najlakših, ka složenijim programima. Međutim, sastavljanje iole složenijih programa često za početnika predstavlja nepremostivu prepreku, jer se nerijetko javlja pitanje odakle uopće početi. U ovom poglavlju ćemo na konkretnim primjerima dati osnovne smjernice kako bi trebalo pristupiti razvoju složenijih programa koristeći metodologiju modularnog dizajniranja. Izlaganje ćemo započeti primjerom konkretnog problema iz prakse za koji se traži računarski program:

Neka kompanija prodaje više različitih vrsta hrane za životinje. Poljoprivrednici mogu telefonom izvršiti narudžbu i podići je sljedeći dan. Kompanija trenutno ima 5 proizvoda na lageru, a cijene su sljedeće:

Šifra proizvoda	1	2	3	4	5
Cijena proizvoda (u KM po toni)	150	200	400	325	250

Kompaniji je potreban program za računar koji će operateru omogućiti da unese podatke o svakoj narudžbi: šifru svakog proizvoda kojeg poljoprivrednik želi kupiti, i broj naručenih tona. Program treba na osnovu toga izračunati ukupnu cijenu narudžbe, tako da se ta informacija može odmah dati kupcu. Na kraju dana, kada nema više narudžbi, program treba da prikaže sumarni izvještaj koji prikazuje ukupan broj naručenih tona za svaki proizvod.

Prikazaćemo kako bi trebao izgledati sistematičan pristup razvoju programa koji rješava ovaj problem. Na prvom mjestu, treba osmisliti kako bi program *trebao da izgleda* i kako bi *trebao da komunicira sa korisnikom*. Radi jednostavnosti, odlučimo se za varijantu programa koji će nakon svake narudžbe postavljati pitanje DA/NE tipa da li operater želi unijeti novu narudžbu. Ako je odgovor “DA”, program će čekati unos sljedeće narudžbe, a ako je odgovor “NE”, on će prikazati sumarni izvještaj. Pri unosu narudžbe, program će neprekidno tražiti unos šifre naručenog proizvoda i naručenu količinu (u tonama). Pošto ne postoji proizvod čija je šifra 0, to se unos šifre 0 može koristiti kao signal da je narudžba gotova.

Prva stvar koju treba uočiti pri razvoju programa je *koje su nam strukture podataka neophodne za čuvanje podataka koji se trebaju obrađivati*. Jasno je da je za smještanje popisa cijena najpogodnije koristiti niz od 5 elemenata (broj 5 je dobro definirati kao imenovanu konstantu, npr. “BrojProizvoda”, jer je tako lakše napraviti izmjenu programa ukoliko se broj proizvoda promijeni). Ovaj niz može biti konstantan, s obzirom da su cijene fiksne. Kako će informacije o cijenama biti potrebne u čitavom programu, prirodno je ovaj niz deklarirati sa globalnom vidljivošću, što je bezbjedno s obzirom da se radi o konstantnom nizu:

```
const int BrojProizvoda(5);
const double cijene[BrojProizvoda] = {150, 200, 400, 325, 250};
```

Pored toga, potreban nam je i niz koji će čuvati ukupne naručene količine za svaki proizvod (nazovimo ga npr. “kolicine”). Naravno, ovaj niz ne može biti konstantan. Mada će se i ovaj niz koristiti u čitavom programu, nije dobra ideja deklarirati ga sa globalnom vidljivošću, s obzirom da mu sadržaj nije konstantan. Bolje ga je deklarirati kao lokalnu promjenljivu u glavnom programu, i po potrebi prenositi ga u dijelove programa koji mu trebaju pristupiti putem mehanizma prenosa parametara. Stoga ćemo tako i postupiti pri pisanju glavne funkcije.

Analizirajmo prvo grubo od kakvih bi se cjelina trebao sastojati program. Možemo primjetiti da se u

programu traže dvije funkcionalnosti. Prva funkcionalnost je unos i obrada narudžbi, dok je druga funkcionalnost prikaz sumarnog izvještaja nakon što se obrade sve narudžbe. Prirodno je ove dvije funkcionalnosti realizirati kao odvojene potprograme, koje možemo nazvati "ObradiNarudzbe" i "PrikaziIzvjestaj". Oba potprograma kao jedini parametar zahtijevaju niz u kojem se čuvaju podaci o naručenim količinama. Jasno je da će potprogramu "ObradiNarudzbe" taj parametar biti *izlazni* (tj. potprogram će *napuniti* taj parametar neophodnim informacijama), dok će potprogramu "PrikaziIzvjestaj" taj parametar biti *ulazni* (tj. potprogram će informacije pohranjene u tom parametru iskoristiti za prikaz izvještaja). Odavde slijedi izgled glavne funkcije programa:

```
int main() {  
    double kolicine[BrojProizvoda];  
  
    void ObradiNarudzbe(double kolicine[BrojProizvoda]);  
    void PrikaziIzvjestaj(const double kolicine[BrojProizvoda]);  
  
    ObradiNarudzbe(kolicine);  
    PrikaziIzvjestaj(kolicine);  
  
    return 0;  
}
```

Sada je potrebno razraditi potprograme "ObradiNarudzbe" i "PrikaziIzvjestaj". U našem slučaju, potprogram "ObradiNarudzbe" predstavlja najsloženiji dio programa. Razmislimo šta bi ovaj potprogram trebao da radi. Prije prve narudžbe, sve naručene količine sigurno treba da budu jednake nuli. Potprogram zatim treba da prihvata jednu po jednu narudžbu, sve dok korisnik (operator) ne kaže da ne želi više narudžbi. Da pojednostavimo unos komande, prepostavimo da unos slova "N" ili "n" predstavlja odgovor "NE", dok svako drugi znak predstavlja odgovor "DA". Uz ovakve prepostavke, algoritam koji obavlja potprogram "ObradiNarudzbe" može se prikazati ovako:

- Postavi ukupne količine na nulu;
 - Radi sljedeće:
 - Obradi jednu narudžbu;
 - Traži unos komande od korisnika;
 - Ponavljam gore navedene akcije sve dok je korisnikova komanda različita od 'N' ili 'n'.

Prilikom formiranja algoritma, trebamo se truditi da prilikom izražavanja pojedinih koraka algoritma koristimo takve iskaze koji se mogu lako prevesti u instrukcije jezika C++ (na primjer, fraze poput “**ako**”, “**sve dok je**” itd. lako se prevode u naredbe poput “**if**”, “**while**”, itd.). Napisani algoritam se lako prevodi u jezik C++, jer su svi njegovi iskazi posve jednostavni, osim iskaza “*Postavi ukupne količine na nulu*” (koja zahtijeva petlju), i ključnog iskaza “*Obradi jednu narudžbu*”, koji zahtijeva detaljniju razradu. Odlučimo se stoga da ova dva koraka realiziramo kao posebne potprograme, sa imenima “*PostaviNaNulu*” i “*ObradiJednuNarudzbu*”. Oba potprograma će ponovo imati niz naručenih količina kao parametar, pri čemu će u slučaju potprograma “*PostaviNaNulu*” taj parametar biti striktno izlaznim dok će u slučaju potprograma “*ObradiJednuNarudzbu*” njegov parametar biti ulazno-izlaznog tipa, s obzirom da se u njemu vrši *ažuriranje* postojećih informacija o naručenim količinama u skladu sa novoizvršenom narudžbom. Stoga bi potprogram “*ObradaNarudzbi*” mogao izgledati ovako:

```
void ObradiNarudzbe(double kolicine[BrojProizvoda]) {  
    void PostaviNaNulu(double kolicine[BrojProizvoda]);  
    void ObradiJednuNarudzbu(double kolicine[BrojProizvoda]);  
  
    PostaviNaNulu(kolicine);  
  
    char komanda; // Korisnikova komanda
```

```

do {
    ObradiJednuNarudzbu(kolicine);
    cout << "Da li želite još narudžbi (D/N) ? ";
    cin >> komanda;
    cin.ignore(10000, '\n');
} while(komanda != 'N' && komanda != 'n');
}

```

Poziv “`cin.ignore()`” iskorišten je za brisanje suvišnih znakova koji su eventualno ostali u ulaznom toku (npr. ukoliko korisnik nije unio samo slovo “D”, već cijelu riječ “DA”). Primijetimo da je u uvjetu “**do** – “**while**” petlje upotrijebljen operator “**&&**” iako je u opisu algoritma korištena riječ “ili” a ne riječ “i”. Ovo je posljedica *nedoslijednosti* sa kojom se u nekim kontekstima riječ “ili” koristi u našem jeziku. Na primjer, kada kažemo da “*x ne smije biti 3 ili 5*”, mi zapravo mislimo da “*x ne smije biti niti 3 niti 5*”, što u stvari znači da “*x ne smije biti 3 i x ne smije biti 5*”. Stoga je logička operacija koja veže dva uvjeta u ovoj rečenici zapravo “i” a ne “ili”.

Razradimo sada potprograme “`PostaviNaNulu`” i “`ObradiJednuNarudzbu`”. Prvi od ova dva potprograma je trivijalan, i ne traži specijalnu razradu, jer se svodi na običnu “**for**” petlju kakvu smo do sada koristili mnogo puta:

```

void PostaviNaNulu(double kolicine[BrojProizvoda]) {
    for(int i = 0; i < BrojProizvoda; i++) kolicine[i] = 0;
}

```

Ovaj potprogram je dovoljno jednostavan da u suštini nije morao ni da bude posebno izdvojen (“**for**” petlju koja čini njegovo tijelo mogli smo pisati direktno u potprogramu “`ObradiNarudzbe`”). Ipak, sa aspekta metodologije modularnog programiranja, dobro je svakom poslu dodijeliti odgovarajući potprogram. Na taj način se program lakše modificira i održava.

Razradimo sada šta bi trebao da radi potprogram “`ObradiJednuNarudzbu`”. Nakon malo razmišljanja, lako možemo zaključiti da se njegov rad može opisati sljedećim algoritmom:

- *Postavi ukupnu cijenu narudžbe na nulu;*
- *Učitaj šifru proizvoda;*
- **Sve dok je šifra proizvoda različita od nule:**
 - *Učitaj zahtijevanu količinu;*
 - *Dodaj zahtijevanu količinu na ukupnu količinu za taj proizvod;*
 - *Izračunaj cijenu stavke kao proizvod količine i cijene po toni;*
 - *Ispisi cijenu stavke;*
 - *Dodaj izračunatu cijenu na ukupnu cijenu narudžbe;*
 - *Učitaj šifru proizvoda;*
- *Ispisi ukupnu cijenu narudžbe.*

Mada se svaki od koraka ovog algoritma može lako neposredno prevesti u naredbe jezika C++, nije loša ideja korake “*Učitaj šifru proizvoda*” i “*Učitaj zahtijevanu količinu*” realizirati kao posebne funkcije, jer se njima može povjeriti i zaštita od unosa pogrešnih podataka. Ukoliko ove funkcije realiziramo kao funkcije bez parametara koje vraćaju kao rezultat unijetu vrijednost, traženi potprogram može izgledati ovako:

```

void ObradiJednuNarudzbu(double kolicine[BrojProizvoda]) {
    int UnesiSifru();                                // Prototipovi
    double UnesiKolicinu();
}

```

```

double ukupno(0);
int sifra = UnesiSifru();

while(sifra != 0) {
    double kolicina = UnesiKolicinu();
    kolicine[sifra - 1] += kolicina;
    double cijena = kolicina * cijene[sifra - 1];
    cout << "Ta stavka će koštati " << cijena << "KM.\n";
    ukupno += cijena;
    sifra = UnesiSifru();
}
cout << "Ukupna cijena narudžbe iznosi: " << ukupno << "KM.\n";
}

```

Primijetimo da prilikom pristupa elementima niza šifru koju je unio korisnik umanjujemo za 1, s obzirom da su šifre proizvoda u opsegu od 1 – 5 (ne računajući nulu koja označava kraj), dok su indeksi niza u kojem treba čuvati odgovarajuće podatke u opsegu od 0 – 4.

Funkcija “UnesiSifru” ne smije dozvoliti da se unese šifra koja nije u opsegu od 0 – 5, dok funkcija “UnesiKolicinu” ne smije dozvoliti unos negativne količine. Naravno, obje funkcije trebaju provjeravati da li je zaista unesen broj. Pored toga, obje funkcije će uvijek prije povratka isprazniti ulazni tok, tako da ćemo uvijek biti sigurni da se iz ulaznog toka čitaju “svježi” podaci. Kako smo funkcije poput ovih već pisali, nećemo razraditi njihovu algoritamsku strukturu, već ćemo samo prikazati njihov mogući izgled:

```

int UnesiSifru() {
    for(;;) {
        int sifra;
        cout << "Unesite šifru proizvoda (0 za kraj): ";
        cin >> sifra;
        bool dobar_unos = cin && sifra >= 0 && sifra <= BrojProizvoda;
        if(!cin) cin.clear();
        cin.ignore(10000, '\n');
        if(dobar_unos) return sifra;
        cout << "Neispravan unos!\n";
    }
}

double UnesiKolicinu() {
    for(;;) {
        double kolicina;
        cout << "Unesite željenu količinu: ";
        cin >> kolicina;
        bool dobar_unos = cin && kolicina >= 0;
        if(!cin) cin.clear();
        cin.ignore(10000, '\n');
        if(dobar_unos) return kolicina;
        cout << "Neispravan unos!\n";
    }
}

```

U funkciji “UnesiSifru”, umjesto broja 5 za testiranje ispravnosti opsega iskorištena je vrijednost konstante “BrojProizvoda”. Na taj način, funkciju ne treba mijenjati ukoliko se promijeni broj proizvoda u postavci zadatka.

Preostao je još potprogram “StampajIzvjestaj”. U principu, ovaj potprogram se svodi na ispis zaglavlja izvještaja, iza kojeg treba ispisati podatke o ukupnim naručenim količinama, što se ponovo svodi

na običnu “**for**” petlju. Stoga nema potrebe da ovaj potprogram prvo opisujemo algoritamski, već možemo odmah prikazati gotovu verziju u formi jezika C++. Naravno, ponovo ne smijemo zaboraviti da su šifre proizvoda za 1 veće nego indeksi u kojima se odgovarajući podaci čuvaju u nizu:

```
void StampajIzvjestaj(const double kolicine[BrojProizvoda]) {
    cout << "Proizvod      Prodata količina (tona)\n"
        "----- ----- \n";
    for(int i = 0; i < BrojProizvoda; i++)
        cout << setw(8) << i + 1 << setw(28) << kolicine[i] << endl;
}
```

Ovim je razvoj programa završen. Da bismo dobili funkcionalnu verziju programa, dovoljno je samo sve navedene funkcije sklopiti u program onim redoslijedom kako su ovdje napisane, i dodati uključivanje zaglavlja biblioteka “*iostream*” i “*omanip*” koje su nam neophodne. Međutim, nakon sklapanja programa, nije loše razmotriti da li je moguće izvršiti neku očiglednu optimizaciju u programu. U našem slučaju možemo primijetiti da se potprogram “*PostaviNaNulu*” može u potpunosti izbjegći ukoliko niz “*kolicine*” inicijaliziramo na nulu odmah pri deklaraciji. Parametar “*kolicine*” funkcije “*ObradiNarudzbe*” tada više nije čisto izlazni, već *postaje ulazno-izlazni*, jer funkcija očekuje da će na ulazu dobiti već inicijaliziran niz (što ranije nije bio slučaj, nego je sama funkcija vršila njegovu inicijalizaciju). Konačna verzija programa sada izgleda ovako:

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

using namespace std;

const int BrojProizvoda(5);
const double cijene[BrojProizvoda] = {150, 200, 400, 325, 250};

int main() {
    double kolicine[BrojProizvoda] = {};// Inicijalizacija na nulu
    void ObradiNarudzbe(double kolicine[BrojProizvoda]);
    void StampajIzvjestaj(const double kolicine[BrojProizvoda]);
    ObradiNarudzbi(kolicine);
    StampajIzvjestaj(kolicine);

    return 0;
}

void ObradiNarudzbe(double kolicine[BrojProizvoda]) {
    void ObradiJednuNarudzbu(double kolicine[BrojProizvoda]);
    char komanda;
    do {
        ObradiJednuNarudzbu(kolicine);
        cout << "Da li želite još narudžbi (D/N)? ";
        cin >> komanda;
        cin.ignore(10000, '\n');
    } while(komanda != 'N' && komanda != 'n');
}

void ObradiJednuNarudzbu(double kolicine[BrojProizvoda]) {
    int UnesiSifru();
    double UnesiKolicinu();
    double ukupno(0);
    int sifra = UnesiSifru();
    while(sifra != 0) {
```

```

    double kolicina = UnesiKolicinu();
    kolicine[sifra - 1] += kolicina;
    double cijena = kolicina * cijene[sifra - 1];
    cout << "Ta stavka će koštati " << Cijena << "KM.\n";
    ukupno += cijena;
    sifra = UnesiSifru();
}
cout << "Ukupna cijena narudžbe iznosi: " << Ukupno << "KM.\n";
}

int UnesiSifru() {
    for(;;) {
        int sifra;
        cout << "Unesite šifru proizvoda (0 za kraj): ";
        cin >> Sifra;
        bool dobar_unos = cin && sifra >= 0 && sifra <= BrojProizvoda;
        if(!cin) cin.clear();
        cin.ignore(10000, '\n');
        if(dobar_unos) return sifra;
        cout << "Neispravan unos!\n";
    }
}

double UnesiKolicinu() {
    for(;;) {
        double kolicina;
        cout << "Unesite željenu količinu: ";
        cin >> kolicina;
        bool dobar_unos = cin && kolicina >= 0;
        if(!cin) cin.clear();
        cin.ignore(10000, '\n');
        if(dobar_unos) return Kolicina;
        cout << "Neispravan unos!\n";
    }
}

void StampajIzvjestaj(const double kolicine[BrojProizvoda]) {
    cout << "Proizvod      Prodata količina (tona)\n"
        "-----      -----\\n";
    for(int i = 0; i < BrojProizvoda; i++)
        cout << setw(8) << i + 1 << setw(28) << kolicine[i] << endl;
}

```

Sljedeći primjer demonstrira modularni razvoj programa, u kojem se postavlja manje zahtjeva nego u prethodnom primjeru, ali je složeniji sa logičkog stanovištva. Kao i u prvom primjeru, započećemo sa opisom problema:

Potrebno je sastaviti program za računar koji će na ekranu odštampati kompletan kalendar (za sve mjesecе) za proizvoljnu unesenu godinu veću od 1997. Kalendar treba biti formiran u skladu sa gregorijanskom konvencijom, po kojoj su prestupne godine sve godine koje su djeljive sa 4, osim one godine na početku svakog vijeka koje nisu djeljive sa 400 (tako npr. godina 2100 nije presupna).

Komunikacija sa korisnikom ovdje je krajnje prosta: korisnik unosi godinu, i dobija prikaz kalendar-a. Slijedi da se na njoj ne trebamo zadržavati. Međutim, algoritam po kojem se formira kalendar traži detaljniju razradu. Da bismo olakšali razvoj i čitljivost programa, koristićemo pobrojane tipove podataka, gdje god je to moguće. Ima smisla uvesti tipove podataka “Dani” koji predstavljaju dane u sedmici, i “Mjeseci” koji predstavljaju mjesecе u godini. Pošto će ovi tipovi podataka vjerovatno biti potrebni u

čitavom programu, deklariraćemo ih sa globalnom vidljivošću:

```
enum Dani {Ponedjeljak, Utorka, Srijeda, Cetvrtak, Petak, Subota,  
Nedjelja};  
  
enum Mjeseci {Januar, Februar, Mart, April, Maj, Juni, Juli, August,  
Septembar, Oktobar, Novembar, Decembar};
```

Razmislimo sada šta je neophodno za rješavanje postavljenog problema. Prvo treba primijetiti da su za formiranje kalendarja potrebne samo dvije informacije: *kojim danom započinje godina i da li je godina prestupna ili nije*. Ukoliko poznajemo ove dvije informacije, *uopće nije potrebno da znamo o kojoj se godini zaista radi*. Prema tome, osnovni problem je odrediti ove dvije informacije. Stoga bi gruba verzija algoritma za rješavanje ovog problema mogla izgledati ovako:

- *Unesi godinu;*
- *Odredi početni dan godine;*
- *Odredi da li je godina prestupna;*
- *Štampaj kalendar (na osnovu nađenih informacija).*

Svakom od ova četiri koraka algoritma dodijelićemo posebnu funkciju. Funkcija “UnesiGodinu” traži od korisnika godinu, uz provjeru ispravnosti podataka, i vraća unesenu godinu kao rezultat. Funkcija “PocetniDanGodine” prihvata kao parametar godinu za koju se određuje početni dan, i kao rezultat vraća dan kojim započinje godina (rezultat je tipa “Dani”). Funkcija “DaLiJePrestupna” vraća logičku vrijednost da li je funkcija zadana kao parametar prestupna ili nije. Konačno, funkcija “StampajKalendar” štampa kalendar, na osnovu informacija o početnom danu i o tome da li je godina prestupna, koje joj se prenose kao parametri. U skladu s tim, glavna funkcija bi mogla izgledati ovako:

```
int main() {  
    int UnesiGodinu();                                     // Prototipovi  
    Dani PocetniDanGodine(int godina);  
    bool DaLiJePrestupna(int godina);  
    void StampajKalendar(Dani pocetni_dan, bool prestupna);  
  
    int godina = UnesiGodinu();  
    StampajKalendar(PocetniDanGodine(godina), DaLiJePrestupna(godina));  
  
    return 0;  
}
```

Unos godine uz provjeru ispravnosti ulaznih podataka je rutinski posao, tako da ćemo funkciju “UnesiGodinu” samo prikazati, bez detaljnijih objašnjenja:

```
int UnesiGodinu() {  
    for(;;) {  
        int godina;  
        cout << "Unesi godinu u opsegu (od 1997 nadalje): ";  
        cin >> godina;  
        if(cin && godina >= 1997) return godina;  
        if(!cin) cin.clear();  
        cout << "Neispravan unos!\n";  
        cin.ignore(10000, '\n');  
    }  
}
```

Razmotrimo sada kako se određuje početni dan godine. Za tu svrhu potrebno je odrediti koliko je dana

proteklo od 1. januara neke godine za koji znamo kojim je danom započela (zvaćemo je *referentna godina*), do 1. januara tražene godine. Tada će ostatak dijeljenja sa 7 nađenog broja dana odrediti početni dan godine. Naime, ostatak 0 govori da su obje godine započele u isti dan, ostatak 1 govori da je tražena godina započela u naredni dan u odnosu na referentnu, itd. Naravno, referentna godina, s obzirom na uvjete zadatka, mora biti *manja* od 1997. Najbolje je za referentnu godinu uzeti neku godinu za koju znamo da je započela *ponedjeljkom*, jer će tada ostatak 0 značiti da i tražena godina započinje ponedjeljkom, ostatak 1 označava da godina započinje utorkom, itd. Takva je npr. godina 1996. Pošto su cjelobrojne vrijednosti u koje se konvertiraju pobrojane konstante “Ponedjeljak”, “Utorak”, itd. iz pobrojanog tipa “Dani” upravo 0, 1, itd. slijedi da je dovoljno ostatak samo konvertirati u tip “Dani”.

Ostaje da vidimo kako možemo odrediti broj dana koji je protekao od 1. januara referentne godine do 1. januara zadane godine. Kada ne bi postojale prestupne godine, dovoljno bi bilo broj godina koji je protekao između referentne i zadane godine pomnožiti sa 365. Međutim, kako postoje prestupne godine, potrebno je na ovaj rezultat dodati i broj prestupnih godina koji se pojавio između ove dvije godine (uključujući eventualno i referentnu godinu, koja je u našem slučaju upravo prestupna). Ovo određivanje broja prestupnih godina je jedini netrivijalni korak u opisanom postupku, pa ćemo ga realizirati posebnom funkcijom “BrojProteklihPrestupnihGodina”, koja vraća kao rezultat broj prestupnih godina u periodu od referentne godine, do godine koja je zadata kao parametar. U skladu sa ovom razradom, funkcija “PocetniDanGodine” mogla bi izgledati ovako:

```
Dani PocetniDanGodine(int godina) {
    int BrojProteklihPrestupnihGodina(int godina);
    long int broj_proteklih_dana = (godina - 1996) * 365
        + BrojProteklihPrestupnihGodina(godina);
    return Dani(broj_proteklih_dana % 7);
}
```

Sljedeći problem je odrediti broj prestupnih godina u intervalu od referentne do zadane godine. Najjednostavniji način je da pomoću “**for**” petlje prosto *prebrojimo* prestupne godine u traženom intervalu, koristeći funkciju “DaLiJePrestupna” koja određuje da li je godina prestupna ili nije:

```
int BrojProteklihPrestupnihGodina(int godina) {
    bool DaLiJePrestupna(int godina);
    int brojac(0);
    for(int i = 1996; i < godina; i++)
        if(DaLiJePrestupna(i)) brojac++;
    return brojac;
}
```

Ukoliko iskoristimo činjenicu da se logičke vrijednosti automatski konvertiraju u cjelobrojne vrijednosti 0 ili 1, istu funkciju možemo kompaktnije napisati ovako:

```
int BrojProteklihPrestupnihGodina(int godina) {
    bool DaLiJePrestupna(int godina);
    int brojac(0);
    for(int i = 1996; i < godina; i++) brojac += DaLiJePrestupna(i);
    return brojac;
}
```

Ovako napisana funkcija sigurno će raditi ispravno. Međutim, njena realizacija zasniva se na onome što se u programiranju naziva *gruba sila* (engl. *brute force*). Pod ovim nazivom smatramo neinteligentna rješenja koja su zasnovana na prostom ispitivanju svih mogućih situacija, ili prebrojavanju svih

mogućnosti. Rješenja zasnovana na gruboj sili nekada nisu loša, a nekada su čak i jedina moguća rješenja. Međutim, u ogromnom broju slučajeva, rješenja zasnovana na gruboj sili su strahovito neefikasna, jer treba ispitati ogroman broj mogućnosti.

Da bismo demonstrirali kako je, uz malo razmišljanja, često moguće eliminirati grubu silu, razmotrimo kako bi se funkcija “`BrojProteklihPrestupnihGodina`” mogla realizirati bez grube sile. Primijetimo prvo da je, u skladu sa pravilima gregorijanskog kalendarja, broj prestupnih godina u intervalu između dvije godine jednak broju godina u tom intervalu koje su djeljive sa 4, umanjen za broj godina koje su djeljive sa 100, a nisu djeljive sa 400. Slijedi da je osnovni problem odrediti broj godina u traženom intervalu koje su djeljive sa 4, 100 i 400. Ukoliko tekuću godinu nazovemo g , uz malo elementarne matematike lako možemo izvesti da je broj godina djeljivih sa 4 u intervalu od godine 1996. do godine g dat formulom $1 + (g - 1997) / 4$, pri čemu se dijeljenje posmatra kao cjelobrojno dijeljenje. Za broj godina dijeljivih sa 100 i 400 u istom intervalu, analognim rezonovanjem dolazimo do formula $(g - 1901) / 100$ i $(g - 1601) / 400$. Slijedi da bi funkcija “`BrojProteklihPrestupnihGodina`” mogla izgledati ovako:

```
int BrojProteklihPrestupnihGodina(int godina) {
    int djeljive_sa_4 = 1 + (godina - 1997) / 4;
    int djeljive_sa_100 = (godina - 1901) / 100;
    int djeljive_sa_400 = (godina - 1601) / 400;
    return djeljive_sa_4 - (djeljive_sa_100 - djeljive_sa_400);
}
```

Ovim je funkcija postala neuporedivo efikasnija, jer se u njoj uopće ne koriste petlje.

Do sada smo se više puta oslanjali na postojanje funkcije “`DaLiJePrestupna`”, koju nismo još napisali. Međutim, definicija ove funkcije je posve trivijalna: potrebno je samo pravilo gregorijanskog kalendarja koje govori kada je godina prestupna prevesti u ispravnu formulu jezika C++:

```
bool DaLiJePrestupna(int godina) {
    return godina % 4 == 0 && (godina % 100 != 0 || godina % 400 == 0);
}
```

Predimo sada na najteži dio posla: formiranje i štampanje samog kalendarja. Štampanje kalendarja za čitavu godinu sigurno se svodi na štampanje kalendarja za svaki od 12 mjeseci, što ćemo povjeriti funkciji “`StampajKalendarZaJedanMjesec`”. Ova funkcija će primati tri parametra: početni dan mjeseca, sam mjesec (da bi se moglo ispisati njegovo ime i odrediti koliko dana ima u mjesecu), i informaciju o tome da li je godina prestupna (ova informacija je potrebna samo ukoliko je mjesec februar). Ovdje se javlja jedan problem: funkcija “`StampajKalendar`” dobija samo informaciju o tome kojim danom počinje $godina$, ali ne i svaki mjesec. Ovaj problem se lako rješava ukoliko uočimo da je početni dan svakog mjeseca sljedeći dan u odnosu na dan kojim se završio prethodni mjesec. Stoga je najprirodnije rješenje napraviti funkciju “`StampajKalendarZaJedanMjesec`” takvom da automatski ažurira vrijednost svog prvog argumenta tako da nakon što se završi ispis kalendarja za jedan mjesec, on sadrži početni dan narednog mjeseca. Naravno, za tu svrhu, prvi parametar mora se prenositi po referenci. Nakon ovih razmatranja, funkcija “`StampajKalendar`” dobija sljedeći oblik:

```
void StampajKalendar(Dani pocetni_dan, bool prestupna) {
    void StampajKalendarZaJedanMjesec(Dani &pocetni_dan, Mjeseci mjesec,
                                         bool prestupna);
    for (Mjeseci m = Januar; m <= Decembar; m = Mjeseci(m + 1))
```

```

        StampajKalendarZaJedanMjesec(pocetni_dan m, prestupna);
    }
}

```

Najkomplikovanija funkcije je upravo funkcija "StampajKalendarZaJedanMjesec". Funkciju veoma sličnu ovoj, pod imenom "StampajKalendar", već smo pisali u poglavljju o prenosu parametara u potprograme (nemojte ovu funkciju pomiješati sa istoimenom funkcijom koju pišemo ovdje). Na početku, svakako treba ispisati naziv mjeseca (što je povjereno funkciji "IspisiMjesec" koja kao parametar prima mjesec čije ime treba ispisati) i zaglavlje kalendara ("Pon Uto Sri Čet Pet Sub Ned"). U narednom redu, ispis treba započeti u odgovarajućoj koloni, koja zavisi od početnog dana. Ovo je lako podesiti, pomoću manipulatora "setw". Naime, broj praznih polja koje treba ispisati jednak je širini kolone u znakovima (4 u našem slučaju) pomnožen sa rednim brojem početnog dana. Sad započinje petlja u kojoj ispisujemo redne brojeve dana od 1 do broja dana u mjesecu. Određivanje broja dana u mjesecu povjereno je funkciji "BrojDanaUMjesecu", koja kao parametre prima mjesec i informaciju da li je godina prestupna (zbog februara). Unutar petlje, naravno, treba ispisivati redne brojeve dana (formatiran prema širini kolone), ali pri tome treba voditi računa kada treba preći u novi red. Najlakše je u svakom prolazu kroz petlju ažurirati informaciju o tekućem danu da sadrži naredni dan. Pravi trenutak za ispis novog reda je kada je tekući dan bio nedjelja, tako da naredni dan postaje ponедjeljak. Ovaj pristup ima dopunsku prednost što će na kraju tekući dan upravo biti dan kojim započinje sljedeći mjesec. Na kraju, da kalendari za različite mjesece ne bi bili sljepljeni jedan na drugi, poželjno je ispisati jedan prazan red. Međutim, ovaj ispis ne treba vršiti ukoliko naredni mjesec započinje ponedjeljkom (odnosno ukoliko se tekući mjesec završava nedjeljom), jer je u tom slučaju prazan red već isписан, kao posljedica prebacivanja u novi red nakon svake nedjelje). U skladu sa svim provedenim razmatranjima, funkcija "StampajKalendarZaJedanMjesec" mogla bi izgledati ovako:

```

void StampajKalendarZaJedanMjesec(Dani &tekuci_dan, Mjeseci mjesec,
bool prestupna) {
    void IspisiMjesec(Mjeseci mjesec);
    int BrojDanaUMjesecu(Mjeseci mjesec, bool prestupna);

    IspisiMjesec(mjesec);
    cout << "\nPon Uto Sri Čet Pet Sub Ned\n"
        << setw(4 * tekuci_dan) << "";
    for(int i = 1; i <= BrojDanaUMjesecu(mjesec, prestupna); i++) {
        cout << setw(3) << i << " ";
        if(tekuci_dan != Nedjelja) tekuci_dan = Dani(tekuci_dan + 1);
        else {
            tekuci_dan = Ponедjeljak; // Iza nedjelje slijedi ponedjeljak,
            cout << endl;           // i to je pravi trenutak da se
                                     // pređe u novi red
        }
        cout << endl;             // Na kraju, treba ispisati još
        if(tekuci_dan != Ponедjeljak) // jedan prazan red, osim kad
            cout << endl;           // naredni mjesec započinje
                                     // ponedjeljkom
    }
}

```

Funkcija za ispis mjeseca je krajnje jednostavna ali glomazna. Kada upoznamo nizove stringova, vidjećemo da se ova funkcija može mnogo elegantnije rješiti. Međutim, sa dosadašnjim znanjem, jedino što možemo uraditi je upotrijebiti "switch" strukturu ili gomilu "if" naredbi:

```

void IspisiMjesec(Mjeseci mjesec) {
    switch(mjesec) {
        case Januar: cout << "Januar"; break;
        case Februar: cout << "Februar"; break;
        case Mart: cout << "Mart"; break;
    }
}

```

```

        case April: cout << "April"; break;
        case Maj: cout << "Maj"; break;
        case Juni: cout << "Juni"; break;
        case Juli: cout << "Juli"; break;
        case August: cout << "August"; break;
        case Septembar: cout << "Septembar"; break;
        case Oktobar: cout << "Oktobar"; break;
        case Novembar: cout << "Novembar"; break;
        case Decembar: cout << "Decembar";
    }
}

```

Konačno, određivanje broja dana u mjesecu je također jednostavno. Februar ima 28 ili 29 dana, ovisno o tome da li je godina prestupna ili ne. April, juni, septembar i novembar imaju 30 dana, dok svi ostali mjeseci imaju 31 dan:

```

int BrojDanaUMjesecu(Mjeseci mjesec, bool prestupna) {
    switch(mjesec) {
        case Februar: return 28 + prestupna;
        case April: case Juni: case Septembar: case Novembar: return 30;
        default: return 31;
    }
}

```

Ovim je razvoj programa završen. Preostaje još da sklopimo sve razvijene dijelove u funkcionalnu cjelinu, čime dobijamo sljedeći program:

```

#include <iostream>
#include <iomanip>

using namespace std;

enum Dani {Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota,
Nedjelja};

enum Mjeseci {Januar, Februar, Mart, April, Maj, Juni, Juli, August,
Septembar, Oktobar, Novembar, Decembar};

int main() {
    int UnesiGodinu();
    Dani PocetniDanGodine(int godina);
    bool DaLiJePrestupna(int godina);
    void StampajKalendar(Dani pocetni_dan, bool prestupna);
    int godina = UnesiGodinu();
    StampajKalendar(PocetniDanGodine(godina), DaLiJePrestupna(godina));

    return 0;
}

int UnesiGodinu() {
    for(;;) {
        int godina;
        cout << "Unesi godinu u opsegu (od 1997 nadalje): ";
        cin >> godina;
        if(cin && godina >= 1997) return godina;
        if(!cin) cin.clear();
        cout << "Neispravan unos!\n";
        cin.ignore(10000, '\n');
    }
}

```

```

}

Dani PocetniDanGodine(int godina) {
    int BrojProteklihPrestupnihGodina(int godina);
    long int broj_proteklih_dana = (godina - 1996) * 365
        + BrojProteklihPrestupnihGodina(godina);
    return Dani(broj_proteklih_dana % 7);
}

int BrojProteklihPrestupnihGodina(int godina) {
    int djeljive_sa_4 = 1 + (godina - 1997) / 4;
    int djeljive_sa_100 = (godina - 1901) / 100;
    int djeljive_sa_400 = (godina - 1601) / 400;
    return djeljive_sa_4 - (djeljive_sa_100 - djeljive_sa_400);
}

bool DaLiJePrestupna(int godina) {
    return godina % 4 == 0 && (godina % 100 != 0 || godina % 400 == 0);
}

void StampajKalendar(Dani pocetni_dan, bool prestupna) {
    void StampajKalendarZaJedanMjesec(Dani &pocetni_dan, Mjeseci mjesec,
                                         bool prestupna);
    for(Mjeseci m = Januar; m <= Decembar; m = Mjeseci(m + 1))
        StampajKalendarZaJedanMjesec(pocetni_dan, m, prestupna);
}

void StampajKalendarZaJedanMjesec(Dani &tekuci_dan, Mjeseci mjesec,
                                      bool prestupna) {
    void IspisiMjesec(Mjeseci mjesec);
    int BrojDanaUMjesecu(Mjeseci mjesec, bool prestupna);

    IspisiMjesec(mjesec);
    cout << "\nPon Uto Sri Čet Pet Sub Ned\n"
        << setw(4 * tekuci_dan) << "";
    for(int i = 1; i <= BrojDanaUMjesecu(mjesec, prestupna); i++) {
        cout << setw(3) << i << " ";
        if(tekuci_dan != Nedjelja) tekuci_dan = Dani(tekuci_dan + 1);
        else {
            tekuci_dan = Ponedjeljak;
            cout << endl;
        }
    }
    cout << endl;
    if(tekuci_dan != Ponedjeljak) cout << endl;
}

void IspisiMjesec(Mjeseci mjesec) {
    switch(mjesec) {
        case Januar: cout << "Januar"; break;
        case Februar: cout << "Februar"; break;
        case Mart: cout << "Mart"; break;
        case April: cout << "April"; break;
        case Maj: cout << "Maj"; break;
        case Juni: cout << "Juni"; break;
        case Juli: cout << "Juli"; break;
        case August: cout << "August"; break;
        case Septembar: cout << "Septembar"; break;
    }
}

```

```

        case Oktobar: cout << "Oktobar"; break;
        case Novembar: cout << "Novembar"; break;
        case Decembar: cout << "Decembar";
    }
}

int BrojDanaUMjesecu(Mjeseci mjesec, bool prestupna) {
    switch(mjesec) {
        case Februar: return 28 + prestupna;
        case April: case Juni: case Septembar: case Novembar: return 30;
        default: return 31;
    }
}

```

Priloženi program je prilično dugačak. Već smo jednom prilikom istakli da modularno rješenje nije i najkraće rješenje. Poređenja radi, ovdje je dat i primjer istog programa napisanog *nemodularno* i *bez uvođenja pobrojanih tipova*. Prikazani program je znatno kraći, znatno teži za razumijevanje, i znatno teži za izmjene, jer *ne prati* prirodan tok misli.

```

#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    int godina;
    for(;;) {
        cout << "Unesi godinu u opsegu (od 1997 nadalje): ";
        cin >> godina;
        if(cin && godina >= 1997) break;
        if(!cin) cin.clear();
        cout << "Neispravan unos!\n";
        cin.ignore(10000, '\n');
    }
    int ostatak = ((godina - 1996) * 365 + 1 + (godina - 1997) / 4
        - (godina - 1901) / 100 + (godina - 1601) / 400) % 7;
    for(int mjesec = 1; mjesec <= 12; mjesec++) {
        switch(mjesec) {
            case 0: cout << "Januar"; break;
            case 1: cout << "Februar"; break;
            case 2: cout << "Mart"; break;
            case 3: cout << "April"; break;
            case 4: cout << "Maj"; break;
            case 5: cout << "Juni"; break;
            case 6: cout << "Juli"; break;
            case 7: cout << "August"; break;
            case 8: cout << "Septembar"; break;
            case 9: cout << "Oktobar"; break;
            case 10: cout << "Novembar"; break;
            case 11: cout << "Decembar";
        }
        cout << "\n\nPon Uto Sri Čet Pet Sub Ned\n"
            << setw(4 * ostatak) << "";
        int broj_dana = 31 - (mjesec == 4 || mjesec == 6
            || mjesec == 9 || mjesec == 11);
        if(mjesec == 2) broj_dana = 28 + (godina % 4 == 0
            && (godina % 100 != 0 || godina % 400 == 0));
        for(int i = 1; i <= broj_dana; i++) {

```

```
    cout << setw(3) << i << " ";
    ostatak = (ostatak + 1) % 7;
    if(ostatak == 0) cout << endl;
}
cout << endl;
if(ostatak != 0) cout << endl;
}
return 0;
}
```

Naročito kratka i nejasna mogu biti tzv. *hakerska* nemodularna rješenja, koja se tipično zasnivaju na upotrebi “prljavih” trikova (u gore prikazanom programu ima i poneki hakerski trik). Naime, klasu “programera” nazvanu “hakeri” apsolutno ne zanima da li će njihov program neko razumjeti ili ne, i ne tiče ih se da li će se program moći lako održavati ili ne, jer njihovo interesovanje za program prestaje onog trenutka kada se program napiše i kada program proradi (njima je pisanje programa samo po sebi svrha). Po tome se oni razlikuju od profesionalnih programera, koji pola svog života provedu unapređujući svoje vlastite programe da udovolje zahtjevima korisnika.

21. Rekurzivne funkcije

Već smo rekli da svaka funkcija može pozivati druge funkcije. Prirodno se postavlja pitanje da li funkcija može pozivati *samu sebe*. Odgovor je potvrđan. Funkcije koje pozivaju same sebe nazivaju se *rekurzivne funkcije*, a programska tehnika u kojoj se koriste takve funkcije naziva se *rekurzija*.

Rekurzija spada u izuzetno moćne, ali i prilično komplikovane programerske tehnike. Rekurziju ćemo prvo ilustrirati na jednom sasvim jednostavnom primjeru. Poznato je da matematičku funkciju *faktorijel* možemo iskazati sljedećom formulom:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

Na osnovu ove formule sasvim lako je napisati funkciju u jeziku C++ koja računa faktorijel svog argumenta, koristeći “**for**“ petlju. Međutim, $n!$ se može definirati i na drugačiji način, preko formule

$$n! = \begin{cases} n \cdot (n-1)! & \text{za } n \neq 0 \\ 1 & \text{za } n = 0 \end{cases}$$

Ova formula definira funkciju faktorijel preko *same sebe*, i predstavlja tipičan primjer *rekurzivne definicije*. Pokažimo, na primjeru, kako teče postupak izračunavanja faktorijela broja 4 pomoću ove formule:

$$\begin{aligned} 4! &= 4 \cdot 3! = 4 \cdot (3 \cdot 2!) = 4 \cdot [3 \cdot (2 \cdot 1!)] = 4 \cdot \{3 \cdot [2 \cdot (1 \cdot 0!)]\} = \\ &= 4 \cdot \{3 \cdot [2 \cdot (1 \cdot 1)]\} = 4 \cdot [3 \cdot (2 \cdot 1)] = 4 \cdot (3 \cdot 2) = 4 \cdot 6 = 24 \end{aligned}$$

Ukoliko bismo ovu formulu neposredno preveli u C++, dobijamo rekurzivnu verziju C++ funkcije za računanje faktorijela:

```
long int Faktorijel(int n) {
    if(n == 0) return 1;
    else return n * Faktorijel(n - 1);
}
```

Upotrebom uvjetnog ternarnog operatora “`? :`” funkcija faktorijel se može zapisati još kompaktnije:

```
long int Faktorijel(int n) {
    return (n == 0) ? 1 : n * Faktorijel(n - 1);
}
```

Na sljedećoj slici prikazan je shematski tok izračunavanja ove funkcije za $n = 4$, na kojem se lijepo može pratiti šta se zapravo dešava prilikom izvršavanja ove funkcije.

```
Faktorijel(4)
= 4 * Faktorijel(3)
= 3 * Faktorijel(2)
= 2 * Faktorijel(1)
```

```
= 1 * Faktorijel(0)
```

```
= 1
```

U ovom primjeru nam rekurzija i nije bila osobito potrebna, jer se funkcija za računanje faktorijela mogla sasvim jednostavno napisati i bez rekurzije. Ipak, rekurzija je mnogo moćnija programska tehnika nego što izgleda. Sasvim je lako pokazati da se svaki problem koji se može riješiti uz upotrebu petlji može riješiti i bez petlji, uz primjenu rekurzije. Stoga čak postoje i programski jezici (npr. *Prolog*, *LISP* i svi jezici koji spadaju u kategoriju tzv. *funkcionalnih jezika*) u kojima uopće ne postoje petlje, nego je rekurzija jedini način da se u tim jezicima ostvari ponavljanje. Ovim ne treba shvatiti da je rekurzija samo drugi način da se realizira petlja. Kao što ćemo uskoro vidjeti, postoje izvjesni problemi koji posjeduju gotovo trivijalna rekurzivna rješenja, a koje je izuzetno teško riješiti bez primjene rekurzije.

Prije nego što navedemo primjere ovakvih problema, ukažimo na neka na prvi pogled veoma čudna svojstva rekurzivnih funkcija. Pretpostavimo da želimo da napišemo funkciju “*IspisiBinarno*“ koja na ekran ispisuje vrijednost svog parametra pretvorenu u binarni zapis (npr. efekat poziva funkcije “*IspisiBinarno(19)*“ trebao bi da bude ispis binarnog broja “10011”). Uz pretpostavku da koristimo metod dijeljenja zasnovan na Hornerovoj shemi (tj. na uzastopnom dijeljenju sa 2), problem nije moguće riješiti bez upotrebe nizova, jer se cifre izdvajaju redom od posljednje ka prvoj, tako da moramo memorirati sve izdvojene cifre prije nego što ih ispišemo u ispravnom poretku (od posljednje izdvojene cifre ka prvoj izdvojenoj cifri). Ova ideja dovodi nas do funkcije koja bi kompaktno zapisano mogla izgledati recimo ovako:

```
void IspisiBinarno(int n) {
    int cifre[32], brojac(0);
    while(n != 0) {
        cifre[brojac++] = n % 2;
        n /= 2;
    }
    while(--brojac >= 0) cout << cifre[brojac];
}
```

Niz “*cifre*“ dimenzioniran je na 32 elementa uz pretpostavku da traženi binarni broj neće imati više od 32 binarne cifre (što je svakako ispunjeno uz pretpostavku da tip “*int*“ zauzima ne više od 32 bita). Međutim, sljedeća *rekurzivna* realizacija *iste funkcije* zasnovana na *istoj ideji* ne samo da je nevjerojatno kratka, nego uopće ne koristi nikakav niz:

```
void IspisiBinarno(int n) {
    if(n > 1) IspisiBinarno(n / 2);
    cout << n % 2;
}
```

Na prvi pogled, nije lako ustanoviti šta se zaista dešava kada se pozove ova funkcija (što je inače tipično za većinu rekurzivnih funkcija). Stvari postaju jasnije nakon što se razmotri sljedeći dijagram, koji ilustrira tok izvršavanja ove funkcije prilikom poziva “*IspisiBinarno(19)*”, pri čemu ćemo, radi uštade u prostoru, umjesto punog imena funkcije “*IspisiBinarno*”, na dijagramu koristiti skraćeno ime “*Bin*”:

n = 19:	n = 9:	n = 4:	n = 2:	n=1:	
Bin(19)	Bin(9)	Bin(4)	Bin(2)	Bin(1)	<i>ispisi “1”</i>
<i>ispisi “1”</i>	<i>ispisi “1”</i>	<i>ispisi “1”</i>	<i>ispisi “0”</i>	<i>ispisi “0”</i>	

Moglo bi se učiniti da je prikazano rekurzivno rješenje znatno efikasnije od prethodnog nerekurzivnog rješenja, s obzirom da ne zahtijeva upotrebu niza za smještanje međurezultata. Međutim, ovo je samo iluzija. Da bismo rasvjetili misteriozni nestanak potrebe za pomoćnim nizom i shvatili šta se zaista dešava, moramo se podsjetiti da se automatske (tj. nestatičke) lokalne promjenljive svake funkcije *stvaraju* svaki put kada se funkcija pozove, i *uništavaju* po završetku funkcije. Isto vrijedi i za formalne parametre funkcije, jer su oni zapravo vrsta lokalnih promjenljivih. U ovom primjeru, poziv funkcije “*IspisiBinarno*(19)” stvara u memoriji novu promjenljivu “n” i inicijalizira je na vrijednost “19”. Nakon toga, kako je 19 veće od 1, vrši se ponovni poziv funkcije “*IspisiBinarno*” sa stvarnim parametrom “n / 2” čija je vrijednost “9”. Ovaj novi poziv funkcije “*IspisiBinarno*” stvara *novi primjerak* (instancu) lokalne promjenljive “n” koja se inicijalizira na vrijednost “9”. Međutim, prethodna lokalna promjenljiva “n” (čija je vrijednost “19”) je i dalje prisutna u memoriji! Naime, u skladu sa ranije opisanim mehanizmom stvaranja i uništavanja lokalnih promjenljivih, ona se uništava *tek kada se tijelo funkcije u potpunosti izvrši*, a izvršavanje tijela je prekinuto novim pozivom funkcije, što je dovelo do stvaranja nove promjenljive “n”. Stara promjenljiva “n” time postaje nedostupna (u korist nove promjenljive “n”) i postaće ponovo dostupna tek kada se izvrši povratak iz pozvane funkcije na mjesto poziva. Međutim, pozvana funkcija ponovo poziva samu sebe, čime se u memoriji stvara treća instance promjenljive “n” (ovaj put sa vrijednošću “4”), tako da u memoriji u ovom trenutku postoje tri instance promjenljive “n”. Ovaj proces se dalje nastavlja, tako da u razmotrenom primjeru u jednom trenutku postoji čak pet instanci promjenljive “n” (koje imaju redom vrijednosti “19”, “9”, “4”, “2” i “1”, kao što je vidljivo na prikazanoj slici). Nakon što je funkcija “*IspisiBinarno*” pet puta pozvala samu sebe, do novog poziva neće doći, jer uvjet “n > 1” više nije ispunjen, s obzirom da tekuća instance promjenljive “n” ima vrijednost “1”. Stoga se izvršava sljedeća naredba “cout << n % 2” koja ispisuje broj “1”, nakon čega dolazi do povratka iz funkcije. Ovim ujedno dolazi i do uništavanja posljednje instance promjenljive “n”. Povratak iz funkcije treba da se izvrši na naredbu iza mjesta poziva funkcije. Kako je funkcija pozvana iz sebe same, povratak će se također izvršiti u nju samu, na naredbu “cout << n % 2” koja slijedi iza mjesta poziva. Međutim, tekuća instance promjenljive “n” je sada ona koja sadrži vrijednost “2”, jer je prethodna instance uništena. Nakon obavljanja ove naredbe, dolazi do novog povratka iz funkcije u samu sebe, pri čemu se ponovo uništava tekuća instance promjenljive “n”, a nova tekuća instance postaje ona čija je vrijednost “4”. Proces se tako nastavlja na način koji je slikovito prikazan na prethodnoj shemi. Tek nakon petog završetka funkcije dolazi do uništavanja svih stvorenih instances promjenljive “n” i do povratka iz funkcije na mjesto odakle je funkcija “*IspisiBinarno*” zaista bila prvi put pozvana (npr. u glavni program).

Opisani mehanizam djeluje dosta komplikirano. Međutim, kako je njegovo razumijevanje od ključne važnosti za pravilnu primjenu rekurzije, veoma je bitno da *ne čitate dalji tekst* prije nego što budete sigurni da ste u potpunosti shvatili opisani mehanizam. Za njegovo razjašnjenje od velike pomoći može biti slikoviti prikaz sa prethodne sheme. Radi lakšeg usvajanja mehanizma rekurzivnih poziva, možete shvatiti i da svaka funkcija čije je izvršavanje prekinuto bilo pozivom neke druge funkcije bilo pozivom sebe same na neki način i dalje “nastavlja da živi” sve dok ne dođe do njenog regularnog završetka. Pri tome, ukoliko je funkcija pozvala samu sebe, stvara se njena “nova instance” (nova instance funkcije), dok “stara instance” i dalje “živi” i čeka da bude nastavljena. Tako, u nekom trenutku može da bude “živo” i po nekoliko instances iste funkcije, pri čemu je samo jedna “aktivna”, dok su ostale “uspavane” i čekaju svoj red. Primijetimo da u prethodnom primjeru do ispisa prve cifre dolazi tek nakon što se stvore sve neophodne instances funkcije “*IspisiBinarno*”, a da se zatim svaka naredna cifra ispisuje tek nakon uništavanja prethodne instance.

Koncept *instance funkcije* može djelovati dosta neprecizno i nejasno, s obzirom da za razliku od automatski lokalnih promjenljivih, funkcije nisu objekti koji se tokom rada “stvaraju” i “uništavaju” u memoriji. Stoga se koncept instance funkcije može preciznije iskazati na sljedeći način. Svaki put kada se neka funkcija pozove, u memoriji se formira izvjesna struktura podataka koja se naziva *okvir funkcije* (engl. *function frame*) i koja sadrži sve lokalne promjenljive i formalne parametre posmatrane funkcije,

kao i informaciju o tome odakle je funkcija pozvana, tj. odakle se program treba da nastavi nakon završetka funkcije. Ova informacija naziva se *povratna adresa* (engl. *return address*). Okvir funkcije uklanja se iz memorije tek nakon što se ona u potpunosti izvrši. Svaki prekid izvršavanja funkcije ostavlja njen okvir i dalje u memoriji. Prisustvo okvira funkcije u memoriji znak je da je funkcija "živa", tj. da je njen izvršavanje započeto a nije još završeno. Pri tome, izvršavanje funkcije može biti prekinuto izvršavanjem neke druge funkcije (pa čak i druge instance sebe same), ali njen okvir još postoji u memoriji i biće iskorišten onda kada funkcija ponovo bude nastavljena. Tako, postojanje više "instanci" iste funkcije u memoriji zapravo znači postojanje više primjeraka okvira te funkcije u memoriji (tj. više *instanci okvira*). Pri tome, jedna i samo jedna instanca može biti "aktivna" dok su sve ostale "na čekanju".

Nije teško uvidjeti da je formiranje novih instanci svih lokalnih promjenljivih prilikom svakog poziva funkcije od vitalnog značaja za ispravan rad rekurzivnih poziva. Naime, pretpostavimo da se prilikom rekurzivnog pozivanja funkcije "IspisiBinarno" nisu stvarale nove instance promjenljive "n", već da je svaki put korištena *ista* promjenljiva "n". Očigledno bi u tom slučaju nakon petog rekurzivnog poziva funkcije "IspisiBinarno" promjenljiva "n" poprimila vrijednost "1", bez ikakve mogućnosti povratka na njene prethodne vrijednosti (koje su zauvijek izgubljene). Stoga bi sve naredbe oblika "`cout << n % 2`" koje će se izvršiti nakon svakog povrata iz funkcije koristile tu vrijednost promjenljive "n", i ispis ne bi bio ispravan (bilo bi ispisano pet jedinica). U nekim starijim programskim jezicima (poput FORTRAN-a 77) nije moguće ispravno definirati rekurzivne funkcije upravo zbog činjenice da se u njima ne stvaraju nove instance lokalnih promjenljivih pri rekurzivnim pozivima (npr. u FORTRAN-u 77 sve lokalne promjenljive ponašaju se poput statičkih promjenljivih u jeziku C++).

Sada nije teško odgovoriti na pitanje kako rekurzivna verzija funkcije "IspisiBinarno" ne zahtijeva upotrebu pomoćnog niza za pamćenje međurezultata. Suština je u tome da se svi međurezultati koji su potrebni za ispravno funkcioniranje funkcije čuvaju u *različitim instancama* formalnog parametra "n" funkcije, odnosno u *različitim okvirima* iste funkcije (primijetimo da izračunavanje i ispisivanje cifara zapravo započinje tek kada se formiraju svi okviri koji su potrebni, a samim tim i svi međurezultati). Međutim, dok se kod nerekurzivne verzije funkcije "IspisiBinarno" o pamćenju potrebnih međurezultata morao brinuti sam programer, kod rekurzivne verzije se pamćenje međurezultata ostvaruje *automatski*, bez znanja programera, kroz mehanizam prenosa parametara po vrijednosti. Doduše, uskoro ćemo vidjeti da se prilikom upotrebe rekurzije često u memoriji pamti i što treba i što ne treba, tako da rekurzija prilično rasipno troši memorijске resurse.

Sljedeći primjer također ilustrira da se primjenom rekurzije često mogu bez upotrebe nizova riješiti problemi koji bi inače zahtijevali njihovu upotrebu. Neka je, na primjer, potrebno sa tastature unijeti *n* brojeva a_1, a_2, \dots, a_n (pri čemu se *n* također prethodno unosi sa tastature), a zatim ispisati i izračunati vrijednost verižnog razlomka

$$a_1 + \cfrac{1}{a_2 + \cfrac{1}{\dots + \cfrac{1}{a_n}}}$$

Kako se računanje verižnog razlomka obavlja od posljednjeg unesenog brojeve ka prvom, izgleda da je jedino rješenje učitati sve brojeve u niz, a zatim "razmotati" razlomak koristeći vrijednosti smještene u niz, počev od posljednjeg elementa ka prvom. Međutim, sljedeće rješenje u kojem se koristi rekurzivna funkcija "VerizniRazlomak" uopće ne koristi niz. Umjesto toga, uneseni brojevi se pamte u *različitim instancama* lokalne promjenljive "a" (koje su sadržane u različitim instancama okvira funkcije):

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

```

double VerizniRazlomak(int n) {
    int a;
    cin >> a;
    if(n == 1) return a;
    else return a + 1 / VerizniRazlomak(n - 1);
}

int main() {
    int n;
    cout << "Koliko zelite unijeti brojeva? ";
    cin >> n;
    cout << "Unesite ih...\n";
    double rezultat = VerizniRazlomak(n);
    cout << "Vrijednost verižnog razlomka je " << rezultat;
    return 0;
}

```

Primjetimo da se u ovom primjeru stvaraju i višestruke instance formalnog parametra “n“ prilikom svakog rekurzivnog poziva, mada nam njihovo stvaranje nije potrebno, s obzirom da se vrijednost “n“ nigdje ne koristi nakon poziva funkcije. Nažalost, stvaranje ovih instanci se ne može izbjegći, čime se nepotrebno troše memorijski resursi (doduše, ovaj problem je *u načelu* moguće izbjegći ukoliko umjesto formalnog parametra “n“ koristimo *globalnu promjenljivu*, ali je takvo rješenje veoma ružno sa aspekta modularnosti). Kada bi se formalni parametri mogli deklarirati kao statički, deklaracija formalnog parametra “n“ kao statičkog riješila bi problem, ali nažalost, ne mogu (barem ne u jeziku C++). Ovu osobinu treba shvatiti kao “kolateralnu štetu” rekurzivnih rješenja. Kasnije će usljediti nešto detaljnija diskusija efikasnosti rekurzivnih rješenja.

Često je prilikom realizacije rekurzivnih rješenja potrebno ostvariti komunikaciju između *različitih instanci iste funkcije*. Kako svaka od instanci ima pristup samo onim instancama lokalnih promjenljivih koje je sama stvorila, jedine mogućnosti za ostvarivanje ove komunikacije su putem prenosa parametara u novostvorenu instancu, ili posredstvom globalnih ili statičkih promjenljivih (pri čemu komunikacija preko globalnih promjenljivih nije dobra sa aspekta modularnosti, dok je komunikacija putem statičkih promjenljivih vezana za neke teškoće vezane za njihovu inicijalizaciju, koje ćemo kasnije razmotriti).

Komunikaciju između različitih instanci iste funkcije ilustriraćemo na konkretnom primjeru. Pretpostavimo, da želimo modificirati prethodno rekurzivno rješenje problema računanja verižnog razlomka tako da nam se prilikom unosa brojeva ispisuje njihov redni broj, npr. “Unesite 1. broj:“, “Unesite 2. broj:“ itd. U klasičnom nerekurzivnom rješenju za tu svrhu bismo prosto iskoristili brojač petlje kojom bismo unosili brojeve u niz. Kako u predloženom rekurzivnom rješenju nema nikakve petlje, pa samim tim niti “brojač petlje”, moraćemo sami uvesti neku promjenljivu (npr. “*brojac*”) kojom ćemo brojati brojeve koje unosimo. Međutim, ona ne smije biti obična lokalna promjenljiva, jer će stvaranje svake njene nove instance dovesti do toga da njena vrijednost bude nedefinirana. Naime, ne smijemo zaboraviti da sve neinicijalizirane lokalne promjenljive nakon stvaranja imaju nedefiniranu vrijednost sve dok im se eksplicitno ne dodijeli vrijednost. Jedna mogućnost je da promjenljiva “*brojac*“ bude globalna, ali to je veoma loše rješenje, jer se time omogućava pristup toj promjenljivoj iz čitavog programa, mada je njena uloga ograničena samo na tijelo funkcije “*VerizniRazlomak*“. Relativno dobro rješenje je učiniti promjenljivu “*brojac*“ *formalnim parametrom*, koji se prilikom svakog poziva funkcije inicijalizira na vrijednost za 1 veću u odnosu na tekuću vrijednost. To bi moglo izgledati recimo ovako:

```

double VerizniRazlomak(int n, int brojac = 1) {
    int a;
    cout << "Unesite " << brojac << ". broj: ";
    cin >> a;
    if(n == 1) return a;
}

```

```

    else return a + 1 / VerizniRazlomak(n - 1, brojac + 1);
}

```

U ovom primjeru, formalni parametar "brojac" ima podrazumijevanu vrijednost "1", jer brojač zaista i kreće od jedinice. Ovim je omogućeno da se funkcija "VerizniRazlomak" i dalje poziva iz glavnog programa na isti način, zadavanjem samo parametra "n". S druge strane, unutar funkcije ona poziva samu sebe svaki put posljedujući kao stvarni parametar vrijednost formalnog parametra "brojac" uvećanu za 1, čime svaka nova instanca promjenljive "brojac" postaje za jedan veća od prethodne. Primijetimo da je gomilanje instanci ove promjenljive također "kolateralna šteta".

Razmotrimo da li je moguće izbjegići gomilanje instanci promjenljive "brojac". Na prvi pogled, rješenje je posve jednostavno. Umjesto prenošenja brojača kao parametra, moguće je brojač deklarirati kao *statičku promjenljivu*, inicijaliziraju na vrijednost 1. Kako se statičke promjenljive stvaraju i inicijaliziraju samo kada se pri izvršavanju programa prvi put nađe na njihovu deklaraciju, to se pri rekurzivnim pozivima ne stvaraju njihove nove instance. Modularnost je pri tome sačuvana, s obzirom da se statičkoj promjenljivoj ne može pristupiti izvan funkcije. Dalje, kako se statičke promjenljive ne uništavaju po završetku funkcije i kako se njihova vrijednost zadržava do sljedećeg poziva funkcije, svaka nova instanca funkcije zateći će onaku vrijednost statičke promjenljive kakva je bila tokom izvršavanja prethodne instance. To nas dovodi do sljedeće realizacije funkcije "VerizniRazlomak":

```

double VerizniRazlomak(int n) {
    static int brojac(1);
    int a;
    cout << "Unesite " << brojac++ << ". broj: ";
    cin >> a;
    if(n == 1) return a;
    else return a + 1 / VerizniRazlomak(n - 1);
}

```

Nažalost, prikazana realizacija posjeduje jedan ozbiljan nedostatak. Sve će biti u redu ukoliko se napisana funkcija pozove samo jedanput iz glavnog programa (ili bilo odakle osim iz same sebe). Međutim, ukoliko ovu funkciju ponovo pozovemo (npr. iz glavnog programa) nakon što je već jedanput izvršena, brojanje unesenih brojeva neće započeti ponovo od jedinice, nego od vrijednosti na kojoj je posljednje odbrojavanje završeno. Naime, kako se statičke promjenljive inicijaliziraju samo prvi put, statička promjenljiva "brojac" nikada više neće biti inicijalizirana na jedinicu nakon što je njena inicijalizacija jedanput izvršena!

Opisani problem pokazuje da je čuvanje informacija o stanju funkcije (npr. o broju izvršenih poziva) u nekoj statičkoj promjenljivoj unutar funkcije vezano za poteškoće ukoliko se javlja potreba da naknadno trebamo iz nekog razloga izvršiti reinicijalizaciju stanja. U objektno orijentiranom programiranju, sa kojim ćemo se upoznati kasnije, ovaj problem se rješava uvođenjem tzv. *funckijskih objekata* umjesto običnih funkcija. Međutim, držimo li se klasičnog pristupa, ovaj problem u općem slučaju nije tako lako rješiv. Potrebno je unutar same funkcije naći pogodan trenutak kada informacija o stanju funkcije više nije potrebna za dalje izvršavanje funkcije, i tada izvršiti reinicijalizaciju (putem eksplisitne dodjele), sa ciljem dovođenja funkcije u ispravno stanje za potrebe kasnijih poziva. Ovo je obično mnogo lakše reći nego izvesti. U konkretnom primjeru funkcije "VerizniRazlomak" to je prilično lako. Naime, lako se može vidjeti da nakon što formalni parametar "n" dostigne vrijednost 1, vrijednost promjenljive "brojac" više neće biti korištena, tako da tada možemo izvršiti njenu reinicijalizaciju na početnu vrijednost, kao u sljedećoj realizaciji:

```

double VerizniRazlomak(int n) {
    static int brojac(1);
}

```

```

int a;
cout << "Unesite " << brojac++ << ". broj: ";
cin >> a;
if(n == 1) {
    brojac = 1;
    return a;
}
else return a + 1 / VerizniRazlomak(n - 1);
}

```

U ovom primjeru smo opisani problem riješili relativno lako. U općem slučaju, rješavanje ovog problema nije jednostavno. Naime, često je za uspješno prepoznavanje trenutka u kojem se stanje smije vratiti na početnu vrijednost potrebno uvesti neke pomoćne (također statičke) promjenljive. Nešto kasnije u ovom poglavlju, pri razmatranju rješavanja problema tzv. *Hanojskih kula*, ovaj problem će biti riješen na prilično općenit način, koji se može primijeniti u brojnim situacijama.

U svim do sada izloženim primjerima nismo imali neke prevelike koristi od rekurzivnih rješenja. Naime, mada prikazana rekurzivna rješenja djeluju jako elegantno, isti problemi su se mogli uz neznatno veći napor riješiti i bez pomoći rekurzije. Naime, svako rekurzivno rješenje u kojem instanca funkcije rekurzivno poziva samu sebe *najviše jedanput* za vrijeme svog života veoma lako se može prepraviti u nerekurzivno rješenje. Pri tome će nam eventualno zatrebati nizovi za pamćenje nekih međurezultata koji bi se inače u rekurzivnom rješenju pamtili automatski u instancama okvira pozivanih funkcija. Također, lako se uočava da su u svim razmotrenim primjerima nerekurzivna rješenja bila *efikasnija* od rekurzivnih rješenja. Naime, u primjeru funkcije “Faktorije1”, svaki rekurzivni poziv stvara novu instancu okvira funkcije čime se bespotrebno troše memorijski resursi, s obzirom da se u odgovarajućoj nerekurzivnoj verziji iste funkcije svi međurezultati mogu pamtitи u jednoj jedinoj promjenljivoj (koja pamti produkt do tada obrađenih brojeva iz raspona od 1 do n). U slučaju funkcije “IspisiBinarno”, nerekurzivna verzija za pamćenje međurezultata koristi niz, dok rekurzivna verzija koristi instance okvira. Na prvi pogled izgleda da je u ovom slučaju utrošak memorije isti. Međutim, okviri funkcije pored vrijednosti lokalnih promjenljivih i formalnih parametara pamte i *povratne adresu*, tako da se ispostavlja da formirani okviri funkcija zauzimaju više prostora u memoriji nego što bi zauzeo odgovarajući niz. Slična je stvar i sa rekurzivnim primjerom funkcije “VerizniRazlomak”, gdje se u okvirima funkcija pored povratnih adresa bespotrebno pamte i vrijednosti formalnih parametara “ n ” i eventualno parametra “ $brojac$ ” (ako koristimo varijantu u kojoj se brojač prenosi kao parametar), tako da je zauzeće memorije ponovo veće nego da smo koristili niz. Sa aspekta *brzine izvršavanja* rekurzivna rješenja su također manje efikasna. Naime, prilikom svakog rekurzivnog poziva troši se izvjesno vrijeme na poziv funkcije, na povratak iz funkcije, kao i na stvaranje i uništavanje okvira funkcije. Mada navedene operacije računar generalno izvršava prilično brzo, one ipak usporavaju izvršavanje programa, pogotovo u slučaju velikog broja rekurzivnih poziva.

Iz navedene diskusije moglo bi se pogrešno zaključiti da od rekurzije uglavnom imamo mnogo više štete nego koristi. Međutim, postoje izvjesni problemi koje je veoma teško riješiti bez primjene rekurzije, a za koje postoje gotovo očigledna rekurzivna rješenja. Takav slučaj se najčešće javlja kada se rješavanje nekog problema može svesti na rješavanje više problema istog tipa, samo manje veličine. Pošto smanjivanje veličine problema ne može ići unedogled, jednom ćemo naići na trivijalan problem, koji znamo riješiti. Tako se rješavanje problema svodi na *rekurzivno rješavanje* manjih potproblema, sve dok se ne dostignu trivijalni slučajevi. Ovaj metod rješavanja problema poznat je pod nazivom *metod redukcije*. U takvim rješenjima instanca funkcije najčešće rekurzivno poziva samu sebe *više od jedanput*, pa čak i *unaprijed neodređen broj puta* (npr. unutar neke petlje). Ovakva rješenja je prilično teško realizirati nerekurzivno. Snagu ovog pristupa ilustriraćemo na primjeru rješavanja igre poznate pod imenom *Hanojske kule*. Na jednom od tri štapa (recimo, prvom) nalazi se niz diskova različite veličine, poredanih po veličini od najmanjeg ka najvećem. Potrebno je sve diskove prebaciti sa prvog na drugi štap,

eventualno koristeći treći štap kao pomoćni, ali tako da se niti u jednom potezu ne smije veći disk staviti na manji. Na sljedećoj slici prikazana je situacija sa tri diska:

Za slučaj tri diska nije teško pronaći ispravan redoslijed poteza da se diskovi prebacuju sa štapa 1 na štap 2 poštujući pravila igre (najkraći redoslijed poteza je $1 \rightarrow 2$, $1 \rightarrow 3$, $2 \rightarrow 3$, $1 \rightarrow 2$, $3 \rightarrow 1$, $3 \rightarrow 2$ i $1 \rightarrow 2$). Međutim, situacija se znatno komplikira sa porastom broja diskova. Pokušajmo napisati program za računar koji bi pronalazio ispravan redoslijed poteza za proizvoljan broj diskova. Na prvi pogled, ne vidimo nikakav očigledan postupak kako bi se ovaj problem mogao riješiti. Ipak, ako malo razmislimo, lako ćemo doći do zaključka da ne postoji nikakav način da prebacimo N diskova sa štapa X na štap Y osim da prvo na neki način prebacimo $N-1$ diskova sa štapa X na štap Z (koristeći štap Y kao pomoćni) zatim da prebacimo jedan preostali (najveći) disk sa štapa X na štap Y , i na kraju da nekako prebacimo $N-1$ diskova sa štapa Z na štap Y (koristeći štap X kao pomoćni). Ovim problem nije riješen, mada je nakon ovog razmatranja problem prebacivanja N diskova sveden na dva problema prebacivanja $N-1$ diskova. Dakle, broj diskova je umanjen za 1. Sličnim rezonovanjem problemi prebacivanja $N-1$ diskova svode se na probleme prebacivanja $N-2$ diskova, itd. Očigledno problemi prestaju kada broj diskova dostigne nulu, jer tada nemamo šta prebacivati. Ovo rezonovanje dovodi nas do sljedećeg algoritma, koji je rekursivan jer se poziva na samog sebe:

- *Prebacivanje N diskova sa štapa X na štap Y koristeći štap Z kao pomoćni:*
 - *Ako N nije nula:*
 - *Prebaci $N-1$ diskova sa štapa X na štap Z koristeći štap Y kao pomoćni;*
 - *Prebaci jedan disk sa štapa X na štap Y ;*
 - *Prebaci $N-1$ diskova sa štapa Z na štap Y koristeći štap X kao pomoćni.*

Ovaj algoritam lako je pretočiti u konkretni C++ program, koji ispisuje redoslijed poteza neophodan za rješavanje problema Hanojskih kula (formalnim parametrima “x”, “y” i “z” date su podrazumijevane vrijednosti 1, 2 i 3, tako da se funkcija “Hanoi” može pozivati samo uz navođenje broja diskova):

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

void Hanoi(int n, int x = 1, int y = 2, int z = 3) {
    if(n != 0) {
        Hanoi(n - 1, x, z, y);
        cout << "Prebaci disk sa štapa " << x << "na štap " << y << endl;
        Hanoi(n - 1, z, y, x);
    }
}

int main() {
    int broj_diskova;
    cout << "Koliko ima diskova? ";
    cin >> broj_diskova;
    Hanoi(broj_diskova);
    return 0;
}
```

Vidimo da je rezultujući program veoma kratak. To ipak ne znači da je on jednostavan. Naprotiv, izuzetno je teško pratiti njegov tok, što je tipično za sve rekurzivne programe. Mnogo je lakše razumjeti kako ovaj program radi *globalno* (tj. kao *cjelina*) nego odgonetnuti kojim zapravo redoslijedom računar izvršava instrukcije ovog programa. Stoga, rekurzivno rješavanje problema leži negdje na granici proceduralnog i neproceduralnog (deskriptivnog) programiranja, jer se rekurzivna rješenja nekada više svode na opis problema iskazan u formi svođenja na jednostavnije probleme, nego na konkretni postupak rješavanja. Pokušamo li, recimo, da prateći ovaj program pronađemo neophodan redoslijed poteza za rješavanje problema sa $N=4$ kule, veoma ćemo se brzo izgubiti. Jedini način da izademo na kraj sa analizom ovog programa je da se poslužimo shematskim prikazom, poput ovog na sljedećoj slici:

Hanoi(4, 1, 2, 3)

Hanoi(3, 1, 3, 2) prebac sa 1 na 2 Hanoi(3, 3, 2, 1)	Hanoi(2, 1, 2, 3) prebac sa 1 na 3 Hanoi(2, 2, 3, 1)	Hanoi(1, 1, 3, 2) prebac sa 1 na 2 Hanoi(1, 3, 2, 1)	prebac sa 1 na 3 prebac sa 3 na 2
	Hanoi(2, 3, 1, 2) prebac sa 3 na 2 Hanoi(2, 1, 2, 3)	Hanoi(1, 2, 1, 3) prebac sa 2 na 3 Hanoi(1, 1, 3, 2)	prebac sa 2 na 1 prebac sa 1 na 3
		Hanoi(1, 3, 2, 1) prebac sa 3 na 1 Hanoi(1, 2, 1, 3)	prebac sa 3 na 2 prebac sa 2 na 1
		Hanoi(1, 1, 3, 2) prebac sa 1 na 2 Hanoi(1, 3, 2, 1)	prebac sa 1 na 3 prebac sa 3 na 2

Sa ove slike možemo ustanoviti da ispravan redoslijed poteza za $N=4$ glasi $1 \rightarrow 3$, $1 \rightarrow 2$, $3 \rightarrow 2$, $1 \rightarrow 3$, $2 \rightarrow 1$, $2 \rightarrow 3$, $1 \rightarrow 3$, $1 \rightarrow 2$, $3 \rightarrow 2$, $3 \rightarrow 1$, $2 \rightarrow 1$, $3 \rightarrow 2$, $1 \rightarrow 3$, $1 \rightarrow 2$ i $3 \rightarrow 2$. Bez crtanja dijagrama poput prethodnog i praćenja vrijednosti koje imaju formalni parametri “ n ”, “ x ”, “ y ” i “ z ”, tok programa je praktički neuhvatljiv, bez obzira što sam program izgleda veoma jednostavno. U ovom primjeru također možemo vidjeti da je njegovo ispravno funkcioniranje zasnovano na činjenici da svaka instanca funkcije “Hanoi” koristi vlastite instance svojih lokalnih parametara, pri čemu pri svakom rekurzivnom pozivu dolazi do presipanja vrijednosti tekućih instanci zadanih u vidu stvarnih parametara u novoformirane instance. To i jeste glavni razlog zbog čega je tačan tok programa tako teško pratiti bez obzira što sama funkcija “Hanoi” djeluje gotovo trivijalno i očigledno.

U prikazanom primjeru funkcije “Hanoi”, svaka instanca njenog okvira čuva pored povratne adrese vrijednosti četiri formalna parametra “ n ”, “ x ”, “ y ” i “ z ”. Ukoliko primijetimo da je zbir vrijednosti “ x ”, “ y ” i “ z ” uvijek 6, možemo izbjegći potrebu za formalnim parametrom “ z ” (odnosno rednim brojem pomoćnog štapa) tako što ćemo njegovu vrijednost računati po relaciji “ $6 - x - y$ ”. Ovim dolazimo do sljedeće verzije funkcije “Hanoi” koja troši manje memorijskih resursa:

```
void Hanoi(int n, int x = 1, int y = 2) {
    if(n != 0) {
        Hanoi(n - 1, x, 6 - x - y);
        cout << "Prebac disk sa štapa " << x << "na štap " << y << endl;
        Hanoi(n - 1, 6 - x - y, y);
    }
}
```

Bitno je naglasiti da je uvjet “ $n \neq 0$ ” koji se nalazi unutar funkcije “Hanoi” od izuzetne važnosti za

rad programa, mada na prvi pogled ne djeluje naročito bitan. Međutim, ukoliko bismo zaboravili ovaj uvjet, računar bi problem sa 0 diskova pokušao svesti na problem sa -1 diskom (tj. na besmislicu), problem sa -1 diskom na problem sa -2 diska, i tako unedogled. Slična stvar bi se dogodila i u svim do sada razmotrenim primjerima rekurzivnih funkcija. Na primjer, rekurzivna funkcija "Faktorijel" bi sama sebe rekurzivno pozivala unedogled ukoliko bismo zaboravili definirati da je $0! = 1$. Naime, tada bismo dobili beskonačan lanac poput

$$4! = 3 \cdot 2! \quad 3! = 3 \cdot 2! \quad 2! = 2 \cdot 1! \quad 1! = 1 \cdot 0! \quad 0! = 0 \cdot (-1)! \quad (-1)! = (-1) \cdot (-2)! \quad \dots$$

Izostavljanje nekog od ključnih uvjeta koji garantiraju da će se rekurzija završiti u konačnom vremenu (tj. da se neće protezati unedogled) predstavlja ubjedljivo najčešću grešku koju neiskusni programeri čine prilikom programiranja rekurzivnih algoritama. Treba uočiti da lanac rekurzivnih poziva koji nikada ne prestaju dovodi do stalnog stvaranja novih instanci okvira funkcija. Okviri funkcija se obično stvaraju u dijelu memorije nazvanom *mašinski stek*, čiji je kapacitet ograničen (kao i, uostalom, kapacitet ma kakve memorije). Kada se mašinski stek prepuni, nove instance se ne mogu stvarati, te dolazi do neizbjježnog kraha programa, eventualno uz neku poruku operativnog sistema poput "Stack overflow" itd. Dakle, greške ovog tipa uvijek su fatalne za izvršavanje programa. Po tome se rekurzivna rješenja bitno razlikuju od nerekurzivnih rješenja, kod kojih izostavljanje tretmana nekog od specijalnih slučajeva uglavnom dovodi do neispravnog rada programa samo za te specijalne slučajeve, a ne i za druge slučajeve. Jasno je i zbog čega je tako: kod rekurzivnog rješavanja problem se neprestano reducira dok se ne svede na neki od specijalnih slučajeva. Stoga je izostavljanje tretmana specijalnih slučajeva pogubno za sve ostale slučajeve!

Razmotrimo sada kako bismo mogli modificirati funkciju "Hanoi" tako da se prije ispisa svakog poteza prikaže i redni broj odgovarajućeg poteza (npr. "Potez br. 5: Prebaci disk sa štapa 3 na štap 1"). Ovaj problem će jasno demonstrirati poteškoće koji mogu nastati pri komunikaciji između različitih instanci iste rekurzivne funkcije. Naime, problem je trivijalan ukoliko dopustimo upotrebu globalne promjenljive koja će brojati poteze. Međutim, želimo li izbjegći globalne promjenljive, suočićemo se sa znatnom poteškoćama. Jedna mogućnost je prenošenje brojača poteza putem parametara, kao u primjeru računanja verižnog razlomka. Međutim, nije teško vidjeti da bi se u ovom slučaju brojač poteza morao prenositi *po referenci*, jer bi nakon povratka iz prvog rekurzivnog poziva funkcije bilo potrebno ispisati vrijednost brojača kakva je bila po završetku ovog rekurzivnog poziva, što je moguće jedino korištenjem prenosa po referenci. Naravno, prenos po referenci sam po sebi načelno nije loša stvar. Ipak, pošto parametri koji se prenose po referenci ne mogu imati podrazumijevane vrijednosti, slijedi da bismo, upotrebom takvog rješenja, prilikom pozivanja funkcije "Hanoi" uvijek morali eksplicitno zadavati početnu vrijednost brojača, koja bi, da stvar bude još gora, morala biti zadana kao *promjenljiva* (s obzirom da se referenca ne može vezati za konstantu). Doduše, postoji jedan relativno jednostavan način rješavanja ove poteškoće, ukoliko modificiramo funkciju "Hanoi" tako da postane funkcija koja vraća *vrijednost*. Čitateljima (a bogami i čitateljkama) se savjetuje da sami razmисle o ovoj mogućnosti (ne treba zaboraviti da se povratna vrijednost funkcije smije ignorirati ukoliko nije potrebna).

Razmotrimo kako bismo mogli izbjegći opisane poteškoće korištenjem statičkih promjenljivih. Ukoliko se funkcija "Hanoi" poziva samo jednom iz glavnog programa, problem je trivijalan. Naime, dovoljno je deklarirati brojač poteza kao statičku promjenljivu inicijaliziranu na jedinicu. Međutim, ukoliko je potrebno ovu funkciju pozivati više puta, suočićemo se sa istim problemom kao pri brojanju unesenih brojeva u primjeru računanja verižnog razlomka. Osnovni je problem ponovo *kada i kako (re)inicijalizirati brojač*. Za razliku od primjera sa verižnim razlomkom, u ovom primjeru nema niti jednog mjesta u funkciji nakon kojeg sigurno možemo tvrditi da vrijednost brojača više neće biti potrebna do kraja izvršavanja svih instanci funkcije. Jasno je da brojač treba inicijalizirati na početnu vrijednost u svim slučajevima *osim kada je funkcija pozvana iz same sebe*. Međutim, *kako možemo znati* da li je funkcija pozvana iz sebe same? Možda da prenosimo informaciju o tome kao parametar logičkog tipa? U

nedostatku bolje ideje, to i nije tako loše rješenje:

```
void Hanoi(int n, int x = 1, int y = 2, bool iz_sebe_same = false) {
    static int brojac;
    if(!iz_sebe_same) brojac = 1;
    if(n != 0) {
        Hanoi(n - 1, x, 6 - x - y, true);
        cout << "Potez br. " << brojac++ << ": Prebaci disk sa štapa "
            << x << "na štap " << y << endl;
        Hanoi(n - 1, 6 - x - y, y, true);
    }
}
```

Ipak, moguća su i mnogo bolja rješenja. Možemo li kako automatski odrediti da li je funkcija pozvana iz same sebe? Ideja je da deklariramo statičku promjenljivu nazvanu npr “dubina” koju ćemo inicijalizirati na 0. Ovu promjenljivu ćemo prije svakog rekurzivnog poziva povećavati za 1, a nakon svakog rekurzivnog poziva smanjivati za 1. Tako, vrijednost ove promjenljive zapravo označava koliko se “duboko” funkcija izvršava u hijerarhiji rekurzivnih poziva. Vrijednost 0 označava da funkcija nije pozvana iz same sebe, vrijednost 1 označava da je funkcija pozvana iz druge instance iste funkcije koja pri tom nije pozvana iz sebe same, vrijednost 2 označava da je funkcija pozvana iz druge instance iste funkcije koja je također pozvana iz neke instance same sebe (koja pri tom nije pozvana iz same sebe), itd. Očigledno je brojač poteza potrebno postaviti na početnu vrijednost 1 ako i samo ako je vrijednost promjenljive “dubina” jednaka nuli, što nas dovodi do sljedećeg rješenja:

```
void Hanoi(int n, int x = 1, int y = 2) {
    static int brojac, dubina(0);
    if(dubina == 0) brojac = 1;
    if(n != 0) {
        dubina++;
        Hanoi(n - 1, x, 6 - x - y);
        dubina--;
        cout << "Potez br. " << brojac++ << ": Prebaci disk sa štapa "
            << x << "na štap " << y << endl;
        dubina++;
        Hanoi(n - 1, 6 - x - y, y);
        dubina--;
    }
}
```

Ako primijetimo da se vrijednost promjenljive “dubina” ne koristi nizašta između dva rekurzivna poziva, možemo uštediti po jedno nepotrebno smanjenje i ponovno uvećavanje ove promjenljive, tako da dolazimo do sljedećeg (konačnog) rješenja:

```
void Hanoi(int n, int x = 1, int y = 2) {
    static int brojac, dubina(0);
    if(dubina == 0) brojac = 1;
    if(n != 0) {
        dubina++;
        Hanoi(n - 1, x, 6 - x - y);
        cout << "Potez br. " << brojac++ << ": Prebaci disk sa štapa "
            << x << "na štap " << y << endl;
        Hanoi(n - 1, 6 - x - y, y);
        dubina--;
    }
}
```

Opisano rješenje problema (re)inicijalizacije statičkih promjenljivih u rekurzivnim funkcijama je prilično univerzalno i primjenljivo je na veliku klasu problema. Izvjesni problemi jedino mogu nastati u rekurzivnim funkcijama koje vraćaju vrijednost, u slučaju da se rekurzivni poziv izvrši direktno unutar parametra naredbe “`return`”. Naime, tada nije moguće izvršiti uvećanje promjenljive “dubina”, jer će naredbom “`return`” izvršavanje funkcije biti prekinuto. Ovaj problem nije teško riješiti prostim prebacivanjem rekurzivnog poziva izvan “`return`” naredbe, eventualno uz upotrebu pomoćne promjenljive za prihvatanje rezultata rekurzivnog poziva.

Problem Hanojskih kula se obično u literaturi navodi kao tipičan primjer problema za koji postoji veoma jednostavno rekurzivno rješenje, a za koje nije nimalo lako naći nerekurzivno rješenje. Prirodno je postaviti pitanje da li nerekurzivno rješenje problema Hanojskih kula uopće postoji. Odgovor je potvrđan. Štaviše, u teoriji algoritama se dokazuje da za svako rekurzivno rješenje postoji i *iterativno rješenje* kojim se postiže isti efekat (pod iterativnim rješenjem se smatra nerekurzivno rješenje zasnovano na primjeni petlji). Čak postoje i sasvim formalizirani, mada ne i posve jednostavnii postupci prevodenja rekurzivnih u nerekurzivna rješenja. Pri tome su dobijena nerekurzivna rješenja obično osjetno komplikovani od polaznih rekurzivnih rješenja. Pored toga, tako formalno dobijena nerekurzivna rješenja obično nisu bitno efikasnija od polaznih rekurzivnih rješenja, jer za dobijanje bitno efikasnijih nerekurzivnih rješenja (u slučaju da takva uopće postoje) treba koristiti posve drugu (nerekurzivnu) *filozofiju razmišljanja*, a ne samo rekurzivno razmišljanje iskazati nerekurzivno. Također, za neke probleme se može dokazati da bitno efikasnija rješenja od rekurzivnog rješenja i ne postoje (problem Hanojskih kula je upravo takav). Međutim, interesantno je, i nije mnogo poznato, da problem Hanojskih kula ima prilično jednostavno iterativno (nerekurzivno) rješenje koje je veoma lako iskazati u prirodnom govornom jeziku:

- *Prebacivanje N diskova sa štapa X na štap Y koristeći štap Z kao pomoćni:*
 - *Sve dok svi diskovi ne budu na štalu Y:*
 - *Ako je N neparan:*
 - *Prebaci najmanji disk u smjeru X → Y → Z → X;*
 - *U suprotnom:*
 - *Prebaci najmanji disk u smjeru X → Z → Y → X;*
 - *Povuci jedini trenutno mogući potez u kojem se ne koristi najmanji disk.*

Ovaj algoritam je veoma jednostavno pratiti ručno, i čovjeku nije nikakav problem provesti ovaj algoritam za proizvoljan broj N , bez obzira što nam je rekurzivni algoritam zadao velikih problema pri praćenju već za $N=4$ (stoga, ukoliko se god budete htjeli pohvaliti pred nekim da znate rješiti problem hanojskih kula *ručno*, koristite upravo ovaj algoritam). S druge strane, ovaj algoritam je daleko od očiglednog, i izuzetno je teško *shvatiti zašto on radi*, iako se na primjerima lako možemo uvjeriti da radi (opravno, matematičkom indukcijom se može dokazati da ovaj algoritam povlači *potpuno identične poteze* kao i već razmotreni rekurzivni algoritam). Pored toga, ovaj algoritam nije posve jednostavno pretvoriti u program za računar. Naročito je problematičan korak algoritma koji glasi “Povuci jedini trenutno mogući potez u kojem se ne koristi najmanji disk”. Naime, čovjeku koji ima ispred sebe sliku diskova veoma je lako uočiti koji je to “jedini mogući potez”. Međutim, da bismo ovaj korak implementirali u programu, moramo voditi evidenciju (npr. u nekom nizu) o poziciji svih diskova da bismo uočili koji su potezi u kojem trenutku dozvoljeni a koji nisu (u rekurzivnom rješenju, pozicije diskova nakon svakog poteza implicitno su memorirane u instancama formalnih parametara). Uglavnom, prevođenje ovog algoritma u konkretni računarski program veoma je korisna vježba, koju bi trebao da pokuša izvesti svaki čitatelj ili čitateljka koji imaju namjeru da se iole ozbiljnije bavi programiranjem. U svakom slučaju, dobijeno rješenje biće osjetno komplikovanije od prikazanog rekurzivnog rješenja.

Izlaganje o problemu Hanojskih kula završićemo i malim komentarom o tome odakle ovaj problem potiče. Problem je 1894. godine postavio matematičar *Édouard Lucas*, inspirisan jednom drevnom

hinduističko-budističkom legendom, Prema ovoj legendi, Bog je, neposredno nakon stvaranja svijeta, zadao grupi svećenika (u nekom manastiru u Tibetu) da rješavaju upravo ovaj problem sa 64 zlatna diska, i rekao je da će smak svijeta nastupiti onog trenutka kada završe sa rješavanjem. Izračunajmo koliko nam je vremena ostalo do smaka svijeta po ovoj legendi. Matematičkom indukcijom se lako može dokazati da minimalni broj poteza neophodan za slaganje N diskova iznosi $2^N - 1$, što za $N=64$ iznosi $2^{64} - 1 = 18446744073709551615$ poteza. Ukoliko bi svećenici u svakoj sekundi obavljadi po jedan potez, i pri tome ne bi ni jednom pogriješili, za obavljanje ovog posla bilo bi im potrebno 584942417355 godina, što je oko 40 puta više od trenutno procijenjene starosti svemira. Što znači da su, u najboljem slučaju, prema ovoj legendi svećenici mogli do sada završiti najviše 40-ti dio ukupnog posla. Dakle, smak svijeta neće nastupiti tako brzo, bar ne što se tiče ove legende...

Slijedi još jedan primjer problema koji je na prvi pogled veoma teško riješiti. Ipak, problem ima relativno jednostavno rekurzivno rješenje, dok je nerekurzivno rješenje znatno složenije. Problem se sastoji u ispisivanju svih mogućih rastava prirodnog broja N na $K \leq N$ sabiraka (ovaj problem postavljen je 2001. godine na bosanskohercegovačkom državnom takmičenju iz programiranja za učenike srednjih škola). Na primjer, za $N=6$ i $K=3$ treba ispisati rastave “1+1+4”, “1+2+3”, “1+3+2”, “1+4+1”, “2+1+3”, “2+2+2”, “2+3+1”, “3+1+2”, “3+2+1” i “4+1+1”. Ukoliko je broj K unaprijed *fiksiran*, sasvim je lako generirati tražene rastave pomoću K ugniježđenih petlji. Međutim, taj pristup ne možemo koristiti u slučaju kada K nije unaprijed poznat. Stoga, moramo razmišljati na drugi način. Primijetimo prvo da je problem trivijalan za $K=1$ (tada je jedina moguća rastava upravo sam broj N). S druge strane, ukoliko neka od mogućih rastava broja N na K sabiraka počinje nekim brojem I , tada ostali sabirci predstavljaju rastavu na $K-1$ sabiraka broja $N-I$. Ovaj zaključak već ukazuje na rekurzivno rješenje. Također, svi preostali sabirci (kojih ima $K-1$) ne mogu biti manji od 1, odnosno njihov zbir ne može biti manji od $K-1$. Odavde slijedi da I ne može da bude veći od $N-K+1$, jer bi u suprotnom zbir svih sabirama bio veći od $I+K-1$, što je veće od N (a to je, prema postavci problema, nemoguće). Ovim se nazire kostur rekurzivnog algoritma za rješavanje postavljenog problema, koji bismo mogli iskazati na sljedeći način:

- *Prikaz svih rastava broja N na K sabiraka:*
 - *Ako je $K=1$:*
 - *Ispisi N*
 - *U suprotnom:*
 - *Za sve vrijednosti I od 1 do $N-K+1$:*
 - *Prikaži sve rastave broja $N-I$ na $K-1$ sabiraka, ali ispred svake rastave prethodno ispiši sabirak I .*

Ovim je formirana osnovna ideja za rješavanje problema. Međutim, ostaje da se riješe još neke poteškoće, na koje se često nailazi pri rekurzivnom rješavanju iole težih problema. U ovom slučaju je problematičan posljednji korak algoritma, odnosno njegov dio koji glasi "*ali ispred svake rastave prethodno ispiši sabirak I* ". On nas sprečava da ovaj korak algoritma prosto svedemo na rekurzivno pozivanje iste funkcije za rastavu broja $N-I$ na $K-1$ sabiraka. Nažalost, ovaj korak se ne može prosto razbiti na dva neovisna koraka koji bi glasili "*Ispisi sabirak I* " i "*Prikaži sve rastave broja $N-I$ na $K-1$ sabiraka*". Naime, u tom slučaju bi se sabirak I ispisao samo *jedanput* prije ispisivanja svih rastava broja $N-I$ na $K-1$ sabiraka, tako da bi samo prva rastava imala sve sabirke. Stoga je očigledno potrebno nekako sabirak I proslijediti funkciji koja će ispisati rastave broja $N-I$ na $K-1$ sabiraka (zapravo, samoj sebi). Sada se postavlja pitanje *na kojem mjestu ispisati taj proslijedeni sabirak*. Istim rezonovanjem koje smo upravo primijenili možemo zaključiti da gdje god da ubacimo njegov ispis, on ponovo neće biti ispisani dovoljan broj puta, jer se pri svakom njegovom ispisu generira znatno više novih rastava u rekurzivnim pozivima unutar petlje (u kojima se taj sabirak ne ispisuje). Slijedi da je jedino rješenje *prikupljati* sve sabirke koji treba da se ispišu u niz i odgoditi njihov ispis sve do trenutka kada se vrši ispis sabirka N (tj. kada se problem svede na situaciju za $K=1$). Ovim dobijamo nešto razrađeniju varijantu algoritma za rješavanje problema, u kojem se već jasno uočava mogućnost neposrednog rekurzivnog pozivanja:

- Prikaz svih rastava broja N na K sabiraka:
 - Ako je $K = 1$:
 - Ispisi sve do tada prikupljene sabirke (ako ih ima)
 - Ispisi N
 - U suprotnom:
 - Za sve vrijednosti I od 1 do $N-K+1$:
 - Stavi sabirak I u popis sabiraka
 - Prikaži sve rastave broja $N-I$ na $K-1$ sabiraka

Ovaj algoritam se već može relativno lako prevesti u odgovarajuću funkciju. Jedino ostaje pitanje gdje definirati niz u koji ćemo prikupljati sabirke. Jedna mogućnost je da ga definiramo kao klasičnu (automatsku) lokalnu promjenljivu unutar funkcije. Međutim, takav pristup je veoma loš, prvenstveno zbog toga što bi tada svaka instanca funkcije posjedovala svoju instancu čitavog niza, što bi dovelo do veoma rasipnog trošenja memorije. Pored toga, te instance su posve odvojene jedna od druge, tako da bi se niz morao prenositi kao parametar u funkciju, jer bi jedino tako nova instanca funkcije mogla da pristupi vrijednostima koje je upisala prethodna instanca. Kako nama uopće ne odgovara da svaka instanca funkcija ima svoju instancu niza, znatno je bolje taj niz definirati kao *statičku promjenljivu* unutar funkcije (mnogo lošija alternativa je definiranje tog niza kao *globalne promjenljive*). Naime, već smo rekli da su statičke promjenljive zajedničke za sve instance iste funkcije pri rekursivnim pozivima. Na taj način je niz lokaliziran unutar funkcije (u smislu da mu se ne može pristupiti izvan same funkcije) a ipak je zajednički za sve instance funkcije. Općenito gledano, statičke promjenljive treba u rekursivnim rješenjima koristiti kad god je potrebno da više instanci iste funkcije dijele zajedničku promjenljivu. Na taj način dobijamo sljedeću izvedbu funkcije koja predstavlja rješenje traženog problema:

```
void PrikaziRastavuNaSabirke(int n, int k) {
    static int niz[100], indeks(0);
    if(k == 1) {
        for(int i = 0; i < indeks; i++) cout << niz[i] << "+";
        cout << n << endl;
    }
    else for(int i = 1; i <= n + 1 - k; i++) {
        niz[indeks++] = i;
        PrikaziRastavuNaSabirke(n - i, k - 1);
        indeks--;
    }
}
```

Primijetimo da smo u ovoj funkciji definirali i statičku promjenljivu "indeks". Njena uloga je da vodi evidenciju o rednom broju (indeksu) sabirka koji se upravo razmatra, sa ciljem njegovog smještanja na pravo mjesto u niz (zapravo, ova promjenljiva je analogna promjenljivoj "dubina" u primjeru Hanojskih kula, samo je upotrijebljena za drugu svrhu). Alternativno smo ovu evidenciju mogli proslijediti kao dodatni parametar u funkciju, međutim ovakvo rješenje je efikasnije (jer se izbjegava nepotrebno gomilanje instanci parametra). Treba napomenuti da je, bez obzira na relativnu kratkoću funkcije "PrikaziRastavuNaSabirke", njen tok veoma teško pratiti. Najbolje je da pokušate da nacrtate shematski prikaz toka izvršavanja ove funkcije za manje vrijednosti N i K (preporučuje se $N=6$ i $K=3$ ili $K=4$). Nakon toga će Vam mnoge stvari postati mnogo jasnije.

Prethodni primjer je zaista težak i predstavlja, na izvjestan način, test zrelosti koji pokazuje u kojoj mjeri ste sposobni da savladate naprednije tehnike programiranja. Ukoliko ne uspijete da u potpunosti razumijete ovo rješenje, vjerovatno nikada nećete moći biti u onoj kategoriji programera koji bi se mogli nazvati "rješavači problema" (na sreću, to ne mora svako da bude). Naime, prikazano rješenje može se

posmatrati kao specijalan slučaj jedne veoma općenite strategije za rješavanje brojnih problema poznate pod nazivom *povratno pretraživanje* (engl. *backtracking*), koja se tipično realizira rekurzivno na način koji se principijelno zasniva na istim idejama kao i opisani primjer. S druge strane, ukoliko ste uspjeli da u potpunosti "svarite" ovo rješenje, možete biti sigurni da Vam niti jedna tehnika programiranja neće biti "nedokučiva". Prije nego što zaista kažete da razumijete ovo rješenje, morate biti sigurni da razumijete sljedeće detalje: zbog čega je neophodno pamćenje sabiraka u nizu, zbog čega je neophodno da niz bude deklariran kao staticki, i kakva je tačno uloga parametra "indeks". Na kraju, sami sebi odgovorite na pitanje šta bi se tačno dogodilo ukoliko niz "niz" ne bi bio definiran kao staticki (pri tome je neophodno da predvidite kako bi se tačno program ponašao).

Opisani problem rastavljanja broja na sabirke spada u klasu tzv. *kombinatornih problema*. Za ove probleme je karakteristično da se prilikom njihovog rješavanja javlja potreba za umetanjem *unaprijed neodređenog broja petlj jedne unutar druge*. Kao što je poznato, tako nešto nije direktno ostvarivo u jeziku C++. Razmotreni primjer prikazuje kako se tako nešto može *simulirati* uz pomoć rekurzije. Slična ideja se može primijeniti na mnoge druge kombinatorne probleme. Tako se, recimo, slično rješenje može primijeniti za realizaciju funkcije koja ispisuje sve *permutacije* nekog zadalog skupa elemenata, koji mogu biti zadani npr. u nekom nizu. Na primjer, sve permutacije skupa elemenata {1, 2, 3} glase {1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2} i {3, 2, 1}. Ukoliko želite da okušate svoju uspješnost na polju rješavanja problema, pokušajte napraviti funkciju sa jednim parametrom N koja će ispisati sve moguće permutacije skupa prirodnih brojeva od 1 do N (ideja za rješavanje je dosta slična ideji iz prethodnog primjera, samo što se tom prilikom moraju vršiti razmjene pojedinih elemenata niza). Poredak ispisa nije bitan (preciziranjem poretku ispisa permutacija problem se osjetno otežava). Odmah na početku treba upozoriti da rješenje nije nimalo jednostavno, mada se može iskazati prilično kratkom funkcijom. Već smo u više navrata naglasili da dužina odnosno kratkoća rješenja nisu ni u kakvoj relaciji sa njegovom složenošću!

Iz dosadašnjih razmatranja smo vidjeli da su rekurzivna rješenja tipično manje efikasna od nerekurzivnih rješenja istih problema, ali da je rekurzivna rješenja često mnogo lakše sastaviti. Dokazuje se da za svako rekurzivno rješenje uvijek postoji nerekurzivno rješenje koje je *efikasnije* od tog rekurzivnog rješenja, kako po pitanju utroška memorijskih resursa, tako i po pitanju brzine izvršavanja. Stoga je prirodno postaviti pitanje *ispлати ли се* tražiti nerekurzivno rješenje nekog problema za koje je poznato rekurzivno rješenje. Odgovor zavisi od toga *koliko je nerekurzivno rješenje efikasnije*, što nažalost nije lako utvrditi. U svim dosadašnjim primjerima, rekurzivno rješenje nije mnogo neefikasnije od nerekurzivnog rješenja. Stoga, na primjer, za problem Hanojskih kula i problem rastave broja na sabirke nije previše isplativo sastavljati nerekurzivna rješenja, s obzirom da su za ova dva primjera ona mnogo složenija od rekurzivnih rješenja. Naravno, računanje faktorijela je bolje realizirati nerekurzivno, jer je u tom primjeru nerekurzivno rješenje također posve jednostavno. Međutim, sljedeći primjer će nas uvjeriti da postoje problemi kod kojih je rekurzivno rješenje *užasno neefikasno* u odnosu na nerekurzivno rješenje. Pretpostavimo da želimo da napravimo funkciju sa parametrom n koja vraća kao rezultat n -ti Fibonačijev broj F_n (radi jednostavnosti, nazovimo ovu funkciju posve kratkim imenom "F") Sama definicija Fibonačijevih brojeva u obliku $F_1=F_2=1$ i $F_n=F_{n-1}+F_{n-2}$ prosto nameće sljedeće rekurzivno rješenje:

```
long int F(int n) {
    if(n < 3) return 1;
    else return F(n - 1) + F(n - 2);
}
```

Nije nikakav problem isprobati ovu funkciju i ustanoviti da ona radi. Ipak, testiranje ove funkcije za različite vrijednosti parametra n dovodi do iznenadujućih zaključaka. Za $n < 20$ funkcija će ponuditi rezultat praktično trenutno. Međutim, već za $n = 40$ na rezultat ćete čekati prilično dugo (više desetina

sekundi) čak i na prilično brzim računarima. Za $n=50$ na rezultat ćete čekati satima, dok se može lako dokazati da za $n = 100$ nećete dočekati rezultat za Vašeg života čak ni ukoliko funkciju isprobate na najbržim računarima koji u ovom trenutku postoje na svijetu! Ovo može djelovati veoma šokantno, s obzirom na činjenicu da se F_{100} može izračunati pomoću svega 98 sabiranja, što se može obaviti npr. pomoću `for` petlje (a moguće je posve lako uraditi čak i ručno). Da bismo vidjeli šta je uzrok ovako neobičnom ponašanju, razmotrimo sljedeću sliku koja prikazuje tok izvršavanja funkcije “ F “ kada se pozove za $n = 7$:

$$\begin{array}{ccccccccc}
 & & & F(7) & & & & & \\
 & & & =F(6)+F(5) & & & & & \\
 \\
 & & =F(5)+F(4) & & & & =F(4)+F(3) & & \\
 \\
 & =F(4)+F(3) & & =F(3)+F(2) & & =F(3)+F(2) & & =F(2)+F(1) & \\
 \\
 & =F(3)+F(2) & =F(2)+F(1) & =F(2)+F(1) & =1 & =F(2)+F(1) & =1 & =1 & =1 \\
 \\
 & =F(2)+F(1) & =1 & =1 & =1 & =1 & =1 & & \\
 \\
 & =1 & & =1 & & & & &
 \end{array}$$

Iz ovog dijagrama vidimo da je prilikom računanja vrijednosti “ $F(7)$ “, funkcija “ F “ pozvala samu sebe 25 puta, bez obzira što se F_7 može izračunati pomoću svega 5 sabiranja. Pored toga, iz dijagrama se vidi da se neki izrazi bespotrebno računaju više puta, iako su već jedanput izračunati. Na primjer, izraz “ $F(3) + F(2)$ “ nepotrebno se računa tri puta, dok se izraz “ $F(2) + F(1)$ “ računa čak pet puta. Sa porastom n situacija se drastično pogoršava. Matematičkom indukcijom se lako može pokazati da će ovako napisana funkcija “ F “ pozvati sama sebe $2F_k - 1$ puta za argument $n=k$. Stoga, trajanje izračunavanja raste eksponencijalno sa porastom n , s obzirom da i F_n eksponencijalno raste sa porastom n . Naime, posve je lako dokazati (također pomoću matematičke indukcije) da za Fibonačijevе brojeve vrijedi eksplisitna formula

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right]$$

Tako će za računanje $F_{50} = 12586269025$ funkcija pozvati sama sebe 25172538049 puta, što zaista mora da traje prilično dugo. Još gore, trajanje izračunavanja broja $F_{100} = 708449696358523830149$ ovako napisanom funkcijom trajalo bi oko 224 godine čak uz pretpostavku da računar može obavljati *bilion* (10^{12}) poziva u sekundi (što je otprilike slučaj sa najmoćnijim računarima kojima danas raspolažemo). Prema tome, rekurzivna varijanta funkcije za računanje Fibonačijevih brojeva je, usprkos tome što izgleda lijepo i elegantno, potpuno neupotrebljiva!

Posljednji primjer posve ubjedljivo ukazuje na to da treba dobro promisliti prije nego što se upotrijebi neko rekurzivno rješenje. Kao pravilo, može se reći da svako rekurzivno rješenje u kojem funkcija poziva sama sebe više od jedanput u toku svog života i u kojem se veličine potproblema koji se rekurzivno rješavaju sporo smanjuju (npr. za konstantan iznos) obavezno vodi eksponencijalnom trajanju izvršavanja

(takvu situaciju imamo u posljednja tri primjera). Takvo rješenje je prihvatljivo jedino ukoliko znamo da problem koji se rješava sam po sebi zahtijeva eksponencijalno trajanje izvršavanja. Na primer, problem Hanojskih kula je upravo takav (jer je neophodan broj poteza eksponencijalna funkcija broja kula), a može se pokazati da to vrijedi i za problem rastavljanja broja na sabirke (tačan broj rastava broja n na k sabiraka iznosi $n!/[k!(n-k)!]$, što recimo za $k=n/2$ po Stirlingovoj formuli iznosi približno $2^{n+1/2}/\sqrt{n\pi}$). Međutim, računanje n -tog Fibonačijevog broja može se izvesti sa $n-2$ sabiranja, odnosno izračunavanje se može izvesti u vremenu koje *linearno* ovisi od n . Kao posljedicu imamo da je rekurzivno računanje Fibonačijevih brojeva katastrofalno.

U slučaju Fibonačijevih brojeva, činjenica da je rekurzivno rješenje veoma neefikasno ne predstavlja preveliku štetu, s obzirom da se isti problem lako rješava nerekurzivno. Međutim, postoje brojni problemi za koje je teško sastaviti nerekurzivno rješenje, a za koje je direktno sastavljeno rekurzivno rješenje veoma neefikasno iz istog razloga kao u slučaju računanja Fibonačijevih brojeva. Stoga su razvijene brojne metode za ubrzavanje rekurzivnih rješenja, koje se zasnivaju na dopunskom memoriranju već izračunatih rezultata i izbjegavanju rekurzivnih poziva za računanje onog što je već izračunato. Najpoznatija takva metoda zasniva se *rekurzija sa pamćenjem* ili *memoizacija* (bez “r”). Na ovom mjestu ćemo izložiti samo osnovnu ideju metode (u složenijim situacijama realizacija ove metode može biti praćena brojnim poteškoćama). Razmotrimo sljedeću (rekurzivnu) realizaciju funkcije “F” za računanje Fibonačijevih brojeva:

```
long int F(int n) {
    const int kapacitet(1000), smece(-1);
    static long int memorija[kapacitet];
    static bool treba_inicijalizacija(true);
    if(treba_inicijalizacija) {
        for(int i = 0; i < kapacitet; i++) memorija[i] = smece;
        treba_inicijalizacija = false;
    }
    if(memorija[n] != smece) return memorija[n];
    if(n < 3) return memorija[n] = 1;
    else return memorija[n] = F(n - 1) + F(n - 2);
}
```

U ovoj funkciji deklariran je (statički) niz nazvan “memorija” kapaciteta 1000 elemenata (što je moguće lako promijeniti promjenom konstante “kapacitet”). Prilikom prvog poziva ove funkcije, svi elementi ovog niza su inicijalizirani na neku vrijednost (nazovimo tu vrijednost “smeće”) koja ne može biti rezultat funkcije “F” ni za kakvu vrijednost njenog argumenta (u ovom primjeru izabrana je vrijednost “-1”, što je lako promijeniti promjenom konstante nazvane “smece”). Obratimo pažnju na koji način je obezbijeđeno da se inicijalizacija niza “memorija” izvrši samo prilikom *prvog poziva* funkcije (upotrijebljena je statička promjenljiva “treba_inicijalizacija”, čija je početna vrijednost “**true**”, a koja se nakon obavljenje inicijalizacije niza postavlja na vrijednost “**false**” i čija se vrijednost nikada više ne vraća na vrijednost “**true**”, tako da se inicijalizacija niza nikada više neće ponoviti). Uloga niza “memorija” je za čuvanje već izračunatih vrijednosti funkcije “F”, što ćemo pokazati u izlaganju koje slijedi.

Razmotrimo sada šta se dešava nakon obavljene inicijalizacije (koja se, ponovimo, obavlja samo pri prvom pozivu funkcije). Ukoliko se u nizu “memorija” na poziciji “n” ne nalazi “smeće”, odnosno ukoliko je vrijednost funkcije *već ranije izračunata*, kao rezultat funkcije se prosti vrati memorirana vrijednost iz niza (ovo se svakako neće nikada desiti pri prvom pozivu funkcije, jer tada cijeli niz sadrži samo “smeće”). U suprotnom se vrijednost funkcije “F” računa pomoću rekurzivne formule, ali se izračunata vrijednost prije povratka iz funkcije *pamti u nizu*. Primjetimo da je pomalo čudna konstrukcija

```
return memorija[n] = F(n - 1) + F(n - 2);
```

zapravo samo skraćeni zapis za sekvencu od sljedeće dvije naredbe:

```
memorija[n] = F(n - 1) + F(n - 2);
return memorija[n];
```

Na ovaj način će vrijednost funkcije biti odmah dostupna u slučaju da se u toku kasnijih rekurzivnih poziva pojavi potreba za ponovnim računanjem funkcije “F” za *istu vrijednost argumenta* (a kao što smo vidjeli, takva potreba će se pojaviti veoma često). Na taj način se izbjegavaju rekurzivni pozivi za računanje već izračunatog. Uštede koje se na ovaj način postižu mogu biti drastične. Nije teško vidjeti da će na ovaj način za računanje k -og Fibonačijevog broja funkcija “F” rekurzivno pozvati sama sebe svega k (umjesto $2F_k - 1$) puta, što je ogromno ubrzanje.

Ubrzana verzija rekurzivne funkcije za računanje Fibonačijevih brojeva mogla se napisati i jednostavnije, ukoliko primijetimo da se statičke promjenljive automatski inicijaliziraju *na nulu* ukoliko nije eksplicitno rečeno drugačije, i ukoliko primijetimo da nula također ne može biti nikada vraćena kao rezultat funkcije “F”. Ovo nam daje mogućnost da kao “smeće” proglašimo upravo nulu, i izbjegnemo potrebu za eksplicitnom inicijalizacijom memorije, kao u sljedećoj realizaciji:

```
long int F(int n) {
    const int kapacitet(1000);
    static long int memorija[kapacitet];
    if(memorija[n] != 0) return memorija[n];
    if(n < 3) return memorija[n] = 1;
    else return memorija[n] = F(n - 1) + F(n - 2);
}
```

Ipak, rješenje sa eksplicitnom inicijalizacijom je univerzalnije.

Primijetimo da opisani metod za ubrzavanje rekurzije posjeduje jedan ozbiljan nedostatak: trošenje memorijskih resursa. Naravno, funkciju za računanje Fibonačijevih brojeva je besmisleno realizirati na opisani način (s obzirom da za isti problem postoji vrlo jednostavno i efikasno iterativno rješenje, koje ne troši nikakve suvišne memorijске resurse), ali je ovdje cilj bio samo ilustracija tehnike ubrzavanja rekurzije na jednom jednostavnom primjeru.

Kao što je već rečeno, opisani metod često je povezan sa praktičnim poteškoćama. Pomenimo samo neke od njih. Šta raditi u slučaju da argument funkcija može uzimati vrijednosti koje nisu prirodni brojevi (nego npr. realni brojevi)? Kako ocijeniti koliki kapacitet niza upotrijebiti za memoriranje međurezultata? Šta raditi u slučaju da funkcija ima više argumenata? Ovo su samo neke od potencijalnih poteškoća (od kojih je posljednja naročito nezgodna). Metodi za prevazilaženje ovih poteškoća razmatraju se u teoriji algoritama. Kao vježbu, razmislite sami kako biste donekle mogli riješiti prvu od opisanih poteškoća (napomenimo da *efikasno* rješavanje ove poteškoće zahtijeva znatno složenije programerske tehnike, kao što su upotreba tzv. *binarnih stabala* ili *heš tabela*, koje usput rješavaju i drugu od opisanih poteškoća).

Interesantno je razmotriti sljedeći primjer, koji pokazuje da problem izbora kapaciteta niza za memoriranje međurezultata može da bude prilično nezgodan. Neko bi mogao pomisliti da su rekurzivne funkcije uvijek takve prirode da im se argumenti pri rekurzivnim pozivima uvijek smanjuju, tako da je dovoljno da kapacitet bude veći od najvećeg očekivanog argumenta. Sljedeći primjer pokazuje da ne mora uvijek biti tako. Razmotrimo sljedeću rekurzivnu funkciju (tzv. *Ulamova funkcija*):

$$U(n) = \begin{cases} U(n/2) + 1 & \text{za } n \text{ parno} \\ U(3n+1) + 1 & \text{za } n \text{ neparno } (n \neq 1) \\ 0 & \text{za } n = 1 \end{cases}$$

Probajmo izračunati npr. $U(7)$. Nije teško izračunati da je $U(7) = 16$. Međutim, prilikom računanja $U(7)$, argument n u rekurzivnim pozivima uzima redom vrijednosti 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2 i 1, odnosno ponaša se posve haotično. Na osnovu početne vrijednosti argumenta n praktično je nemoguće predvidjeti kakve će sve vrijednosti uzimati ovaj argument u toku rekurzivnih poziva. Ovo naravno otežava predviđanje kapaciteta memorije koji bi se trebao koristiti za ubrzavanje rekurzije u slučaju da ovu funkciju želimo realizirati rekurzivno (zanemarimo činjenicu da se ova funkcija veoma jednostavno može implementirati *iterativno*, pomoću “**while**” petlje, bez korištenja rekurzije – uradite to sami kao vježbu). Prethodni primjer haotičnog ponašanja argumenta ove funkcije pri rekurzivnim pozivima dovodi do prirodnog pitanja da li se za svaku vrijednost početnog argumenta n vrijednost $n=1$ uopće dostiže, odnosno može li se desiti da se rekurzija protegne u beskonačnost? Hipotezu da se ova rekurzija uvijek završava u konačno mnogo koraka za svaku početnu vrijednost argumenta n postavili su dosta davno *Lothar Collatz* i *Stanislaw Ulam* (neovisno jedan od drugog), ali do današnjeg dana niko nije uspio niti da dokaže niti da opovrgne postavljenu hipotezu, iako je za rješenje ponuđena velika nagrada. Ovo je jedan od poznatih otvorenih matematičkih problema današnjice.

Potrebno je napomenuti da svako rekurzivno rješenje u kojem funkcija rekurzivno poziva sama sebe više od jedanput u toku svog života ne mora nužno biti sporo. Naime, takva rješenja mogu biti i jako brza, pod uvjetom da se veličine potproblema koji se rekurzivno rješavaju rapidno smanjuju (na primjer, na polovinu prethodne veličine pri svakom rekurzivnom pozivu). Primjer jednog takvog brzog rekurzivnog rješenja biće naveden u poglavljju o sortiranju.

Ovom prilikom smo govorili samo o tzv. *prostoj rekurziji*, u kojoj funkcija poziva samu sebe. Postoji i složenija metodologija nazvana *uzajamna rekurzija* (engl. *mutual recursion*), u kojoj se više funkcija međusobno poziva između sebe (npr. funkcija “A” poziva funkciju “B”, funkcija “B” poziva funkciju “C”, dok funkcija “C” ponovo poziva funkciju “A”). Uzajamna rekurzija je također izuzetno moćna programerska tehnika, ali je složenost problema koji zahtijevaju rješavanje ovom tehnikom isuviše velika da bi bili prikazani na ovom mjestu.

Na kraju izlaganja o rekurzijama moramo reći da rekurzija iz dana u dan dobija sve veći značaj u modernom programiranju, jer su mnogi savremeni koncepti u suštini rekurzivne prirode. Na primjer, operacije nad folderima i menijima često se izvode rekurzivno s obzirom na činjenicu da folderi mogu sadržavati podfoldere, meniji mogu sadržavati podmenije itd. i to do proizvoljne dubine. Stoga, prilikom pretrage nekog foldera (npr. pri traganju za datotekom navedenog imena), potrebno je *rekurzivno* pregledati sve njegove podfoldere (koji opet mogu sadržavati podfoldere na koje se primjenjuju dalji rekurzivni pozivi). Pojam aritmetičkog ili logičkog izraza u matematici također se definira rekurzivno, odakle slijedi da su postupci za manipulacije sa izrazima u principu rekurzivne prirode (i to često uzajamno rekurzivne). Čak i same definicije nekih pojmove iz jezika C++ su rekurzivne. Npr. definicija pojma prave konstante je uzajamno rekurzivna: prava konstanta je konstanta koja je inicijalizirana brojem ili *konstantnim izrazom*, dok je konstantni izraz koji od operanada sadrži samo brojeve i *prave konstante*.

Rekurzije predstavljaju osnovnu metodologiju programiranja u većini neproceduralnih jezika (kao što su npr. *LISP*, *Prolog* i *ML*), i nezamjenljive su za programiranje logičkih igara (poput šaha) kao i metoda koje spadaju u domen vještačke inteligencije (u posljednja dva slučaja naročito se mnogo koristi upravo uzajamna rekurzija). Razlog za to leži u činjenici da su definicije mnogih pojmove koji se susreću u realnom životu zapravo rekurzivne prirode (na primjer, definicija “Vaš predak je osoba koja je ili vaš

roditelj, ili je predak nekog od vaših roditelja.” predstavlja tipičan primjer rekurzivne definicije). Mnogi kažu da je rekurzija “jedno od najubojitijih programerskih oružja” kojim se često mogu savladati i najteži problemi, ali kao i svako oružje, može nanijeti neprocjenjivu štetu dospije li “u pogrešne ruke”.

22. Nizovi znakova i stringovi

Svi programi koje smo do sada razmatrali obrađivali su pretežno *brojčane podatke*. Međutim, u praksi se često javlja potreba za programima koji obrađuju *tekstualne podatke*, odnosno podatke nenumeričke prirode. Stoga je u jezik C++ ugrađena podrška za rad sa tekstualnim podacima, odnosno *stringovima* kako se oni običajeno nazivaju u programiranju.

C++ podržava dva načina za rad sa stringovima. Prvi (klasični) način posmatra stringove kao obične *nizove znakova*, pri čemu su podržane neke dodatne konvencije koje ne vrijede za obične nizove. Ovaj način rada je po postanku stariji, i preuzet je iz jezika C, uz neke neznatne dopune. Drugi (moderni) način je u jezik C++ uveden znatno kasnije i posmatra stringove kao poseban tip podataka nazvan “*string*”, koji se može koristiti neovisno od nizova. Ovaj tip podataka je, slično tipu “*vector*”, izvedeni tip podataka (definiran u istoimenoj biblioteci također nazvanoj “*string*”), definiran uz pomoć tzv. *klasa*, koje ćemo kasnije detaljno obrađivati. U ovom poglavlju ćemo detaljnije razmotriti prvi način, odnosno tretiranje stringova kao nizova znakova. Drugi način, odnosno upotreba tipa “*string*”, biće na ovom mjestu objašnjena samo informativno. Mada je upotreba ovog tipa znatno jednostavnija i fleksibilnija nego korištenje klasičnog načina rada sa stringovima, masovnije korištenje ovog tipa podataka ćemo odgoditi do trenutka kada u potpunosti postanu jasni principi na kojima je on internu implementiran. U suprotnom, naviknemo li se prebrzo na korištenje tipa “*string*”, nećemo moći u dovoljnoj mjeri ovladati principima na kojima se zasniva kreiranje vlastitih tipova podataka, niti ćemo u potpunosti moći razumjeti sam tip “*string*”.

Osnovne ideje rada sa stringovima kao nizovima znakova već su uvedene u poglavljima koja su govorila o znakovnim tipovima, naredbama ponavljanja i nizovima. Tako smo, na primjer, vidjeli da sljedeći program učitava sa tastature rečenicu (odnosno proizvoljan *slijed znakova* ne duži od 1000 znakova) i ispisuje je unatraške:

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

int main() {
    const int MaxBrojZnakova(1000);
    char znak, recenica[MaxBrojZnakova];
    cout << "Unesi rečenicu: ";
    int broj_znakova(0);
    while((znak = cin.get()) != '\n' && broj_znakova < MaxBrojZnakova)
        recenica[broj_znakova++] = znak;
    cout << "Rečenica izgovorena naopako glasi: ";
    for(int i = broj_znakova - 1; i >= 0; i--) cout << recenica[i];
    return 0;
}
```

Sa do sada izloženim konceptima bi se, barem principijelno, mogao napisati bilo koji program koji obrađuje podatke tekstualne prirode. Međutim, iole složenija manipulacija sa tekstualnim podacima bila bi izuzetno mukotrpsna ukoliko bi se oni morali obrađivati isključivo *znak po znak*. Stoga C++ posjeduje skupinu funkcija koja omogućava manipulacije sa skupinom znakova *kao cjelinom*. Na primjer, umjesto učitavanja znakova sa tastature *znak po znak* uzastopnim pozivanjem funkcije “*cin.get*”, moguće je pomoću funkcije “*cin.getline*” učitati čitav niz znakova *odjedanput*. Ova funkcija se može koristiti na više načina, a obično se koristi sa dva ili tri parametra. Ukoliko se koristi sa *dva parametra*, tada prvi parametar predstavlja *niz znakova u koji se smještaju pročitani znakovi*, dok drugi parametar predstavlja broj koji je *za jedinicu veći od maksimalnog broja znakova koji će se pročitati* (smisao ove razlike od

jednog znaka biće uskoro razjašnen). U slučaju da se prije toga dostigne kraj reda, biće pročitano manje znakova, odnosno neće biti zatražen unos novih znakova. Drugim riječima, naredba

```
cin.getline(recenica, 50);
```

zatražiće od korisnika unos sa ulaznog uređaja (recimo tastature), a zatim smjestiti *najviše 49 pročitanih znakova iz ulaznog toka* u niz “*recenica*” (ukoliko je uneseno manje od 49 znakova smjestiće se svi znakovi). Kao i do sada, pročitani znakovi se *uklanjaju* iz ulaznog toka (eventualno nepročitani znakovi ostaju i dalje u ulaznom toku). Ukoliko se funkcija “*cin.getline*“ pozove sa *tri parametra* (što se rjeđe koristi), tada prva dva parametra imaju isto značenje kao i u prethodnom slučaju, dok treći parametar predstavlja *znak koji će se koristiti kao oznaka završetka unosa*. Drugim riječima, čitaće se svi znakovi sve dok se ne pročita navedeni znak ili dok se ne pročita onoliko znakova koliko je zadano drugim parametrom. Tako će poziv funkcije

```
cin.getline(recenica, 50, '.'');
```

zatražiti unos sa ulaznog uređaja, a zatim pročitati najviše 49 znakova iz ulaznog toka u niz “*recenica*”, pri čemu se čitanje prekida ukoliko se pročita znak ‘.’ (tačka). Pri tome se tačka *uklanja iz ulaznog toka*, ali se *ne prebacuje u niz*. Ukoliko se dostigne kraj reda, a nije pročitano 49 znakova, niti je pročitan znak ‘.’, biće zatražen unos novih znakova sa ulaznog uređaja. Treba napomenuti da je poziv funkcije “*cin.getline*“ sa dva parametra identičan pozivu ove funkcije sa tri parametra u kojem je treći parametar jednak oznaci kraja reda ‘\n’, Drugim riječima, sljedeća dva poziva su identična:

```
cin.getline(recenica, 50);
cin.getline(recenica, 50, '\n');
```

Vrijedi još napomenuti da se i funkcija “*cin.get*“ također može pozivati sa dva ili tri parametra, poput funkcije “*cin.getline*“, sa veoma sličnim dejstvom. Pri tome je jedina razlika što za razliku od funkcije “*cin.getline*“ funkcija “*cin.get*“ *ne uklanja granični znak* (oznaku kraja reda ili znak naveden kao treći parametar) iz ulaznog toka, nego ga ostavlja u ulaznom toku.

U slučaju kada smo znakove iz ulaznog toka čitali znak po znak uzastopnim pozivanjem funkcije “*cin.get*”, mogli smo tačno znati koliko smo znakova unijeli, s obzirom da smo brojali unesene znakove (pomoću promjenljive “*broj_znakova*”). Međutim, ukoliko unos vršimo pozivom funkcije “*cin.getline*“ javlja se problem *kako da utvrdimo koliko smo znakova unijeli*. Jedan od načina je **pozvati funkciju “*cin.gcount*“ koja vraća kao rezultat broj znakova koji su pročitani prilikom posljednjeg čitanja**. Ova funkcija *nema parametara*, tj. poziva se pozivom poput “*cin.gcount()*“. Međutim, u jezicima C i C++ ovaj problem, kao i mnogi srodni problemi koji nastaju zbog činjenice da tekstovi sa kojima radimo mogu biti različite dužine, riješen je na mnogo univerzalniji način uvođenjem jednog posebnog znak sa ASCII šifrom 0, koji se uzima kao oznaka kraja stringa (to nije znak ‘0’ nego *znak čija je ASCII šifra nula*). Nula je iskorištena kako zbog jednostavnosti manipulacija sa brojem 0, tako i zbog činjenice da 0 nije ASCII šifra ni jednog “normalnog” znaka koji bi nam mogao zatrebatи pri ispisu. Ovaj specijalni znak tipično se naziva “*NUL*”. Konvencija o tzv. *nul-terminalanim stringovima*, po kojoj se nailazak na znak “*NUL*“ tretira kao kraj stringa bez obzira da li iza njega slijedi još znakova, preuzeta je iz jezika C, i često se uzima kao definicija *stringa* (odnosno stringovi se definiraju kao nizovi znakova čiji je logički završetak označen znakom “*NUL*”). Gotovo sve funkcije jezika C++ poštjuju ovu konvenciju. Zbog toga, funkcije “*cin.getline*“ odnosno “*cin.get*“ (u slučaju kada se ona koristi sa parametrima) obavezno smještaju znak “*NUL*” nakon posljednjeg pročitanog znaka. Tako, ukoliko nakon poziva funkcije

```
cin.getline(recenica, 50);
```

sa tastature unesemo frazu “Za 10 dana”, u nizu “*recenica*“ ćemo imati ovakvu situaciju (brojevi u

zagradi prikazuju ASCII šifre navedenih znakova):

0	1	2	3	4	5	6	7	8	9	10	11	12	13
'z'	'a'	' '	'1'	'0'	' '	'd'	'a'	'n'	'a'	NUL	?	?	?

Sa oznakom “?” smo označili elemente niza čiji je sadržaj nepredvidljiv (oni će sadržavati vrijednosti koje su se od ranije zatekle na tim lokacijama). Na osnovu konvencije o “NUL” graničniku postaje jasno zbog čega je maksimalni broj znakova koji se čitaju iz ulaznog toka za jedan manji od drugog parametra funkcija “cin.getline” odnosno “cin.get”. Naime, ove funkcije predviđaju činjenicu da će jedan element niza uvijek biti iskorišten za smještanje znaka “NUL”.

Uz konvenciju o nul-terminaliranim stringovima, posve je lako napisati funkciju koja će odrediti koliko je zaista dugačak string, odnosno koliko se znakova nalazi ispred graničnika:

```
int DuzinaStringa(const char niz[]) {
    int brojac = 0;
    while(niz[brojac] != 0) brojac++;
    return brojac;
}
```

Ovu funkciju možemo iskoristiti npr. na sljedeći način:

```
char a[100];
cout << "Unesi neki niz znakova: ";
cin.getline(a, sizeof a);
cout << "Unijeli ste " << DuzinaStringa(a) << " znakova.";
```

Primijetimo kako je u ovom primjeru operator “**sizeof**” iskorišten unutar drugog parametra u pozivu funkcije “**cin.getline**”. Na taj način sam poziv je postao neovisan od deklaracije niza “a”, tj. ostaje isti ukoliko se kasnije kapacitet ovog niza promijeni.

Jezik C++ već posjeduje ugrađenu funkciju nazvanu “**strlen**” iz biblioteke “**cstring**” (zaglavljene biblioteke u jeziku C i u starijim verzijama C++ kompjlera naziva se “**string.h**” a ne “**cstring**”). Ova funkcija radi potpuno istu stvar kao i upravo napisana funkcija “**DuzinaStringa**”. Dakle, u praksi, ovu funkciju nećemo pisati, nego ćemo koristiti gotovu funkciju “**strlen**”. Slijedi alternativna verzija programa koji ispisuje naopako rečenicu učitanu sa tastature:

```
#include <iostream>
#include <cstring>

using namespace std;

int main() {
    char recenica[1000];
    cout << "Unesi rečenicu: ";
    cin.getline(recenica, sizeof recenica);
    cout << "Rečenica izgovorena naopako glasi: ";
    for(int i = strlen(recenica) - 1; i >= 0; i--) cout << recenica[i];
    return 0;
}
```

Mada operatori umetanja “<<” i izdvajanja “>>” generalno ne mogu raditi sa čitavim nizovima nego samo sa individualnim elementima niza, oni ipak prihvataju *nizove znakova* kao operande. Tako, ukoliko

se iza operatora “<<” navede neki niz znakova, na izlazni tok će biti ispisani svi znakovi niza jedan za drugim, sve do oznake kraja stringa (graničnika “NUL”). Ova činjenica znatno olakšava rad sa stringovima, jer omogućava pisanje konstrukcija poput:

```
char recenica[100];
cout << "Unesi neku rečenicu: ";
cin.getline(recenica, sizeof recenica);
cout << "Unesena rečenica glasi: " << recenica;
```

Slično vrijedi i za operator “>>”. Tako, mada konstrukcija “`cin >> a`“ generalno nije dozvoljena za slučaj kada promjenljiva “`a`“ predstavlja niz, ona je dozvoljena ukoliko je “`a`“ niz znakova. Ipak, ova konstrukcija nije ekvivalentna pozivu funkcije “`cin.getline`“ iz dva razloga. Prvi razlog je nepostojanje ograničenja na broj znakova koji će biti pročitani. Doduše, ovo ograničenje je moguće uvesti pozivom funkcije “`cin.width`“ ili korištenjem manipulatora “`setw`”, na analogan način kao što se pozivom funkcije “`cout.width`“ ili manipulatora “`setw`” zadaje širina ispisa na izlazni tok. Drugi razlog je činjenica da operator “`>>`” uvijek prekida izdvajanje iz ulaznog toka nailaskom na prvu prazninu (razmak, tabulator ili kraj reda), i po tom pitanju se ne može ništa učiniti. Na primjer, pretpostavimo da smo napisali sljedeću sekvencu naredbi:

```
char recenica[100];
cout << "Unesi neku rečenicu: ";
cin >> setw(sizeof recenica) >> recenica;
cout << "Unesena rečenica glasi: " << recenica;
```

Upotreboom manipulatora “`setw(sizeof recenica)`” ograničili smo maksimalni broj znakova koji će biti pročitan (pri tome se podrazumijeva da smo u program uključili biblioteku “`iomanip`” koja definira ovaj manipulator). Ipak, ovaj program ne bi radio na očekivani način. Naime, na ekranu bi iz unesene rečenice bila ispisana *samo prva riječ*, jer bi se izdvajanje iz ulaznog toka završilo *na prvom razmaku*, dok bi svi ostali znakovi (odnosno ostale riječi rečenice) ostali u ulaznom toku, i mogli bi se kasnije izdvijiti ponovnom upotreboom operatora “`>>`”. Ukoliko nam zbog nekog razloga upravo treba da iz ulaznog toka izdvajamo riječ po riječ, operator “`>>`” može biti od koristi. Ipak, u većini slučajeva, za unos stringova sa tastature znatno je povoljnije koristiti funkciju “`cin.getline`“ ili eventualno “`cin.get`“ nego operator “`>>`”. Od koristi može biti i ranije pomenuta funkcija “`cin.peek`“ bez argumenata koja vraća kao rezultat koji je tekući znak u ulaznom toku, ali bez njegovog izdvajanja iz ulaznog toka, što bi uradila funkcija “`cin.get`”.

Bitno je istaći da u slučaju kada se u istom programu naizmjenično koriste operator “`>>`” i funkcije “`cin.getline`“ odnosno “`cin.get`“ može veoma lako doći do ozbiljnih problema ukoliko programer nije oprezan. To se najlakše može dogoditi ukoliko se u istom programu unose i brojčani i tekstualni podaci. Na primjer, razmotrimo sljedeći isječak iz programa:

```
int broj;
char tekst[100];
cout << "Unesi neki broj: ";
cin >> broj;
cout << "Unesi neki tekst: ";
cin.getline(tekst, sizeof tekst);
cout << "\nBroj: " << broj << "Tekst: " << tekst << endl;
```

Ovaj isječak neće raditi kako je očekivano. Naime, probleme pravi operator izdvajanja “`>>`” koji prekida izdvajanje na prvom razmaku ili oznaci kraja reda, ali ne uklanja iz izlaznog toka znak na kojem je izdvajanje prekinuto. Tako, pretpostavimo da je na pitanje “Unesi neki broj: “ korisnik unio neki broj

(npr. 123) i pritisnuo tipku ENTER. Tada će se u ulaznom toku nalaziti znakovi '1', '2', '3' i oznaka kraja reda '\n'. Izraz "cin >> broj" će izdvajati znakove '1', '2', '3' iz ulaznog toka, ali oznaka kraja reda '\n' i dalje ostaje u ulaznom toku. Kada bi se nakon toga ponovo koristio operator ">>", ne bi bilo problema, jer on svakako podrazumijevano ignorira sve praznine, poput razmaka i oznake kraja reda. Međutim, funkcija "cin.getline" (koja ne ignorira praznine) odmah na početku pronalazi oznaku kraja reda u ulaznom toku (i pri tom ga uklanja), čime se njen rad odmah završava, i u niz "tekst" neće se smjestiti ništa osim graničnika "NUL", tako da "tekst" postaje prazan string, što svakako nije ono što bi trebalo da bude!

Postoji više jednostavnih načina za rješavanje gore opisanog problema. Na primjer, moguće je nakon učitavanja broja pozvati funkciju "cin.ignore" pomoću koje ćemo forsirano ukloniti sve znakove iz ulaznog toka, uključujući i problematičnu oznaku kraja reda. Tako, sljedeći programski isječak radi posve korektno:

```
int broj;
char tekst[100];
cout << "Unesi neki broj: ";
cin >> broj;
cin.ignore(10000, '\n');
cout << "Unesi neki tekst: ";
cin.getline(tekst, sizeof tekst);
cout << "\nBroj: " << broj << "Tekst: " << tekst << endl;
```

Poziv funkcije "cin.ignore" će zapravo isprazniti čitav ulazni tok, a ne samo ukloniti oznaku za kraj reda. Međutim, u nekim slučajevima je potrebno iz ulaznog toka ukloniti sve praznine (poput razmaka, oznaka za kraj reda, itd.) sve do prvog znaka koji nije praznina, a ostaviti znakove koji eventualno slijede (ako ih uopće ima) u ulaznom toku. Jasno je da tako nešto možemo uraditi pozivom funkcije "cin.get" u petlji. Međutim, jednostavniji način je upotrijebiti objekat nazvan "gutač praznina" (engl. *whitespace eater* ili *whitespace extractor*), koji je u jeziku C++ imenovan prosto imenom "ws". Radi se o specijalnom objektu za rad sa tokovima (poput objekta "endl"), koji je također neka vrsta manipulatora. Ovaj objekat, upotrijebljen kao drugi argument operatora izdvajanja ">>", uklanja sve praznine koje se eventualno nalaze na početku ulaznog toka, ostavljajući sve druge znakove u ulaznom toku netaknutim, i ne izazivajući nikakav drugi propratni efekat. Drugim riječima, nakon izvršenja izraza poput "cin >> ws" eventualne praznine sa početka ulaznog toka biće uklonjene. Stoga smo prethodni primjer mogli napisati i ovako:

```
int broj;
char tekst[100];
cout << "Unesi neki broj: ";
cin >> broj >> ws;
cout << "Unesi neki tekst: ";
cin.getline(tekst, sizeof tekst);
cout << "\nBroj: " << broj << "Tekst: " << tekst << endl;
```

Ovo rješenje ipak nije ekvivalentno prethodnom rješenju. Naime, u ovom primjeru se nakon unosa broja, iz ulaznog toka uklanjaju *samo praznine* (uključujući i oznaku kraja reda), ali ne i eventualni ostali znakovi. Da bismo jasnije uočili razliku, razmotrimo nekoliko sljedećih scenarija. Neka je na pitanje "Unesi neki broj:" korisnik zaista unio broj i pritisnuo tipku ENTER. U tom slučaju, oba rješenja ponašaju se identično, kao što je prikazano na sljedećoj slici:

Unesi neki broj: 123
Unesi neki tekst: ABC

```
Broj: 123 Tekst: ABC
```

S druge strane, pretpostavimo da je korisnik nakon unosa broja, prije pritiska na tipku ENTER unio razmak iza kojeg slijede neki znakovi. Rješenje koje koristi poziv funkcije “`cin.ignore()`” ignoriraće suvišne znakove unesene prilikom unosa broja, kao što je prikazano na sljedećoj slici:

```
Unesi neki broj: 123 XY
Unesi neki tekst: ABC

Broj: 123 Tekst: ABC
```

Međutim, rješenje u kojem se koristi “`ws`” objekat, ukloniće samo prazninu iz ulaznog toka. Nailaskom na poziv funkcije “`cin.getline()`” ulazni tok neće biti prazan, tako da uopće neće biti zatražen novi unos sa tastature, već će biti iskorišteni znakovi koji se već nalaze u ulaznom toku! Ovo je jasno prikazano na sljedećoj slici:

```
Unesi neki broj: 123 XY
Unesi neki tekst:
Broj: 123 Tekst: XY
```

Prikazano ponašanje nije uopće tako loše, pod uvjetom da ga korisnik očekuje. Na primjer, posmatrajmo sljedeći programski isječak, u kojem se korisnik odmah na početku obavještava da treba prvo da unese broj, a zatim tekst:

```
int broj;
char tekst[100];
cout << "Unesi neki broj, a zatim neki tekst: ";
cin >> broj >> ws;
cin.getline(tekst, sizeof tekst);
cout << "\nBroj: " << broj << "Tekst: " << tekst << endl;
```

U ovom programskom isječku, korisnik ima slobodu da li će broj i tekst prilikom unošenja razdvojiti praznim redom (tj. pritiskom na ENTER), razmakom ili tabulatorom. U rješenju koje bi koristilo poziv funkcije “`cin.ignore()`” takva sloboda ne bi postojala. Slijedi da odluku o tome da li koristiti objekat “`ws`” ili funkciju “`cin.ignore()`” treba donijeti na osnovu toga kakvu komunikaciju između programa i korisnika želimo da podržimo. Nije na odmet napomenuti da je projektiranje *dobre i kvalitetne* komunikacije između programa i korisnika često jedna od najsloženijih (i najdosadnijih) etapa u razvoju programa, pogotovo što je, u iole ozbiljnijim primjenama, uvjek potrebno ostvariti dobru zaštitu od unosa pogrešnih podataka (jedna čuvena i često citirana “teorema” vezana za razvoj programa glasi: “Koliko god se trudili da zaštitimo program od unosa pogrešnih podataka, neka budala će naći način da pogrešni podaci

ipak uđu.”).

Sve do sada, mi smo *stringovne konstante*, odnosno skupine znakova omeđene znacima navoda, koristili isključivo zajedno sa objektom “cout”, kao drugi operand operatora “<<”. Međutim, u jeziku C++ stringovne konstante imaju mnogo šire značenje, i mogu se upotrijebiti kao stvarni parametar za bilo koju funkciju čiji je odgovarajući formalni parametar tipa *konstantni niz znakova*. Takve su, na primjer, maločas napisana funkcija “DuzinaStringa” i njoj ekvivalentna ugrađena funkcija “strlen”, tako da su sljedeće konstrukcije posve ispravne (prva će ispisati broj “12” na ekran, a druga će promjenljivo “broj_znakova” dodijeliti vrijednost 12):

```
cout << DuzinaStringa("Moj tekst...");  
int broj_znakova = strlen("Moj tekst...");
```

Bitno je napomenuti da stringovne konstante omeđene navodnicima uvijek imaju upisan “NUL” graničnik iza posljednjeg znaka, iako ga mi eksplisitno ne pišemo. Da nije tako, navedeni primjeri ne bi radili kako treba.

Stringovne konstante ne samo da se mogu upotrijebiti kao parametri funkcija koje očekuju konstantne nizove znakova, nego stringovne konstante zapravo i jesu konstantni nizovi znakova, i stoga se mogu upotrijebiti *bilo gdje* gdje bi se inače mogao upotrijebiti konstantni niz znakova. Stoga je sljedeći primjer posve korektan, i ispisuje niz znakova iz riječi “Primjer”, svaku u posebnom redu:

```
for(int i = 0; i < 7; i++) cout << "Primjer"[i] << endl;
```

Mogućnost navođenja stringovnih konstanti kao parametara funkcijama koje očekuju konstantne nizove znakova kao parametre može se jako lijepo iskoristiti. Na primjer, moguće je napisati funkciju poput sljedeće:

```
void UnesiCijeliBroj(const char prompt[], int &x) {  
    cout << prompt;  
    cin >> x;  
}
```

Ovu funkciju bismo mogli koristiti na sljedeći način:

```
int broj1, broj2;  
UnesiCijeliBroj("Prvi broj: ", broj1);  
UnesiCijeliBroj("Drugi broj: ", broj2);
```

Naravno, u ovom primjeru nema osobitog razloga da za unos broja koristimo posebnu funkciju. Puni smisao se dobija ukoliko funkciju “UnesiCijeliBroj” proširimo da obavlja i neke dodatne akcije, npr. zaštitu od unosa pogrešnih podataka, što olakšava razvoj dijela programa za komunikaciju sa korisnikom. Također, funkcija “UnesiCijeliBroj” se može proširiti da postane generička (šablonska) funkcija, tako da se istom funkcijom mogu unositi različiti tipovi podataka, a ne samo cijeli brojevi.

Kao i svaki drugi niz, nizovi znakova se također mogu inicijalizirati prilikom deklaracije. Na primjer:

```
char recenica[10] = {'D', 'o', 'b', 'a', 'r', ' ', 'd', 'a', 'n', 0};  
cout << "Rečenica glasi: " << recenica << endl;  
cout << "Njena dužina je " << strlen(recenica) << " znakova.";
```

Obratimo pažnju na “NUL” graničnik koji smo *eksplisitno* napisali iza posljednjeg znaka. Doduše, on bi bio umetnut i da smo ga zaboravili napisati, jer je poznato da u slučaju da niz ima više elemenata nego što ima članova u inicijalizacionoj listi, ostatak članova se popunjava nulama. Međutim, da smo zaboravili

da napišemo graničnik i da smo pri tome napisali da niz ima devet elemenata (ili da smo pisali samo “[]” koristeći svojstvo da se kod inicijaliziranih nizova dužina može izostaviti pri čemu se ona automatski određuje na osnovu dužine inicijalizacione liste), graničnik ne bi bio ubačen, i primjer ne bi radio kako treba. Najvjerojatnije bi se iza poruke “Dobar dan” ispisala još gomila besmislenih znakova (koji su se slučajno zatekli u memoriji iza znaka “n”) sve dok se ne bi naišlo na memorijsku lokaciju koja sadrži nulu (koja bi tada bila shvaćena kao graničnik). Iz istog razloga, funkcija “`strlen`” bi vjerovatno vratila znatno veću dužinu nego što treba.

Opisana inicijalizacija znakovnih nizova `znak po znak` je veoma nezgrapna, naročito ukoliko je tekst dugačak. Zbog toga C++ dozvoljava da se nizovi znakova inicijaliziraju *stringovnim konstantama*. Tako je sljedeći primjer potpuno ekvivalentan prethodnom, a zapisan je mnogo kompaktnije i preglednije:

```
char recenica[10] = "Dobar dan";
cout << "Rečenica glasi: " << recenica;
cout << "Njena dužina je " << strlen(recenica) << "znakova.";
```

Dužinu niza smo mogli i izostaviti, tj. mogli smo pisati samo

```
char recenica[] = "Dobar dan";
```

Prostor neophodan za pamćenje “NUL” graničnika će automatski biti predviđen, jer se podrazumijeva da stringovne konstante sadrže graničnik.

Poznato je da se, nažalost, svim elementima nizova mogu istovremeno dodijeliti vrijednosti *samo prilikom deklaracije*. Isto vrijedi i za stringove, bez obzira na način inicijalizacije. To isključuje mogućnost pisanja konstrukcija poput sljedeće, koje bi mogle biti jako korisne:

```
char recenica[100];
recenica = "Ja sam prva rečenica...";
cout << recenica << endl;
recenica = "A ja sam druga rečenica..."
cout << recenica << endl;
```

Navedena sekvenca je, nažalost, sintaksno neispravna, s obzirom da napisane dodjele predstavljaju dodjelu jednog niza drugom što, kao što znamo, nije podržano u jeziku C++ (vidjećemo kasnije da je tip “`string`”, između ostalog, uveden i da ispravi ovaj nedostatak, kao što tip “`vector`” ispravlja mnoge nedostatke klasičnih nizova). Srećom, ipak je, uz neznatno drugačiju sintaksu, moguće na relativno jednostavan način postići sličan efekat. Prepostavimo da smo napisali funkciju “`KopirajString`” sa dva parametra od kojih su oba nizovi znakova, koja kopira sve znakove drugog niza u prvi niz do “NUL” graničnika, uključujući i njega. Također prepostavimo da je drugi parametar *konstantan niz*, tako da se prilikom poziva funkcije može kao parametar proslijediti stringovna konstanta (drugi parametar smije biti konstantan niz, s obzirom da funkcija neće mijenjati sadržaj drugog, već samo prvog niza). Tada je posve legalno pisati sljedeće naredbe, koje će imati isti efekat kakav bi trebao da ima prethodni slijed naredbi koji nije dozvoljen:

```
char recenica[100];
KopirajString(recenica, "Ja sam prva rečenica...");
cout << recenica << endl;
KopirajString(recenica, "A ja sam druga rečenica...");
cout << recenica << endl;
```

Razmotrimo kako bi se mogla napisati funkcija “`KopirajString`”. Vjerovatno je najočiglednija sljedeća izvedba:

```

void KopirajString(char odredisni[], const char izvorni[]) {
    int i = 0;
    while(izvorni[i] != 0) {
        odredisni[i] = izvorni[i];
        i++;
    }
    odredisni[i] = 0;
}

```

Međutim, ista funkcija se može napisati mnogo efikasnije, na prilično neobičan način:

```

void KopirajString(char odredisni[], const char izvorni[]) {
    int i = 0;
    while(odredisni[i] = izvorni[i]) i++;
}

```

Obratimo pažnju da je unutar uvjeta petlje upotrijebljen operator “=” a ne “==”. Nije u pitanju greška: ovom konstrukcijom se *i*-ti znak niza “izvorni” kopira u *i*-ti znak niza “odredisni” i *usput* se provjerava da li je kopirani znak različit od nule (u skladu sa načinom kako se tretiraju uvjeti u C++-u). Ukoliko nije, petlja se prekida. Istu stvar možemo postići i na sljedeći način, pomoću “**for**” petlje koja je *dvostruko neobična*: tijelo joj je prazno, a u uvjetu se koristi operator “=” umjesto “==”:

```

void KopirajString(char odredisni[], const char izvorni[]) {
    for(int i = 0; odredisni[i] = izvorni[i]; i++)
}

```

Ovakvi neobični primjeri poslastica su većini autora udžbenika o C-u, a nešto rijede o C++-u. Međutim, još bolje rješenje je funkciju “KopirajString” *uopće ne pisati*, s obzirom da jezik C++ već posjeduje ugrađenu funkciju “**strcpy**” iz biblioteke “**cstring**” koja u principu radi istu stvar kao napisana funkcija “KopirajString” (koju smo pisali čisto iz edukativnih razloga). Dakle, sljedeći primjer je potpuno korektan, pod uvjetom da smo uključili biblioteku “**cstring**” u program:

```

char recenica[100];
strcpy(recenica, "Ja sam prva rečenica...");
cout << recenica << endl;
strcpy(recenica, "A ja sam druga rečenica...");
cout << recenica << endl;

```

Funkcija “**strcpy**” se često koristi za kopiranje jednog niza znakova u drugi. Tako je sljedeća konstrukcija posve legalna:

```

char s1[100], s2[50];
...
strcpy(s1, s2);

```

Jedina stvar o kojoj moramo voditi računa je da odredišni niz mora biti *dovoljno velik* da primi sve znakove izvornog niza. U suprotnom može doći do kraha programa, jer će se pristupati memoriji izvan dozvoljenog prostora (tj. prostora koji je eksplicitno rezerviran za potrebe smještanja elemenata niza).

Već je rečeno da se stringovne konstante mogu proslijediti kao parametri funkcijama pod uvjetom da je odgovarajući formalni parametar *konstantan niz znakova*. Stringovne konstante se ne bi trebale proslijediti u slučaju da je odgovarajući formalni parametar *obični (nekonstantni) niz znakova*. Na primjer, razmotrimo sljedeću funkciju koja *mijenja* sadržaj niza (preciznije, njegov prvi znak) koji je proslijeđen kao parametar:

```
void ProblematicniPrimjer(char s[]) {
    cout << s << endl;
    s[0] = 'E';
}
```

Ovakvu funkciju nikad ne bi trebalo pozivati kao u sljedećem primjeru:

```
ProblematicniPrimjer("Alma");
```

Postoji nekoliko razloga zbog čega ovo ne treba raditi. Prvo, kako je stringovna konstanta "Alma" po svojoj prirodi *konstanta*, kompjajler ima puno pravo da je smjesti u dio memorije koji dozvoljava *čitanje*, ali ne i *upis* odnosno *izmjenu*. U tom slučaju, naredba dodjele "s[0] = 'E'" će pokušati izvršiti upis u područje memorije koje je zaštićeno od upisa, i program će se vjerovatno "srušiti". Čak i ukoliko ova dodjela "prode" (što zavisi kako od kompjajlera, tako i od operativnog sistema na kojem se program izvršava), njen efekat će biti *izmjena sadržaja* nečega što bi trebalo da bude nepromjenljivo, što na kraju može imati posve neočekivan efekat. Na primjer, suprotno očekivanjima, naredba

```
for(int i = 1; i <= 3; i++) ProblematicniPrimjer("Alma");
```

ne bi tri puta ispisala tekst "Alma" na ekranu, već jedanput tekst "Alma" a dva puta tekst "Elma", jer bi nakon prvog poziva funkcije "ProblematicniPrimjer" sadržaj "konstante" "Alma" bio promijenjen! Mogući su i još gori scenariji. Na primjer, pogledajmo sljedeću sekvencu naredbi:

```
ProblematicniPrimjer("Alma");
cout << "Alma" << endl;
```

Ovaj primjer, čak i ukoliko se program ne "sruši", na većini kompjajlera doveće do toga da će naredba ispisana nakon poziva funkcije ispisati tekst "Elma". Ovo je potpuno neočekivano, jer ova naredba eksplicitno ispisuje tekst "Alma". Međutim, većina kompjajlera višestruke pojave iste stringovne konstante (u ovom primjeru "Alma") radi uštade memorije *čuva na jednom te istom mjestu u memoriji*, što je razumna odluka, s obzirom da je njihov sadržaj nepromjenljiv (odnosno, takav bi barem *trebao da bude*). Ovo svojstvo naziva se *stapanje stringova* (engl. *string merging*). Stoga, ukoliko poziv funkcije "ProblematicniPrimjer" izmijeni sadržaj stringovne konstante, ta izmjena će se odraziti i na sve druge pojave iste stringovne konstante u programu. Ovo bi svakako moglo da bude veoma zbumujuće.

Da bi se spriječile gore opisane konfuzne situacije, novi standard jezika C++ strogo propisuje da je *zabranjeno* prosljedivati stringovne konstante funkcijama ukoliko odgovarajući formalni parametri nisu deklarirani kao konstantni, i kompjajler *ne bi trebao da dozvoli* ovakve pozive. Preciznije, standard precizira da sve stringovne konstante imaju tip "**const char []**", i kao takve, ne mogu se prosljedivati u funkcije ukoliko odgovarajući formalni parametar nije deklariran kao konstantan. Nažalost, veliki broj kompjajlera (gotovo svi) *toleriraju* ovakve pozive, uz nepredvidljive posljedice. Naime, po standardu jezika C, kao i po ranijim standardima jezika C++, tip stringovnih konstanti definiran je kao "**char []**" (a ne kao "**const char []**"), tako da nema ograničenja na konstantnost odgovarajućih formalnih parametara prilikom prenosa u funkcije. Stoga su mnogi kompjajleri za C++, vjerovatno plašeći se gubitka kompatibilnosti sa jezikom C, i dalje zadržali ovu konvenciju (kod nekih kompjajlera može se vršiti izbor između stare i nove konvencije), bez obzira što je sama ideja "upisa" u stringovne konstante veoma loša, tako da je ne bi trebao koristiti niti jedan "normalan" program (uključujući i programe pisane u jeziku C). To ne vrijedi samo za korisnički napisane funkcije, nego i za ugrađene funkcije, kao što je "strcpy". Na primjer, sljedeća konstrukcija je po novom C++ standardu nedozvoljena (i upitno je šta bi se njom uopće trebalo da postigne), tako da kompjajler *ne bi trebao da je dopusti*:

```
strcpy("abc", "def");
```

Međutim, ova konstrukcija će “proći” na većini kompjlera (u boljem slučaju uz upozorenje), uz posve nepredvidljive posljedice koje variraju od mogućeg kraha programa pa do zamjene sadržaja stringovne konstante "abc" stringovnom konstantom "def" koja će se odraziti svugdje gdje je stringovna konstanta "abc" upotrijebljena u programu. Mada nekome ovakav efekat može djelovati kao “zgodan trik”, njegova upotreba je, čak i u slučaju da je konkretan kompjaler i operativni sistem dozvoljavaju, *izuzetno loša taktika* koju ne bi trebalo koristiti ni po koju cijenu (činjenica da je po standardu jezika C ovakav poziv načelno *dozvoljen* ne znači da ga treba ikada koristiti, čak i ukoliko programiramo striktno u jeziku C). Još gore posljedice može imati naredba

```
strcpy("abc", "defghijk");
```

U ovom slučaju se duža stringovna konstanta "defghijk" pokušava prepisati u dio memorije u kojem se čuva kraća stringovna konstanta "abc", što ne samo da će prebrisati ovu stringovnu konstantu, nego će i prebrisati sadržaj nekog dijela memorije za koji ne znamo kome pripada (za stringovnu konstantu "abc" rezervirano je tačno onoliko prostora koliko ona zauzima, tako da već prostor koji slijedi može pripadati nekom drugom). Naravno, posljedice su potpuno nepredvidljive i nikada nisu bezazlene. Kao zaključak, prvi argument funkcije “strcpy” smije biti isključivo niz za koji eksplisitno znamo da mu se sadržaj smije mijenjati, i koji je dovoljno velik da primi sadržaj niza koji se u njega kopira.

Funkcija “strcat” iz biblioteke “cstring” slična je funkciji “strcpy”, ali ne kopira drugi parametar u prvi, nego *nadovezuje* drugi parametar na kraj stringa predstavljenog prvim parametrom. Stoga će sljedeći primjer ispisati rečenicu “Dobar dan!”:

```
char recenica[100];
strcpy(recenica, "Dobar ");
strcat(recenica, "dan!");
cout << recenica << endl;
```

Česta greška kod početnika je pisanje konstrukcija poput

```
strcat("Dobar ", "dan!");
```

Ova greška je naročito česta kod korisnika koji poznaju neke druge programske jezike, koje posjeduju funkcije sa kojima se može ovako raditi. Međutim to nije slučaj sa funkcijom “strcat”: ovaj poziv je neispravan zbog istog razloga kao u slučaju funkcije “strcpy”. Naime, ovaj poziv bi pokušao da nadoveže stringovnu konstantu "dan!" na kraj onog dijela memorije gdje se čuva stringovna konstanta "Dobar ", čime ne samo da bi promijenio sadržaj ove stringovne konstante, nego bi i prebrisao sadržaj memorijskih lokacija koje joj ne pripadaju. Stoga se može zaključiti da funkciju “strcat” ima smisla koristiti samo ukoliko je njen prvi argument niz koji već sadrži neki string (eventualno prazan) i koji je dovoljno veliki da može prihvati string koji će nastati nadovezivanjem. Možemo istu stvar reći i ovako: korisnici koji poznaju neke druge programske jezike mogu pomisliti da funkcija “strcat” vraća *reultat* string koji je nastao nadovezivanjem prvog i drugog argumenta (ne mijenjajući svoje argumente), s obzirom da takve funkcije postoje u mnogim programskim jezicima. Međutim, to nije slučaj sa funkcijom “strcat” iz jezika C++: ona *nadovezuje drugi argument na prvi*, pri tome mijenjajući prvi argument. Pri tome, ne treba pogrešno pomisliti da funkcija “strcat” ne vraća nikakav rezultat, odnosno da je ona “**void**” funkcija. Kakav rezultat ova funkcija vraća vidjećemo kasnije (radi se o adresi prvog znaka prvog parametra), a za sada ćemo njenu povratnu vrijednost (tj. rezultat) prosto ignorirati. U neku ruku, razlika između funkcije “strcat” i srodnih funkcija u drugim programskim jezicima najlakše se može uporediti sa razlikom koja postoji između izraza “x += y” i “x + y”.

Sljedeća stvar o kojoj trebamo govoriti je problem *poređenja stringova*. Zamislimo da je potrebno

napraviti program koji će tražiti od korisnika da unese neku lozinku. U slučaju da lozinka nije ispravna, od korisnika treba da se traži novi unos, sve dok lozinka ne bude ispravna. Pretpostavimo, na primjer, da lozinka glasi "HRKLJUŠ". Prirodan pokušaj bi bio da probamo napisati programski isječak poput sljedećeg:

```
char lozinka[100];
do {
    cout << "Unesi lozinku:";
    cin.getline(lozinka, sizeof lozinka);
    if(lozinka != "HRKLJUŠ") cout << "Pogrešna lozinka!";
} while(lozinka != "HRKLJUŠ");
```

Mada će ovaj primjer proći sintaksno, on neće raditi u skladu sa očekivanjima. Naime, kako su stringovi *nizovi*, na njih se ne mogu normalno primjenjivati operatori poređenja "<", ">". "<=", ">=", "==" i "!=", nego se poređenja moraju vršiti element po element. Razlog zbog kojeg ovaj primjer prolazi sintaksno leži u činjenici da se svi nizovi upotrijebljeni samostalno bez operatora indeksiranja automatski konvertiraju u odgovarajuće *adrese* na kojima su smješteni u memoriji (na ovu činjenicu smo ukazivali na nekoliko mesta), tako da navedeni primjer zapravo poredi adrese a ne same nizove, što sigurno nije ono što nam treba. Stoga unesenu lozinku moramo porediti *znak po znak* sa željenom lozinkom da bismo ispitali jednakost. Na primjer, moglo bi se raditi ovako:

```
char lozinka[100];
do {
    cout << "Unesi lozinku:";
    cin.getline(lozinka, sizeof lozinka);
    bool lozinka_je_dobra(true);
    for(int i = 0; i < 7; i++)
        if(lozinka[i] != "HRKLJUŠ"[i]) lozinka_je_dobra = false;
    if(!lozinka_je_dobra) cout << "Pogrešna lozinka!";
} while(!lozinka_je_dobra);
```

Ne bi bio nikakav problem napisati funkciju koja bi za dva stringa proslijedena kao parametri ispitala da li su jednaki ili nisu (i vratila odgovor kao rezultat), što čitatelj ili čitateljka mogu uraditi kao korisnu vježbu. Medutim, takva funkcija već postoji ugradena u biblioteku "cstring" pod imenom "strcmp". Ova funkcija vraća kao rezultat *nulu* ukoliko su dva stringa proslijedena kao parametri jednaki, a u suprotnom vraća neki broj *različit od nule* (ne nužno jedinicu). Stoga bismo prethodni primjer mogli jednostavnije napisati ovako:

```
char lozinka[100];
do {
    cout << "Unesi lozinku:";
    cin.getline(lozinka, sizeof lozinka);
    if(strcmp(lozinka, "HRKLJUŠ") != 0) cout << "Pogrešna lozinka!";
} while(strcmp(lozinka, "HRKLJUŠ") != 0);
```

Obratimo pažnju da se uvjet " $\neq 0$ " može izostaviti s obzirom na to kako funkcioniraju uvjeti u jeziku C++. Bitno je da za jednakost stringova svi znaci moraju biti *doslovno jednaki*, tj. moraju se podudarati kako velika i mala slova. Stoga, stringovi "HRKLJUŠ" i "Hrkljuš" nisu jednakci. Također, dužine stringova moraju biti jednakе, tako da stringovi "HRKLJUŠ" i "HRKLJUŠ " nisu jednakci.

Funkcija "strcmp" radi znatno više od običnog testiranja stringova na jednakost. U slučaju da je prvi string *leksikografski* (tj. po *abecednom poretku*) ispred *drugog stringa*, funkcija "strcmp" vraća kao rezultat neki broj *manji od nule* (standard ne predviđa koji, ali vidjećemo da zbog načina kako se funkcija "strcmp" obično koristi, to nije ni bitno). U slučaju da je prvi string *leksikografski iza drugog stringa*,

funkcija “strcmp” vraća kao rezultat neki broj *veći od nule*. Ovo je najbolje ilustrirati na konkretnom primjeru:

```
const int MaxDuzina(50);
char rijec_1[MaxDuzina], rijec_2[MaxDuzina];
cout << "Unesi dvije riječi: ";
cin >> setw(MaxDuzina) >> rijec_1 >> setw(MaxDuzina) >> rijec_2;
if(!strcmp(rijec_1, rijec_2)) cout << "Riječi su jednake\n";
else {
    cout << "Riječ " << rijec_1 << " je po abecedi ";
    if(strcmp(rijec_1, rijec_2) < 0) cout << "ispred"
    else cout << "iza";
    cout << " riječi " << rijec_2 << endl;
}
```

Funkcija “strcmp” zapravo ne vrši poređenje tačno po abecednom poretku, nego po poretku ASCII šifri, a abecedni poredak je prosto posljedica činjenice da ASCII šifre rastu po abecednom poretku. Međutim, po ASCII tablici sva velika slova dolaze ispred malih slova, tako da se string "Damir" nalazi po poretku ispred stringa "DAMIR". Što je još gore, string "Drvo" nalazi se (zbog velikog slova "D") po poretku ispred stringa "cipela". Iz istog razloga, svi stringovi koji počinju ciframa nalaze se po poretku ispred stringova koji počinju slovima. Neke verzije kompjlera za C++ poznaju i funkciju “stricmp” koja je potpuno analogna funkciji “strcmp”, ali ne pravi nikakvu razliku između velikih i malih slova. Ipak, ova funkcija nije standardna u jeziku C++. Stoga, ukoliko želimo da poredimo stringove po abecedi ne praveći razliku između malih i velikih slova, najbolje je prije poređenja sva mala slova u stringovima pretvoriti u velika, ili obrnuto. To možemo lako učiniti koristeći aritmetiku nad znakovnim vrijednostima, ili korištenjem funkcije “toupper” iz biblioteke “ctype”). Na primjer, sljedeća naredba će pretvoriti sva mala slova u stringovnoj promjenljivoj “s” u velika:

```
for(int i = 0; i < strlen(s); i++) s[i] = toupper(s[i]);
```

Još jedan problem koji se javlja prilikom upotrebe funkcije “strcmp” na našim područjima je što ova funkcija ne zna ispravno porediti stringove u kojima se javljaju naša slova poput “ć”, “š” itd. s obzirom da oni nemaju jedinstvene šifre po ASCII standardu. Ovaj problem se sve do nedavno rješavao na razne ne baš jednostavne načine. Tek u novijim standardima jezika C++ uvedene su i funkcije koje rade slično poput funkcije “strcmp”, ali koje vode računa o regionalnim specifičnostima poput znakova koji se javljaju u nacionalnim (neengleskim) jezicima u skladu sa regionalnim postavkama računara na kojem se program izvršava. Ove funkcije su nešto komplikovanije za upotrebu i zahtijevaju izvjesne predradnje, tako da o njima ovdje neće biti govora. Zainteresirani čitatelji i čitateljke upućuju se na literaturu koja razmatra objekte i funkcije iz biblioteke nazvane “locale”.

Pored opisanih funkcija “strlen”, “strcpy”, “strcat” i “strcmp”, standardna biblioteka “cstring” posjeduje još oko dvadesetak funkcija za rad sa klasičnim nul-terminaliziranim stringovima, uključujući funkcije za izdvajanje pojedinačnih dijelova stringa, pronalaženje pozicije na kojoj se neki tekst eventualno nalazi unutar stringa, itd. Ove funkcije nećemo ovdje opisivati iz dva razloga. Prvo, njihova upotreba uglavnom zahtijeva poznavanje rada sa pokazivačima, sa kojima se još nismo upoznali. Neke od tih funkcija biće ukratko objašnjene u poglavljju koje govori o pokazivačima. Drugo, rad sa svim tim funkcijama, kao i uostalom kompletan rad sa nul-terminaliziranim stringovima, dosta je nezgrapan. Stoga se za iole složeniju manipulaciju sa stringovima u jeziku C++ umjesto klasičnih nul-terminaliziranih stringova preporučuje korištenje tipa “string”, koji će ukratko biti opisan u ostatku ovog poglavlja. Ovaj tip i operacije sa njim definirani su u istoimenoj biblioteci “string” (ne “cstring”). Stoga svi primjeri koji slijede podrazumijevaju da smo uključili zaglavlje ove biblioteke u program.

Promjenljive tipa "string", kojemožemo zvati *dinamički stringovi*, deklariraju se na uobičajeni način, kao i svaka druga promjenljiva (naravno, "string" nije ključna, već predefinirana riječ). Za razliku od običnih nizova znakova, pri deklaraciji promjenljivih tipa "string" ne navodi se maksimalna dužina stringa, s obzirom da se njihova veličina, zahvaljujući postupcima tzv. *dinamičke alokacije memorije* (sa kojima ćemo se upoznati kasnije) automatski prilagodava tokom rada. Promjenljive tipa "string", ukoliko se eksplisitno ne inicijaliziraju, automatski se inicijaliziraju na *prazan string* (tj. string koji ne sadrži niti jedan znak, odnosno string dužine 0), Dakle, deklaracija

```
string s;
```

deklarira promjenljivu "s" tipa "string" koja je automatski inicijalizirana na prazan string. S druge strane, promjenljive tipa "string" mogu se inicijalizirati bilo nizom znakova (što uključuje i stringovne konstante), bilo drugom stringovnom promjenljivom, bilo proizvoljnim stringovnim izrazom, odnosno izrazom čiji je rezultat tipa "string" (uskoro ćemo vidjeti kako se ovakvi izrazi mogu formirati). Pri tome se može koristiti bilo sintaksa sa znakom dodjele "=", bilo sintaksa koja koristi zagrade. Drugim riječima, obje deklaracije koje slijede su legalne i ekvivalentne:

```
string s = "Ja sam string";
string s("Ja sam string");
```

Također, uz prepostavku da je "recenica" neki niz znakova (tj. promjenljiva tipa "**char** []" ili "**const char** []"), legalne su i sljedeće deklaracije:

```
string s = recenica;
string s(recenica);
```

Za razliku od običnih znakovnih nizova, koji se također mogu inicijalizirati stringovnim konstantama (ali ne i drugim stringovnim nizovima, i to samo uz upotrebu sintakse sa znakom dodjele), promjenljivim tipa "string" se u bilo koje vrijeme (za vrijeme trajanja njihovog života) može pomoći znaka "=" *dodijeliti* drugi niz znakova (uključujući naravno i stringovne konstante), druga dinamička stringovna promjenljiva, ili čak proizvoljan stringovni izraz. Ovim je omogućeno da se ne moramo patiti sa funkcijom "strcpy" i njoj srodnim funkcijama, već možemo prosto pisati konstrukcije poput

```
string s = "Ja sam string!";
cout << s << endl;
s = "A sada sam neki drugi string...";
cout << s << endl;
```

Naravno, obrnuta dodjela (tj. dodjela stringovne promjenljive ili stringovnog izraza običnom nizu znakova) nije moguća, u skladu sa činjenicom da se nizovima ne može ništa dodjeljivati, odnosno nizovi se ne mogu naći sa lijeve strane operatora dodjele (bitno je prihvatićti činjenicu da promjenljive tipa "string" *nisu nizovi znakova*, nego promjenljive sasvim posebnog tipa, mada interno u sebi *čuvaju* nizove znakova).

Iz navedenog primjera se jasno vidi i da je dinamičke stringove također moguće ispisivati na izlazni tok putem operatora "<<", što je vjerovatno u skladu sa očekivanjima. Neće biti veliko iznenađenje ukoliko kažemo da se dinamički stringovi mogu čitati iz ulaznog toka pomoći operatora ">>", pri čemu ovaj put ne moramo voditi računa o maksimalno dozvoljenoj dužini (naravno, izdvajanje iz ulaznog toka se prekida na prvoj praznini, u skladu sa uobičajenim ponašanjem operatora ">>"). Međutim, ukoliko želimo u promjenljivu tipa "string" pročitati

čitavu liniju ulaznog toka (sve do oznake kraja reda), sintaksa se neznatno razlikuje u odnosu na slučaj kada koristimo obične nizove znakova. Naime, s obzirom da ovaj put ne moramo voditi računa o maksimalnom broju znakova koji se čitaju, prirodno bi bilo očekivati da će se za čitanje čitave linije ulaznog toka u promjenljivu “*s*” tipa “*string*” vršiti konstrukcijom poput

```
cin.getline(s);
```

Umjesto toga, koristi se neznatno izmijenjena sintaksa:

```
getline(cin, s);
```

Razlozi za ovu nedoslijednost su čisto tehničke prirode. Naime, da bi se podržala prva sintaksa, bilo bi neophodno izmijeniti strukturu biblioteke “*iostream*” na način koji bi ovu biblioteku učinio ovisnom o biblioteci “*string*”. Tvorci jezika C++ nisu željeli uvoditi nepotrebne međuovisnosti između biblioteka, te su podržali drugu sintaksu, koja nije tražila uvođenje takve međuovisnosti.

Za pristup pojedinim znakovima unutar promjenljivih tipa “*string*” koristi se indeksiranje pomoću uglastih zagrada “[]” na *isti način kao da se radi o klasičnim nizovima znakova* (ali, ponavljamo, one *nisu nizovi znakova*). Stvarnu dužinu promjenljive tipa “*string*” možemo saznati primjenom operacije “*size*”, na isti način kao i u slučaju promjenljivih tipa “*vector*”. Umjesto naziva operacije “*size*”, za promjenljive tipa “*string*” može se ravnopravno koristiti (sa istim značenjem) i operacija “*length*”. Tako, sljedeći program učitava rečenicu sa tastature i ispisuje je naopačke:

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

int main() {
    string recenica;
    cout << "Unesi rečenicu: ";
    getline(cin, recenica);
    cout << "Rečenica izgovorena naopako glasi: ";
    for(int i = recenica.length() - 1; i >= 0; i--) cout << recenica[i];
    return 0;
}
```

Klasični nul-terminirani stringovi su po prirodi nizovi znakova, pa se kao i svi drugi nizovi mogu prenositi kao parametri u funkcije *isključivo po referenci* (odnosno, barem tako izgleda), i *ne mogu se vraćati kao rezultat iz funkcije*. Ova ograničenja ne vrijede za dinamičke stringove. Rezultat funkcije sasvim legalno može biti tipa “*string*”, a parametri tipa “*string*” mogu se prenositi u funkcije bilo po vrijednosti, bilo po referenci. Ipak, prenos po

vrijednosti se *ne preporučuje*, s obzirom da tada uvijek dolazi do *kopiranja* stvarnih parametara u formalne, a objekti tipa “string” spadaju u masivne objekte čije kopiranje nije uvijek efikasno. Stoga se preporučuje sve parametre tipa “string” prenositi *po referenci* i to, kad je god moguće, po *referenci na konstantu*, što omogućava da se kao stvarni parametri mogu koristiti i stringovni izrazi. Ovo je demonstrirano na primjeru sljedeće funkcije nazvane “Sastavi”, koja *vraća kao rezultat* string koji nastaje sastavljanjem dva stringa koji su proslijedeni kao parametri:

```
string Sastavi(const string &s1, const string &s2) {
    int duzina1 = s1.length(), duzina2 = s2.length();
    string s3;
    s3.resize(duzina1 + duzina2);
    for(int i = 0; i < duzina1; i++) s3[i] = s1[i];
    for(int i = 0; i < duzina2; i++) s3[i + duzina1] = s2[i];
    return s3;
}
```

U ovoj funkciji iskorištena je operacija “*resize*”, koja se koristi na isti način kao u slučaju promjenljivih tipa “vector”, kojom se osigurava da će promjenljiva “s3” imati dužinu dovoljnu da prihvati sve znakove sadržane u oba stringa koja treba sastaviti (formirani prostor inicijalno se popunjava razmacima). Princip rada ove funkcije dovoljno je jasan. Primijetimo da se ova funkcija osjetno razlikuje od funkcije “*strcat*” iz biblioteke “cstring” po tome što *kreira novi dinamički string i vraća ga kao rezultat*, bez modifikacije svojih stvarnih parametara. Tako se, uz pretpostavku da su “str1”, “str2” i “str3” tri promjenljive tipa “string”, sljedeća konstrukcija je sasvim korektna:

```
str3 = Sastavi(str1, str2);
```

U ovom primjeru, konstrukcija “Sastavi(str1, str2)” predstavlja jednostavan primjer *stringovnog izraza*. Međutim, interesantno je da su posve legalne i sljedeće konstrukcije (uz pretpostavku da je “znakovi_1” i “znakovi_2” neke promjenljive tipa klasičnog niza znakova):

```
str3 = Sastavi(str1, Sastavi(str2, str3));
str3 = Sastavi(str1, "bla bla");
str3 = Sastavi("bla bla", str1);
str3 = Sastavi("Dobar ", "dan!");
str3 = Sastavi(znakovi_1, str1);
str3 = Sastavi(znakovi_1, "bla bla");
str3 = Sastavi(znakovi_1, znakovi_2);
```

Prva konstrukcija je jasna: stvarni parametar može biti i znakovni izraz. Međutim, sve ostale konstrukcije su na prvi pogled neobične, jer stvarni parametri u njima nisu tipa “string”. Sve

je ovo najlakše objasniti ako prihvatimo da postoji automatska konverzija iz tipa "niz znakova" u tip "string" (kao što postoje automatske konverzije iz tipa "int" u tip "double" itd.). Stoga, kad god je formalni parametar tipa "string" (ne referenca na tip "string") ili referenca na konstantni string (odnosno tipa "const string &"), kao stvarni parametar je moguće upotrijebiti *klasični niz znakova*. Ovo ne vrijedi kada je formalni parametar obična referenca (na ne-konstantu), jer se reference na ne-konstante nikada ne mogu vezati za objekat koji nije striktno istog tipa. Obrnuta automatska konverzija (tj. konverzija iz tipa "string" u tip "niz znakova") *ne postoji*, tako da funkcije koje kao formalni parametar imaju klasične nizove znakova *neće prihvatiti* kao stvarne parametre promjenljive i izraze tipa "string".

Gore pomenuta funkcija "Sastavi" napisana je samo iz edukativnih razloga, s obzirom da se identičan efekat može se ostvariti primjenom operatora "+". Na primjer, neke od gore napisanih konstrukcija moguće su se napisati na sljedeći, mnogo pregledniji način (i to bez potrebe za pisanjem posebne funkcije):

```
str3 = str1 + str2 + str3;  
str3 = str1 + "bla bla";  
str3 = "bla bla" + str1;  
str3 = znakovi_1 + str1;
```

U posljednje tri konstrukcije dolazi do automatske pretvorbe operanda koji je tipa "niz znakova" u tip "string". Ipak, za primjenu operatora "+" *barem jedan od operanada* mora biti dinamički string. Stoga, sljedeće konstrukcije *nisu legalne*:

```
str3 = "Dobar " + "dan!";  
str3 = znakovi_1 + "bla bla";  
str3 = znakovi_1 + znakovi_2;
```

Naravno, kao i u mnogim drugim sličnim situacijama, problem je rješiv uz pomoć *eksplicitne pretvorbe tipa*, na primjer, kao u sljedećim konstrukcijama (moguće su i brojne druge varijante):

```
str3 = znakovi_1 + string("bla bla");  
str3 = (string)znakovi_1 + znakovi_2;
```

Vidimo da je rad na opisani način mnogo elegantniji nego korištenje rogobatnih funkcija "strcpy" i "strcat". Definiran je i operator "+=", pri čemu je, kao što je uobičajeno, izraz "s1 += s2" načelno ekvivalentan izrazu "s1 = s1 + s2". Na primjer:

```
string s = "Principi ";  
s += "programiranja";
```

```
cout << s;
```

Veoma je korisna i operacija “`substr`”. Ova operacija daje kao rezultat dinamički string koji je izdvojen kao dio iz stringovne promjenljive na koju je primijenjen, a koristi se sa dva cjelobrojna parametra: *indeks od kojeg počinje izdvajanje i broj znakova koji se izdvajaju*. Na primjer, sljedeća sekvenca naredbi ispisaće tekst “Ovdje nešto fali...” na ekran:

```
string s = "Ovdje fali...";  
cout << s.substr(0, 5) + " nešto" + s.substr(5, 8);
```

Funkcija “`strcmp`” je također nepotrebna pri radu sa promjenljivim tipa “`string`”, jer sa njima relacione operacije “`==`”, “`!=`”, “`<`”, “`>`”, “`<=`” i “`>=`” rade posve u skladu sa očekivanjima, odnosno vrše usporedbu po abecednom kriteriju (uz pretpostavku da je barem jedan operand dinamički string). Ovim konstrukcije poput sljedeće rade u skladu sa očekivanjima:

```
string lozinka;  
cout << "Unesi lozinku: ";  
getline(cin, lozinka);  
if(lozinka != "HRKLJUŠ") cout << "Neispravna lozinka!\n";
```

Postoji još čitavo mnoštvo korisnih operacija definiranih nad dinamičkim stringovima, koje ovdje nećemo opisivati, s obzirom da bi njihov opis odnio isuviše prostora. Zainteresirani čitatelji i čitateljke upućuju se na literaturu koja razmatra biblioteku “`string`”. Međutim, već su i do sada razmotrene operacije sasvim dovoljne za prilično udoban rad sa tekstualnim podacima. U poglavljima koja slijede, tip “`string`” ćemo koristiti uglavnom samo u slučajevima kada nam bude potrebna puna fleksibilnost koju ovaj tip omogućava. U jednostavnijim slučajevima, zadovoljićemo se klasičnim nul-terminiranim stringovima.

Za kraj, ostaje da razmotrimo još jedan detalj. Ponekad je potrebno promjenljivu ili izraz tipa “`string`” proslijediti funkciji koja kao parametar očekuje klasični (nul-terminirani) konstantni niz znakova. Nažalost, već smo rekli da automatska konverzija iz tipa “`string`” u tip znakovnog niza nije podržana (što je učinjeno namjerno, da se spriječe mogući previdi koji bi mogli rezultirati iz takve konverzije). Međutim, postoji operacija “`c_str`” (bez parametara) kojom se ovakva konverzija može zatražiti *eksplicitno*. Ova operacija, primijenjena na promjenljivu tipa “`string`”, vrši njenu konverziju u *konstantni nul-terminirani niz znakova*, odnosno u tip “`const char []`”, koji se može proslijediti funkciji koja takav tip parametara očekuje. Na primjer, ukoliko je potrebno promjenljivu “`s`” tipa “`string`” iskopirati u niz znakova “`znakovi`”, to najlakše možemo učiniti na sljedeći način:

```
strcpy(znakovi, s.c_str());
```

Ovdje operacija “`c_str`” vrši odgovarajuću pretvorbu sa ciljem pretvorbe promjenljive “`s`” u tip kakav funkcija “`strcpy`” očekuje, a ostatak posla obavlja upravo ova funkcija.

23. Višedimenzionalni nizovi

Obični (jednodimenzionalni) nizovi pogodni su za predstavljanje podataka koji se mogu urediti u neki slijed, kao što su npr. troškovi neke fabrike po mjesecima za jednu godinu. Međutim, postoje situacije u kojima jednodimenzionalni nizovi nisu pogodni za opis podataka koje želimo razmatrati. Na primjer, pomoću jednodimenzionalnih nizova je lako zapamtiti ocjene iz *svih* predmeta za *jednog* studenta, ili ocjene iz *jednog* predmeta za *sve* studente na godini. Ovi podaci mogu se zamisliti kao jedan red ili jedna kolona tabele, čiji redovi predstavljaju studente, a kolone predmete. Šta ukoliko je, međutim, potrebno zapamtiti podatke o ocjenama iz *svih* predmeta za *sve* studente (tj. ukoliko je potrebno zapamtiti čitavu tabelu)? Da bi se riješili ovakvi problemi, C++ dozvoljava da elementi nizova budu bilo kojeg tipa, *uključujući i same nizove*. To nam omogućava da kreiramo *višedimenzionalne nizove* kao nizove nizova. Na primjer, pretpostavimo da imamo sljedeću deklaraciju:

```
typedef int Red[5];
```

Ovom deklaracijom smo definirali novi *tip* “Red”, koji predstavlja *niz od najviše 5 cijelih brojeva* (uočimo da smo pri tome samo definirali novi tip, ali ne i konkretna primjerak neke promjenljive tog tipa, nad kojoj bismo mogli obavljati neke operacije). Ovakva deklaracija nam omogućava da naknadno definiramo konkretnе primjerke promjenljivih tipa “Red”, na primjer:

```
Red r1, r2;
```

Na ovaj način smo definirali dvije konkretnе promjenljive “r1” i “r2” od kojih svaka predstavlja konkretnе nizove od najviše 5 elemenata, nad kojima se mogu obavljati operacije. Međutim, posve je legalno definirati *niz* čiji će svaki element biti tipa “Red” (isto kao što je legalno definirati niz čiji će svaki element biti tipa “int”). Dakle, sljedeća deklaracija je posve legalna:

```
Red tabela[4];
```

Ovim smo definirali promjenljivu “tabela” koja predstavlja *niz od najviše 4 elementa*, koje možemo nazvati “tabela[0]”, “tabela[1]”, “tabela[2]” i “tabela[3]”. Međutim, svaki od ovih elemenata je tipa “Red”, koji predstavlja *niz od najviše 5 cijelih brojeva*. Stoga promjenljivu “tabela” možemo posmatrati kao *dvodimenzionalnu tabelu* koja se sastoji od maksimalno 4 reda i 5 kolona (s obzirom da svaki red ima do 5 elemenata). Kako su redovi tabele (odnosno elementi niza “tabela”) sami po sebi jednodimenzionalni nizovi (tipa “Red”), to pojedinačnim elementima svakog reda možemo pristupiti navođenjem još jednog indeksa. Na primjer, trećem redu tabele “tabela” možemo pristupiti pomoću izraza “tabela[2]” (s obzirom da indeksi počinju od nule), dok pojedinačnim elementima trećeg reda tabele “tabela” možemo pristupiti respektivno pomoću izraza “tabela[2][0]”, “tabela[2][1]”, “tabela[2][2]” i “tabela[2][3]”. Ovo je slikovito prikazano na sljedećoj slici, na kojoj je red “tabela[2]” osjenčen, a pojedinačni element “tabela[2][1]” dvostruko osjenčen:

```
tabela
```

Niz definiran na ovaj način, tj. kao niz običnih nizova, naziva se *dvodimenzionalni niz*, a često se susreće i izraz *matrica*, zbog analogne strukture podataka u matematici. Očigledno, dvodimenzionalni nizovi su naročito pogodni za predstavljanje podataka koji se mogu organizirati kao tabele sa više redova i više kolona. Stoga, C++ omogućava da se dvodimenzionalni nizovi definiraju i *neposredno*, bez prethodnog definiranja posebnog nizovnog tipa koji predstavlja pojedinačne redove tabele. Za tu svrhu, iza navođenja imena promjenljive potrebno je navesti *dva para uglastih zagrada* i u njima odgovarajuće dimenzije (koje, slično kao kod jednodimenzionalnih nizova, moraju biti *prave konstante*, odnosno konstante čija je tačna vrijednost poznata u trenutku provođenja programa). Na primjer, promjenljivu "tabela" mogli smo neposredno deklarirati na sljedeći način:

```
int tabela[4][5];
```

Smisao ove deklaracije je isti kao i smisao prethodne deklaracije u kojoj smo prvo definirali tip "Red", a zatim ga iskoristili za deklariranje promjenljive "tabela". Naime, ovom deklaracijom "tabela" postaje niz od najviše 4 elementa u kojem je svaki element niz od najviše 5 cijelih brojeva, što možemo tumačiti i kao tabelu sa najviše 4 reda i najviše 5 kolona. Bez obzira na način deklaracije, svaki red dvodimenzionalnog niza, sam za sebe se ponaša kao običan jednodimenzionalni niz, i kao takav se može koristiti svugdje gdje je dozvoljena upotreba jednodimenzionalnih nizova.

Rad sa dvodimenzionalnim nizovima ćemo prvo ilustrirati na jednom jednostavnom primjeru. Neka je potrebno napraviti program koji traži da se unesu ocjene za grupu studenata iz nekoliko predmeta, a koji zatim računa i ispisuje prosječnu ocjenu za svakog studenta posebno. Za tu svrhu definiran je dvodimenzionalni niz "ocjene" koji čuva tabelu ocjena studenata, i čiji redovi predstavljaju studente, a kolone predmete. Program demonstrira tipičan način manipulacije sa dvodimenzionalnim nizovima, u kojem se koriste ugnježdene "for" petlje (tj. jedna **for** petlja unutar druge). Vanjska petlja broji redove niza, a unutrašnja petlja elemente unutar trenutno razmatranog reda:

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

using namespace std;

const int MaxBrojStudenata(50), MaxBrojPredmeta(10);

int main() {
    int broj_studenata, broj_predmeta;
    cout << "Koliko ima studenata: ";
    cin >> broj_studenata;
    cout << "Koliko ima predmeta: ";
    cin >> broj_predmeta;
    int ocjene[MaxBrojStudenata][MaxBrojPredmeta];
    for(int student = 0; student < broj_studenata; student++) {
        cout << "Unesi ocjene za studenta " << student + 1 << ":\n";
        for(int predmet = 0 ; predmet < broj_predmeta; predmet++) {
            cout << " Predmet " << predmet + 1 << " : ";
            cin >> ocjene[student][predmet];
        }
        cout << endl;
    }
    cout << "Broj studenta      Prosječna ocjena\n";
    cout << "-----      ----- \n";
    for(int student = 0; student < broj_studenata; student++) {
```

```

double suma(0);
for(int predmet = 0; predmet < broj_predmeta; predmet++)
    suma += ocjene[student][predmet];
cout << setw(8) << student + 1 << setw(20) << setprecision(2)
    << suma / broj_studenata << endl;
}
return 0;
}

```

Problem što se dimenzije niza moraju znati unaprijed (s obzirom da moraju biti konstantne) rješava se slično kao kod jednodimenzionalnih nizova tako što se predvide dimenzije koje su veće od najvećih dimenzija koje se mogu očekivati prilikom korištenja programa (u slučaju da je potrebna veća fleksibilnost, moguća su rješenja zasnovana na upotrebi tipa “vector” ili na dinamičkoj alokaciji memorije, o čemu ćemo kasnije govoriti). Pri tome, treba biti oprezan zbog činjenice da dvodimenzionalni nizovi znatno više troše memoriju nego obični nizovi. Na primjer, tabela cijelih brojeva od 200 redova i 100 kolona u suštini zahtijeva prostor za memoriranje $200 \cdot 100 = 20000$ cijelih brojeva! Zbog toga, sa dimenzijama dvodimenzionalnih nizova nikada ne treba pretjerivati.

Neki programski jezici, kao što je npr. *Pascal*, dozvoljavaju da se indeksi višedimenzionalnih nizova pišu u jednom paru uglastih zagrada razdvojeni zarezom, npr.

```
tabela[2,1]
```

Što bi trebao da bude sinonim za izraz “`tabela[2][1]`”. Na žalost, ova konstrukcija, mada je sintaksno ispravna, u jeziku C++ neće proizvesti očekivano dejstvo. Naime, izraz u zagradi “`2,1`” biće shvaćen kao primjena već više puta spominjanog i “po zlu čuvenog” *zarez-operatora*. Rezultat ovog izraza biće vrijednost “`1`”, tako da će konstrukcija “`tabela[2,1]`” biti zapravo shvaćena kao “`tabela[1]`”, a ne kao “`tabela[2][1]`”. Na ovaj detalj trebaju naročito da paze korisnici koji su prije C++-a koristili *Pascal* ili neki sličan programski jezik.

Kako je svaki red dvodimenzionalnog niza obični jednodimenzionalni niz, to se redovi dvodimenzionalnog niza mogu proslijediti kao parametri funkcijama koje kao parametre očekuju jednodimenzionalne nizove odgovarajućeg tipa elemenata. Na primjer, neka je data sljedeća funkcija, koja ispisuje elemente jednodimenzionalnog niza realnih brojeva:

```

void IspisiNiz(double niz[], int broj_elemlenata) {
    for(int i = 0; i < broj_elemlenata; i++) cout << niz[i] << endl;
}

```

Ukoliko prepostavimo da je data sljedeća deklaracija:

```
double a[10][10];
```

tada su sljedeći pozivi posve legalni, i ispisuju respektivno treći i peti red tablice realnih brojeva “`a`” (ne zaboravimo da numeracije redova i kolona počinju od nule):

```

IspisiNiz(a[2], 10);
IspisiNiz(a[4], 10);

```

Međutim, sljedeći poziv nije ispravan, jer funkcija “`IspisiNiz`” očekuje *niz realnih brojeva*, dok je “`a`” *niz nizova realnih brojeva*:

```
IspisiNiz(a, 10);
```

Da bi pozivi poput prethodnog bili mogući, potrebno je napraviti funkciju koja kao parametar očekuje upravo *niz nizova* (tj. dvodimenzionalni niz). Primjer takve funkcije je sljedeća funkcija, koja ispisuje elemente dvodimenzionalnog niza proslijedenog kao parametar red po red, pri čemu su elementi unutar jednog reda razdvojeni jednim razmakom:

```
void IspisiTablicu(double tab[][][10], int m, int n) {
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) cout << tab[i][j] << " ";
        cout << endl;
    }
}
```

Ovakvu funkciju možemo pozvati na sljedeći način:

```
IspisiTablicu(a, 10, 10);
```

Bitno je uočiti jedno ograničenje kod funkcija koje kao parametre prihvataju dvodimenzionalne nizove. Naime, mada prva dimenzija niza može a i ne mora biti zadana prilikom deklaracije formalnog parametra u zaglavlju funkcije, druga dimenzija niza *obavezno mora biti zadana*. Drugim riječima, zaglavlj funkcije “IspisiTablicu“ *nije moglo izgledati ovako*:

```
void IspisiTablicu(double tab[][], int m, int n)
```

Postoje dva razloga za ovo ograničenje: *formalni* i *suštinski*. Sa *formalnog aspekta*, funkcije dopuštaju da njihovi formalni parametri budu nizovi nepoznate dužine, ali *tip elemenata niza* mora biti u potpunosti poznat (osim u slučaju generičkih funkcija, što ćemo uskoro posebno razmatrati). Kako su dvodimenzionalni nizovi zapravo nizovi nizova (tj. nizovi čiji je svaki “element” ponovo niz), njihova “dužina” (koja zapravo predstavlja broj redova) može ostati nepoznata, ali tip njihovih “elemenata” (koji su ponovo nizovi) mora biti *u potpunosti poznat, bez ikakvih nedoumica*, što zapravo znači da i broj elemenata svakog “elementa” mora biti poznat, a to je upravo broj kolona izvornog dvodimenzionalnog niza. Ovo možda zvuči zapetljano, ali postaje jasnije ukoliko dvodimenzionalni niz “a“ definiramo posredno, uvodeći pomoćni tip “Red”:

```
typedef double Red[10];
Red a[10];
```

i ukoliko funkciju “IspisiTablicu“ definiramo da ima sljedeće zaglavlj, koje je u suštini posve ekvivalentno zaglavljkoje smo prvo napisali:

```
void IspisiTablicu(Red tab[], int m, int n)
```

U ovom slučaju, formalni parametar “tab“ je niz *nepoznate dužine*, ali čiji su elementi tipa “Red”, koji je *potpuno poznat* (i predstavlja tip niza od najviše 10 realnih brojeva).

Pored navedenog razloga formalne prirode, koji se mogao prevazići da su autori jezika C++ to željeli, postoji i drugi razlog *suštinske prirode* koji je ipak motivirao *zabranu* deklaracije formalnih parametara koji predstavljaju dvodimenzionalne nizove sa obje nepoznate dimenzije. Naime, čovjek obično zamišlja da su elementi dvodimenzionalnih nizova razmješteni kao elementi neke *dvodimenzionalne tablice*, kao u sljedećem primjeru koji ilustrira kako se obično zamišlja raspored elemenata nekog dvodimenzionalnog niza (recimo “a”) sa 3 reda i 4 kolone:

0	1	2	3
---	---	---	---

0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Međutim, memorija računara nije *dvodimenzionalna* nego *jednodimenzionalna* (barem u većini računarskih arhitektura), tako da se elementi u memoriji mogu redati jedino *linearno*, odnosno *jedan za drugim*. Tako je, u suštini, tipični raspored ovih elemenata u memoriji predstavljen sljedećom slikom:

0	1	2	3	4	5	6	7	8	9	10	11
a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Vidimo da se element za koji zamišljamo da se nalazi u i -tom redu i j -toj koloni zapravo nalazi na $(4i+j)$ -toj poziciji u sekvencijalnom poretku elemenata u memoriji. Generaliziranjem ovog primjera lako možemo zaključiti da se element u i -tom redu i j -toj koloni dvodimenzionalne tablice koja ima M redova i N kolona nalazi na $(N \cdot i + j)$ -toj poziciji u sekvencijalnom poretku. Dakle, da bi se pristupilo elementu u i -tom redu i j -toj koloni, broj kolona N mora biti poznat (da bi se izračunao izraz $N \cdot i + j$), mada broj redova M ne mora da bude poznat!

Bez obzira na razloge (formalne ili suštinske), funkcija koja ima formalni parametar tipa dvodimenzionalnog niza prihvatiće kao stvarni parametar isključivo dvodimenzionalni niz čija se *deklarirana druga dimenzija* slaže sa *drugom dimenzijom navedenom u definiciji formalnog parametra*. Prepostavimo, na primjer, da imamo sljedeće deklaracije:

```
double tab1[10][10], tab2[20][10], tab3[10][20];
```

Uz ovakve deklaracije od sljedeća četiri poziva funkcije “IspisiTablicu”, prva tri su legalna, dok četvrti nije dozvoljen (jer se *deklarirana druga dimenzija* niza “tab3” koja iznosi 20 ne slaže sa *očekivanom drugom dimenzijom* formalnog parametra “tab” koja iznosi 10). Primjetimo da prvi poziv ispisuje samo prvih 5 redova i 6 kolona tablice “tab1” od ukupno 10 redova i 10 kolona (što ujedno ilustrira kakva je svrha postojanja parametara “m” i “n”, a pogotovo parametra “n” s obzirom da druga dimenzija mora biti fiksirana u deklaraciji parametra):

```
IspisiTablicu(tab1, 5, 6);
IspisiTablicu(tab1, 10, 10);
IspisiTablicu(tab2, 20, 10);
IspisiTablicu(tab3, 10, 20);
```

Opisani problem nije bio rješiv sve do uvođenja mehanizma generičkih funkcija (osim višestrukim preklapanjem funkcija po tipu, odnosno pisanjem posebnih funkcija sa identičnim tijelom za svaku očekivanu drugu dimenziju dvodimenzionalnog niza). Mehanizam generičkih funkcija pruža nekoliko mogućnosti da se ipak naprave funkcije poput funkcije “IspisiTablicu”, a koje će priхватiti dvodimenzionalni niz čija ni jedna dimenzija nije fiksna (dakle, za koje će sva četiri prethodna poziva biti posve legalna). Jedno (lošije) rješenje zasniva se na sljedećem. Poznato je da generičke (šablonske) funkcije mogu da rade sa tipovima koji nisu unaprijed poznati. Stoga, možemo da napravimo sljedeću generičku funkciju:

```
template <typename Tip>
```

```

void IspisiTablicu(Tip tab[], int m, int n) {
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) cout << tab[i][j] << " ";
        cout << endl;
    }
}

```

Ideja ove funkcije zasniva se na činjenici da ukoliko joj kao prvi parametar proslijedimo npr. dvodimenzionalni niz "tab2", tip "Tip" će se, u postupku dedukcije tipa, identificirati sa *nizovnim tipom od najviše 10 realnih elemenata* (jer je "tab2" niz od 20 takvih nizova), što se uklapa u šablon, i funkcija se može izvršiti. Ovakvo rješenje je nezgodno iz dva razloga. Prvo, ovako napisanoj funkciji "IspisiTablicu" može se kao prvi parametar proslijediti ne samo dvodimenzionalni niz, nego i bilo kakav obični jednodimenzionalni niz (pri čemu će se tip "tip" identificirati sa tipom elemenata tog niza), nakon čega će funkcija da "padne" pri pokušaju da se parametar "tab" indeksira sa dva indeksa. Drugi razlog je što se pri regularnom pozivu funkcije nad dvodimenzionalnim nizom tip "tip" identificira sa čitavim *nizovnim tipom*, dok ne postoji nikakav način da saznamo kakav je tip *individualnih elemenata tog niza*, što može jako ograničiti primjenu funkcije. Na primjer, uz ovakav pristup ne postoji nikakav način da unutar funkcije deklariramo neku lokalnu promjenljivu čiji se tip slaže sa tipom individualnih elemenata dvodimenzionalnog niza (što bi nam, recimo, trebalo ukoliko bismo željeli napisati funkciju koja vraća kao rezultat sumu svih elemenata dvodimenzionalnog niza elemenata proizvoljnog tipa za koji je definirana operacija sabiranja). Primijetimo da smo u ovom rješenju koristili *djelimičnu dedukciju tipa*, s obzirom da smo u zaglavlju funkcije eksplisitno specificirali da parametar "tab" predstavlja niz. U konkretnom primjeru funkcije "IspisiTablicu" mogli smo koristiti i *potpunu dedukciju tipa* (izostavljanjem uglastih zagrada u zaglavlju funkcije). U tom slučaju, tip "Tip" bi se pri pozivu funkcije sa stvarnim prvim parametrom "tab2" poistovijetio sa *cjelokupnim tipom* parametra "tab2" (odnosno sa *nizovnim tipom čiji su elementi nizovi od najviše 10 elemenata*). Jasno je da bi uz ovaku dedukciju funkcija "IspisiTablicu" i dalje radila ispravno, ali time uočene probleme nismo riješili, nego smo ih samo pogoršali. Naime, u ovom slučaju, funkcija "IspisiTablicu" mogla bi se pozvati sa prvim parametrom koji čak *uopće nije niz* (nakon čega bi funkcija ponovo "pala" pri pokušaju indeksiranja parametra "tab").

Rješenje koje uklanja gore opisane nedostatke ipak postoji, i posve je jednostavno, ali začudo, gotovo je u potpunosti nepoznato među C++ programerima (od više desetina knjiga o jeziku C++ koje su poznate autoru ovog materijala, ovo rješenje se samo uzgred spominje jedino u knjizi "Thinking in C++" autora Brucea Eckela). Ideja se sastoji u sljedećem: parametri šablona (tj. parametri navedeni u šiljastim zgradama iza ključne riječi "**template**") ne moraju obavezno biti *metatipovi* (deklarirani ključnom riječju "**typename**"), već mogu imati i neke fiksne tipove, poput tipa "**int**". Ovakvi parametri šablona koriste se u nekim specifičnim primjenama, i za njih se obično smatra da nikada ne učestvuju u postupku dedukcije tipa, već da se uvijek moraju eksplisitno specificirati (navođenjem unutar šiljastih zagrada) prilikom poziva generičke funkcije. Međutim, ovo nije u potpunosti tačno, jer postoje rijetke situacije u kojima parametri šablona koji *nisu metatipovi* ipak učestvuju u dedukciji tipa, što se može iskoristiti za *pogačanu djelimičnu dedukciju tipa*. Jedan takav primjer je sljedeća verzija funkcije "IspisiTablicu", u kojoj su rješena oba ranije uočena problema:

```

template <typename Tip, int DrugaDimenzija>
void IspisiTablicu(Tip tab[] [Drugadimenzija], int m, int n) {
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) cout << tab[i][j] << " ";
        cout << endl;
    }
}

```

U ovom primjeru, pored metatipa “Tip”, kao parametar šablonu upotrijebljen je i cjelobrojni parametar “DrugaDimenzija”, koji je dalje upotrijebljen unutar specifikacije tipa formalnog parametra “tab”. Kako je broj elemenata nizova koji tvore svaki od redova dvodimenzionalnog niza sastavni i neodvojivi dio tipa, to će i parametar šablonu “DrugaDimenzija” aktivno učestvovati u dedukciji tipa. Na primjer, ukoliko se kao prvi stvarni parametar u ovu funkciju proslijedi dvodimenzionalni niz “tab2”, u postupku dedukcije tipa će se zaključiti da je uklapanje u šablon moguće jedino ukoliko se parametar šablonu “Tip” poistovijeti sa tipom “**double**”, a parametar šablonu “DrugaDimenzija” sa *brojem* 10. Dakle, parametar šablonu “Tip” se identificira sa *tipom individualnih elemenata niza*, što je upravo ono što nam treba. Na taj način bismo mogli napraviti i neku funkciju u kojoj je ovakva informacija *bitna*, npr. funkciju koja sabira elemente unutar prvih “m” redova i “n” kolona dvodimenzionalnog niza nepoznatih dimenzija. Ona bi mogla izgledati recimo ovako:

```
template <typename Tip, int DrugaDimenzija>
Tip SumaElemenata(Tip tab[] [DrugaDimenzija], int m, int n) {
    Tip suma(0);
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++) suma += tab[i][j];
    return suma;
}
```

Primijetimo da u ovako napisanim funkcijama prvi parametar zaista *mora biti dvodimenzionalni niz* (jer u suprotnom uklapanje u šablon nije moguće), tako da su na ovaj način uklonjena oba nedostatka o kojima smo prethodno diskutirali. Primijetimo također da nam je, u ovako napisanim funkcijama, preko parametra “DrugaDimenzija” dostupna i informacija o stvarnoj (deklariranoj) drugoj dimenziji upotrijebljenog stvarnog parametra, što ponekad može biti veoma korisno. Parametri šablonu koji nisu metatipovi (poput parametra “DrugaDimenzija” u ovim primjerima) tretiraju se unutar tijela funkcije kao *prave konstante*. Interesantno je još napomenuti da se dedukcija tipa može izvesti čak i ukoliko niti jedan parametar šablonu nije metatip. Na primjer, da smo funkciju “IspisiTablicu” htjeli modificirati tako da prihvata samo dvodimenzionalne nizove *realnih brojeva* proizvoljnih dimenzija, to smo mogli učiniti ovako:

```
template <int DrugaDimenzija>
void IspisiTablicu(double tab[] [DrugaDimenzija], int m, int n) {
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) cout << tab[i][j] << " ";
        cout << endl;
    }
}
```

Treba napomenuti da opisana rješenja, zasnovana na pojačanoj djelimičnoj dedukciji tipa, ipak nisu idealna (mada rješavaju prethodno istaknute probleme). Naime, kako je mehanizam generičkih funkcija zapravo vrsta *automatskog preklapanja funkcija po tipu parametara*, bilo koja od prethodno napisanih generičkih funkcija “IspisiTablicu” ne predstavlja jednu samostalnu funkciju, nego *cijelu familiju preklopnih funkcija*. Pri tome se, svaki put kada se na mjestu prvog argumenta funkcije upotrijebi dvodimenzionalni niz čiji se broj kolona *razlikuje* od broja kolona nizova upotrijebljenih u *prethodnim pozivima* iste funkcije, kreira *nova instanca funkcije* prilagođena tipu koji je određen u postupku dedukcije. Na taj način, veličina izvršnog programa može osjetno da naraste ukoliko se ova funkcija poziva na mnogo mjesta u programu i ukoliko pri tome stvarni parametri u tim pozivima imaju različite brojne kolone. Naravno, ista primjedba vrijedi i za funkciju “SumaElemenata”, i sve druge funkcije zasnovane na istom principu. Zbog toga, opisana rješenja zasnovana na primjeni generičkih funkcija treba koristiti samo u slučajevima u kojima se traži velika općenitost i fleksibilnost.

Tehnika prenosa višedimenzionalnih nizova kao parametara u funkcije dodatno je ilustrirana kroz sljedeći primjer. Radi se o istom primjeru za računanje prosječnih ocjena za skupinu studenata koji je već bio razmotren, samo što je ovaj put napisan na modularan način, uz upotrebu funkcija i prenosa parametara u funkcije. Funkcija "UnesiOcjene" unosi ocjene za sve predmete u dvodimenzionalni niz koji je proslijeden kao parametar, dok funkcija "Prosjek" računa prosjek ocjena za jednog studenta, čije su ocjene proslijedene kao parametar:

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

using namespace std;

const int MaxBrojStudenata(50), MaxBrojPredmeta(5);

void UnesiOcjene(int tabela[][MaxBrojPredmeta],
                  int broj_studenata, int broj_predmeta) {
    for(int student = 0; student < broj_studenata; student++) {
        cout << "Unesi ocjene za studenta " << student + 1 << ":\n";
        for(int predmet = 0; predmet < broj_predmeta; predmet++) {
            cout << " Predmet " << predmet + 1 << ": ";
            cin >> tabela[student][predmet];
        }
        cout << endl;
    }
}

double Prosjek(const int ocjene_jednog_studenta[], int n) {
    double suma(0);
    for(int i = 0; i < n; i++) suma += ocjene_jednog_studenta[i];
    return suma / n;
}

int main() {
    int broj_studenata, broj_predmeta;
    cout << "Koliko ima studenata: ";
    cin >> broj_studenata;
    cout << "Koliko ima predmeta: ";
    cin >> broj_predmeta;
    int ocjene[MaxBrojStudenata][MaxBrojPredmeta];
    UnesiOcjene(ocjene, broj_studenata, broj_predmeta);
    cout << "Broj studenta      Prosječna ocjena\n";
    cout << "-----      ----- \n";
    for(int student = 0; student < broj_studenata; student++)
        cout << setw(8) << student + 1 << setw(20) << setprecision(2)
            << Prosjek(ocjene[student], broj_predmeta) << endl;
    return 0;
}
```

Slično kao kod rada sa jednodimenzionalnim nizovima, i kod rada sa dvodimenzionalnim nizovima često se pokazuje isplatno pomoću ključne riječi "**typedef**" imenovati novi tip koji predstavlja dvodimenzionalni nizovni tip, a zatim novoimenovani tip koristiti za deklaraciju konkretnih promjenljivih tog tipa, kao i za deklaraciju formalnih parametara tog tipa. Na primjer, deklaracija

```
typedef double Matrica[10][10];
```

definira novi tip "Matrica" koji predstavlja (apstraktni) dvodimenzionalni niz (matricu) realnih brojeva sa najviše 10 redova i najviše 10 kolona. Ovaj tip možemo iskoristiti za definiranje konkretnih matrica. Na

primjer, deklaracijom

```
Matrica a, b, c;
```

deklariramo tri konkretnе promjenljive "a", "b" i "c" čiji je tip *dvodimenzionalni niz realnih brojeva sa najviše 10 redova i kolona*. Također je moguće novodefinirani tip "Matrica" iskoristiti za deklaraciju formalnih parametara funkcija. Na primjer, sljedeći prototip funkcije

```
void IspisiMatricu(Matrica a, int m, int n);
```

potpuno je ekvivalentan sljedećem prototipu:

```
void IspisiMatricu(double a[10][10], int m, int n);
```

Opisani pristup je naročito koristan u slučajevima kada se u programu koristi više funkcija koje kao parametre primaju dvodimenzionalne nizove iste vrste. Na taj način se omogućava konzistentno tipiziranje parametara u svim funkcijama. Tako, u slučaju da se npr. promijene deklarirane dimenzije matrice, izmjenu je dovoljno izvršiti samo u deklaraciji tipa, dok cijeli program ostaje potpuno neizmijenjen. Ovaj pristup je ilustriran u sljedećem programu koji od korisnika traži da unese dvije matrice istog formata (broj redova i kolona također zadaje korisnik), a zatim računa i ispisuje zbir dvije unesene matrice. Obratite pažnju na funkciju "UnesiMatricu" koja prikazuje korisniku informacije o indeksima reda i kolone elementa koji se unosi na način kako je uobičajeno u matematici (indeksiranje od jedinice naviše):

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

using namespace std;

const int MaxM(20), MaxN(20);

typedef double Matrica[MaxM][MaxN];

void UnesiMatricu(Matrica a, int m, int n) {
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++) {
            cout << " Element (" << i + 1 << "," << j + 1 << "): ";
            cin >> a[i][j];
        }
}

void SaberiMatrice(const Matrica a, const Matrica b, Matrica c,
int m, int n) {
    for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++)
            c[i][j] = a[i][j] + b[i][j];
}

void IspisiMatricu(const Matrica a, int m, int n) {
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++)
            cout << setw(8) << a[i][j];
        cout << endl;
    }
}

int main() {
    int broj_redova, broj_kolona;
```

```

    Matrica m1, m2, m3;
    cout << "Unesi broj redova i broj kolona matrica: ";
    cin >> broj_redova >> broj_kolona;
    cout << "Unesi prvu matricu:\n";
    UnesiMatricu(m1, broj_redova, broj_kolona);
    cout << "\nUnesi drugu matricu:\n";
    UnesiMatricu(m2, broj_redova, broj_kolona);
    SaberiMatrice(m1, m2, m3, broj_redova, broj_kolona);
    cout << "\nZbir ovih matrica je:\n";
    IspisiMatricu(m3, broj_redova, broj_kolona);
    return 0;
}

```

Funkcija “Saberimatrice“ u ovom programu prima tri matrice kao parametre: “a“ i “b“ predstavljaju matrice koje se sabiraju, dok “c“ predstavlja matricu u koju će biti smješten izračunati zbir dvije matrice. Bilo bi mnogo ljepše kad bi funkcija “Saberimatrice“ mogla da vrati zbir matrica kao rezultat, tj. kad bi njen prototip mogao izgledati ovako:

```
Matrica SaberiMatrice(const Matrica a, const Matrica b, int m, int n);
```

Pri tome bi poziv funkcije “Saberimatrice“ trebao izgledati ovako:

```
m3 = SaberiMatrice(m1, m2, broj_redova, broj_kolona);
```

Na žalost, već znamo da funkcije u jeziku C++ ne mogu vraćati nikakve nizove kao rezultat, pa tako ni višedimenzionalne nizove. Stoga, ovakve deklaracije i ovakvi pozivi nisu mogući (bar ne sa ovako definiranim tipom “*Matrica*”). Kasnije ćemo vidjeti da se ovaj problem može riješiti upotrebom tzv. *struktura* ili *klasa* (tj. definiranjem matrice kao strukture ili klase, a ne kao niza), jer za razliku od nizova, funkcije mogu vraćati strukture ili klase kao rezultat.

Dvodimenzionalni nizovi se također mogu inicijalizirati prilikom deklaracije. Pri tome se inicijalizaciona lista mora sastojati od listi koje služe za inicijalizaciju svakog od redova posebno. Na primjer:

```
int a[3][4] = {{3, 4, 2, 5}, {4, 1, 2, 3}, {3, 1, 0, 2}};
```

Inicijalizacione liste za dvodimenzionalne nizove često se pišu tako da se svaka od listi koje služe za inicijalizaciju pojedinih redova piše u novom redu. Na taj način inicijalizacija postaje preglednija, a matrična struktura dvodimenzionalnog niza uočljivija:

```
int a[3][4] = {{3, 4, 2, 5},
                {4, 1, 2, 3},
                {3, 1, 0, 2}};
```

Kao i u slučaju jednodimenzionalnih nizova, ukoliko inicijalizaciona lista sadrži manje elemenata nego što iznose deklarirane dimenzije niza, nedostajući elementi se inicijaliziraju na nulu. Tako se u sljedećem primjeru, u matrici “m“ prva tri elementa u prvom redu (opravno redu 0) i prva dva elementa u sljedećem redu inicijaliziraju na navedene vrijednosti, dok se svi ostali elementi (od ukupno 10×10 elemenata uz pretpostavku da je tip “*Matrica*“ definiran kao matrica sa maksimalno 10 redova i 10 kolona, kao u jednom od prethodnih primjera) inicijaliziraju na nulu:

```
Matrica m = {{3, 5, 6}, {4, -2}};
```

Kod inicijaliziranih dvodimenzionalnih nizova, prva dimenzija se može izostaviti, pri čemu se tada

ona automatski određuje na osnovu broja podlisti za inicijalizaciju redova unutar inicijalizacione liste. Tako će, u sljedećem primjeru, prva dimenzija niza biti automatski postavljena na 3:

```
int a[][][4] = {{3, 4, 2, 5}, {4, 1, 2, 3}, {3, 1, 0, 2}};
```

Međutim, druga dimenzija se *ne smije izostaviti*, tj. ne smije se pisati:

```
int a[][][] = {{3, 4, 2, 5}, {4, 1, 2, 3}, {3, 1, 0, 2}};
```

Razlozi za ovu zabranu su identični razlozima zbog čega nije dozvoljeno izostaviti drugu dimenziju u deklaraciji formalnih parametara funkcija.

Među dvodimenzionalnim nizovima naročito se ističu *dvodimenzionalni nizovi znakova*, koji se, uz poštovanje konvencije o “NUL” graničniku, mogu posmatrati kao *nizovi nul-terminaliranih stringova*. Svaki element takvog niza je *obični niz znakova*, odnosno *nul-terminalirani string*, sa kojim se može postupati na način koji je opisan za rad sa nizovima znakova. Ovakve nizove možemo koristiti kad god je potrebno pamćenje skupine tekstualnih podataka, na primjer pamćenje imena svih učenika u razredu, imena svih mjeseci u godini, itd. Kao ilustraciju kako raditi sa dvodimenzionalnim nizom znakova, sljedeći program omogućava korisniku da unese spisak imena, i tada da izabere podspisak (na primjer imena sa rednim brojevima od 10 do 20) koji će biti isписан. Primijetimo da u ovom programu konstanta “MaxDuzina” predstavlja maksimalnu dužinu imena uvećanu za 1, s obzirom da se jedan znak rezervira za pamćenje “NUL” graničnika:

```
#include <iostream>
using namespace std;
const int MaxVelicinaSpiska(100), MaxDuzina(50);
void UnesiImena(char imena[][MaxDuzina], int n) {
    for(int i = 0; i < n; i++) {
        cout << "Unesi " << i + 1 << ". ime: ";
        cin.getline(imena[i], MaxDuzina);
    }
}
void IspisiImena(char imena[][MaxDuzina], int odakle, int dokle) {
    for(int i = odakle; i <= dokle; i++)
        cout << imena[i - 1] << endl;
}

int main() {
    int broj, pocetak, kraj;
    char spisak[MaxVelicinaSpiska][MaxDuzina];
    cout << "Koliko imena želite da unesete? ";
    cin >> broj;
    cin.ignore(10000, '\n');
    UnesiImena(spisak, broj);
    cout << "\nUnesite redni broj prvog i zadnjeg imena kojeg želite: ";
    cin >> pocetak >> kraj;
    IspisiImena(spisak, pocetak, kraj);
    return 0;
}
```

Može se primijetiti da u funkciji “IspisiImena” formalni parametar “imena” nije označen kao “**const**” parametar, iako izgleda da bi mogao da bude. Međutim, zbog prilično čudnih konverzija nizova

u pokazivače koji se odvijaju prilikom prenosa dvodimenzionalnih nizova kao parametara u funkcije, upotreba kvalifikatora “**const**” u ovom primjeru bi samo izazvala probleme (kompajler bi prilikom poziva funkcije posve neočekivano prijavio neslaganje tipova). Ovdje nećemo detaljno elaborirati zbog čega je tako, već ćemo samo istaći da kvalifikator “**const**” treba izbjegavati kad god formalni parametar ima jasnu sintaksnu formu dvodimenzionalnog niza. S druge strane, ukoliko je ta dvodimenzionalnost na neki način *zamaskirana* (npr. pomoću “**typedef**” deklaracija, kao u primjeru u kojem je uveden tip “*Matrica*”), upotreba kvalifikatora “**const**” ne dovodi do problema.

Prilikom rada sa nizovima nul-terminaliranih stringova, dobra je ideja prvo pomoći “**typedef**” naredbe definirati poseban tip koji predstavlja niz znakova određene maksimalne dužine, a zatim taj tip iskoristiti za definiranje nizova takvog tipa. Ovaj pristup je naročito pogodan ukoliko treba definirati više različitih nizova nad istim stringovnim tipom. Ukoliko se pojavi potreba da mijenjamo strukturu stringovnog tipa, tada se promjena treba izvršiti samo na mjestu gdje je tip definiran (tj. u “**typedef**” deklaraciji). Sljedeći program je modifikacija prethodnog programa, u kojem je iskorišten ovaj pristup:

```
#include <iostream>
using namespace std;
const int MaxVelicinaSpiska(100), MaxDuzina(50);
typedef char Ime[MaxDuzina];
void UnesiImena(Ime imena[], int n) {
    for(int i = 0; i < n; i++) {
        cout << "Unesi " << i + 1 << ". ime: ";
        cin.getline(imena[i], MaxDuzina);
    }
}
void IspisiImena(const Ime imena[], int odakle, int dokle) {
    for(int i = odakle; i <= dokle; i++)
        cout << imena[i - 1] << endl;
}
int main() {
    int broj, pocetak, kraj;
    Ime spisak[MaxVelicinaSpiska];
    cout << "Koliko imena želite da unesete? ";
    cin >> broj;
    cin.ignore(10000, '\n');
    UnesiImena(spisak, broj);
    cout << "\nUnesite redni broj prvog i zadnjeg imena kojeg želite: ";
    cin >> pocetak >> kraj;
    IspisiImena(spisak, pocetak, kraj);
    return 0;
}
```

Primijetimo da je formalni parametar “imena” u funkciji “IspisiImena” propisno označen kvalifikatorom “**const**”, što u ovom primjeru ne dovodi do problema, jer parametar “imena” u ovom primjeru ne izgleda formalno poput dvodimenzionalnog niza. (dvodimenzionalnost je zamaskirana uvođenjem posebnog tipa “*Ime*”, tako da parametar “imena” izgleda kao jednodimenzionalni niz čiji su elementi tipa “*Ime*”).

Dvodimenzionalne nizove znakova je također moguće inicijalizirati prilikom deklaracije. Poput svih dvodimenzionalnih nizova, i njih je moguće inicijalizirati element po element (tj. znak po znak), kao npr. u sljedećem primjeru:

```

char imena_dana[7][12] =
{{'P', 'o', 'n', 'e', 'd', 'j', 'e', 'l', 'j', 'a', 'k', 0},
 {'U', 't', 'o', 'r', 'a', 'k', 0},
 {'S', 'r', 'i', 'j', 'e', 'd', 'a', 0},
 {'Č', 'e', 't', 'v', 'r', 't', 'a', 'k', 0},
 {'P', 'e', 't', 'a', 'k', 0},
 {'S', 'u', 'b', 'o', 't', 'a', 0},
 {'N', 'e', 'd', 'j', 'e', 'l', 'j', 'a', 0}};

```

Kako je ovakva inicijalizacija krajne nezgrapna, dozvoljeno je da se pojedini redovi unutar dvodimenzionalnog niza znakova inicijaliziraju stringovnim konstantama (svaki red se pri tome tretira kao nul-terminirani string), kao u sljedećoj inicijalizaciji, koja predstavlja mnogo pregledniju varijantu prethodne inicijalizacije:

```

char imena_dana[] [12] = {"Ponedjeljak", "Utorak", "Srijeda",
 "Četvrtak", "Petak", "Subota", "Nedjelja"};

```

Ovdje je također iskorištena činjenica da se prva dimenzija prilikom definiranja inicijaliranih dvodimenzionalnih nizova može izostaviti. Međutim, kako se druga dimenzija *mora zadati*, nju je potrebno zadati prema dužini *najdužeg nul-terminiranog stringa* koji će se pamtitи u nizu (ovdje je to string “Ponedjeljak” čija je dužina 11, a dimenzija 12 je postavljena zbog potrebe za pamčenjem “NUL” graničnika). Očigledno, ovakvo rješenje ima nedostatak što se za svaki string mora zauzeti *ista količina memorije* (prema dužini najdužeg stringa), bez obzira što neki stringovi mogu biti znatno kraći. Na ovaj način se neracionalno troši memorija. Kasnije ćemo vidjeti da se ovaj problem može efikasno riješiti uz pomoć pokazivača. Međutim, uvođenje dinamičkih stringova, odnosno tipa “string” u jezik C++, omogućilo je još jednostavnije rješenje: umjesto dvodimenzionalnih nizova znakova, može se koristiti *niz čiji su elementi tipa “string”* (čija dužina nije fiksna, nego se može dinamički mijenjati tokom rada programa). Mada takvi nizovi striktno rečeno nisu dvodimenzionalni nizovi (jer elementi tipa “string” nisu nizovi), oni se u velikom broju situacija mogu koristiti istovjetno kao dvodimenzionalni nizovi znakova (s obzirom da se elementi tipa “string” u mnogim kontekstima mogu tretirati na isti način kao nizovi znakova). Na primjer, moguća je deklaracija poput sljedeće:

```

string imena_dana[] = {"Ponedjeljak", "Utorak", "Srijeda",
 "Četvrtak", "Petak", "Subota", "Nedjelja"};

```

Naravno, svaki element niza elemenata tipa “string” je sam za sebe tipa “string”, tako da se na individualne elemente ovakvih nizova mogu primijeniti sve operacije definirane sa stringovima koje su opisane u prethodnom poglavlju. Također, prethodni primjer programa za rad sa spiskom imena može se na trivijalan način prepraviti da radi sa nizom elemenata tipa “string”. Za tu svrhu, dovoljno je izbaciti deklaracije konstante “MaxDuzina” i tipa “Ime”, umjesto tipa “Ime” koristiti tip “string”, i izmijeniti naredbu za unos u funkciji “UnesiImena”. Ova modifikacija ostavlja se čitatelju ili čitateljici za vježbu.

Inicijalizirani nizovi stringova (bilo klasičnih nul-terminiranih, bilo dinamičkih) mogu se često iskoristiti za osjetno skraćivanje programa. Na primjer, sljedeća funkcija, koja ispisuje naziv mjeseca čiji je redni broj zadan kao parametar, bez nizova stringova mogla bi se napisati jedino pomoću glomaznih višestrukih “if” – “else” konstrukcija ili pomoću “switch” – “case” konstrukcija:

```

void IspisiImeMjeseca(int mjesec) {
    char imena_mjeseci[][10] = {"Januar", "Februar", "Mart", "April",
 "Maj", "Juni", "Juli", "August", "Septembar", "Oktobar",
 "Novembar", "Decembar"};
    cout << imena_mjececi[mjesec - 1] << endl;
}

```

Prikazana funkcija koristi klasični dvodimenzionalni niz znakova. Sljedeća funkcija ostvaruje isti efekat, ali uz upotrebu nizova elemenata tipa "string". Ova funkcija je efikasnija sa aspekta utroška memorije, ali zahtijeva uključenje biblioteke "string" u program:

```
void IspisiImeMjeseca(int mjesec) {
    string ImenaMjeseci[] = {"Januar", "Februar", "Mart", "April",
    "Maj", "Juni", "Juli", "August", "Septembar", "Oktobar",
    "Novembar", "Decembar"};
    cout << imena_mjececi[mjesec - 1] << endl;
}
```

Naravno, posljednja funkcija bi se lako mogla prepraviti tako da *vrati kao rezultat ime mjeseca* umjesto da ga prosto ispiše, s obzirom da funkcije bez problema mogu vraćati rezultate tipa "string". To nije slučaj sa prethodnom funkcijom, koja koristi dvodimenzionalni niz znakova.

Kao što je moguće da elementi nizova budu *dinamički stringovi* (odnosno elementi tipa "string"), moguće je formirati i *nizove vektora* (odnosno nizove čiji su elementi tipa "vector"). Također je moguće formirati i *vektore vektora* (odnosno vektore čiji su elementi tipa "vector") kao i *vektore dinamičkih stringova* (odnosno vektore čiji su elementi tipa "string"). Na taj način dobijamo raznovrsne strukture podataka koje, striktno posmatrano, nisu dvodimenzionalni nizovi (kao što vektori nisu nizovi), ali po mnogim osobinama liče na njih, i u mnogim kontekstima se mogu koristiti na isti način kao i dvodimenzionalni nizovi. Na primjer, sljedeća deklaracija definira niz "a" od 10 elemenata, pri čemu je svaki od elemenata vektor realnih brojeva:

```
vector<double> a[10];
```

Primijetimo da su ovoj deklaraciji upotrijebljene uglaste zagrade, kao što je tipično prilikom deklaracije nizova. Da smo umjesto uglastih zagrada upotrijebili obične (okrugle) zagrade, deklarirali bismo, kao što već znamo, *obični vektor* koji inicijalno sadrži 10 realnih brojeva. Uglaste zgrade naglašavaju da deklariramo *niz* (ova razlika je analogna razlici između deklaracije "int a(5)" koja deklarira cjelobrojnu promjenljivu "a" inicijaliziranu na vrijednost "5", i deklaracije "int a[5]" koja deklarira niz koji može primiti 5 cjelobrojnih vrijednosti). Inicijalno su elementi ovog niza *prazni vektori* (koje možemo napuniti recimo izvršavanjem operacije "push_back" nad njegovim elementima, npr. konstrukcijom poput "a[3].push_back(1)"). Da bismo niz "a" mogli koristiti kao matricu, moramo sve vektore koji su njegovi elementi "raširiti" primjenom operacije "resize" na isti broj elemenata, koji će predstavljati željeni broj kolona matrice. Na primjer, ukoliko želimo da matrica ima 5 kolona, to možemo obaviti na sljedeći način:

```
for(int i = 0; i < 10; i++) a[i].resize(5);
```

Nakon toga, niz "a" možemo koristiti poput klasične dvodimenzionalne matrice sa 10 redova i 5 kolona. Međutim, opisana struktura je znatno fleksibilnija od običnog dvodimenzionalnog niza. Na primjer, možemo primijetiti da pomoću operacije "resize" lako možemo postići da pojedini redovi ovakve strukture imaju *različit broj elemenata*, što jasno ne odgovara strukturi matrice. Strukture podataka koje liče na matrice, ali čiji pojedini redovi mogu imati različit broj elemenata nazivaju se *grbavi nizovi* (engl. *ragged arrays*), i pogodni su npr. za pamćenje *simetričnih matrica*, kod kojih nije neophodno pamtitи elemente iznad glavne dijagonale, s obzirom da su oni identični elementima ispod glavne dijagonale na pozicijama koje su simetrično preslikane u odnosu na glavnu dijagonalu. O grbavim nizovima ćemo detaljnije govoriti u poglavljima koja govore o pokazivačima i dinamičkoj alokaciji memorije.

Primijetimo da je opisana struktura podataka pogodna za definiranje matrica čiji je broj redova unaprijed poznat, ali *kod kojih broj kolona nije unaprijed poznat* (s obzirom da se veličina vektora može dinamički određivati prilikom izvršavanja programa). Ukoliko unaprijed nije poznat niti broj redova niti

broj kolona matrice, moguće je koristiti *vektore vektora*. Na primjer, pomoću sljedeće konstrukcije formira se vektor vektora realnih brojeva pod nazivom “*a*”, koji se može koristiti poput matrice sa “*m*” redova i “*n*” kolona, pri čemu niti “*m*” niti “*n*” ne moraju biti konstante:

```
vector<vector<double>> a (m);  
for(int i = 0; i < m; i++) a[i].resize(n);
```

Ovdje treba istaći da je razmak između dva znaka “>” *bitan*, jer bi se u suprotnom slijed od dva znaka “>” zaredom mogao pogrešno interpretirati kao operator “>>”.

Analogno dvodimenzionalnim nizovima, postoje i *trodimenzionalni, četverodimenzionalni* i općenito *višedimenzionalni nizovi*, pri čemu se rijetko koriste više od 3 dimenzije. U suštini, trodimenzionalni niz možemo shvatiti kao *niz čiji su elementi dvodimenzionalni nizovi*, ili kao *dvodimenzionalni niz čiji su elementi obični nizovi*. U svakom slučaju, trodimenzionalnim nizovima može se pristupati preko *jednog, dva ili tri indeksa*. Ako zamislimo trodimenzionalni niz kao višeslojnu prostornu strukturu čiji svaki sloj predstavlja matricu, tada navođenjem jednog indeksa pristupamo navedenom sloju, navođenjem dva indeksa pristupamo navedenom redu u navedenom sloju, dok navođenjem tri indeksa pristupamo navedenom elementu u navedenom redu u navedenom sloju. Kao primjer trodimenzionalnog niza, možemo posmatrati situaciju u kojoj imamo školu sa 10 odjeljenja, u kojoj u svakom odjeljenju ima po 36 učenika, a svaki učenik ima 12 predmeta. Ocjene svih učenika u ovoj školi možemo predstaviti sljedećim trodimenzionalnim nizom:

```
int ocjene[10][36][12];
```

Elementima ovakvog niza pristupamo sa tri indeksa. Prvi indeks predstavlja broj odjeljenja, drugi broj predstavlja redni broj učenika unutar izabranog odjeljenja, dok treći indeks predstavlja redni broj predmeta za izabranog učenika. Na primjer, izraz “*ocjene[4]*” predstavlja ocjene svih učenika iz četvrтog odjeljenja (dvodimenzionalni niz), izraz “*ocjene[4][7]*” predstavlja ocjene sedmog učenika iz četvrтog odjeljenja (jednodimenzionalni niz), dok izraz “*ocjene[4][7][5]*” predstavlja petu ocjenu sedmog učenika iz četvrтog razreda (broj).

Trodimenzionalne nizove možemo definirati i postupno pomoću “**typedef**“ naredbe na više različitih načina. Na primjer, niz “*ocjene*” iz prethodnog primjera moguće je definirati i na sljedeća četiri ekvivalentna načina, koji ujedno ilustriraju različite načine na koje se trodimenzionalni niz može interpretirati:

```
typedef int OcjeneJednogUcenika[12];  
OcjeneJednogUcenika ocjene[10][36];  
  
typedef int OcjeneJednogUcenika[12];  
typedef OcjeneJednogUcenika OcjeneJednogOdjeljenja[36];  
OcjeneJednogOdjeljenja ocjene[10];  
  
typedef int OcjeneJednogOdjeljenja[36][12];  
OcjeneJednogOdjeljenja ocjene[10];  
  
typedef int OcjeneSkole[10][36][12];  
OcjeneSkole ocjene;
```

Slična logika vrijedi i za nizove sa više od tri dimenzije. O višedimenzionalnim nizovima vrijedi još reći da prilikom prenošenja parametara u funkcije koji su višedimenzionalni nizovi, u odgovarajućem formalnom parametru sve dimenzije moraju biti zadane osim prve, koja se može izostaviti (ovaj problem se također može riješiti pomoću generičkih funkcija i dedukcije tipa). Inicijalizacija višedimenzionalnih nizova može se vršiti pomoću ugniježdenih listi, analogno kao u slučaju dvodimenzionalnih nizova. Pri

tome se također sve dimenzije moraju eksplisitno navesti, osim eventualno prve dimenzije.

24. Izuzeci i njihova obrada

Prilikom pisanja iole komplikovanih programa često se dešava da neke od funkcija koje trebamo napisati nemaju smisla za sve vrijednosti argumenata. Na primjer, mnoge matematičke funkcije nisu definirane za sve vrijednosti svojih argumenata (ova primjedba se ne odnosi samo na matematičke funkcije – recimo, funkcija koja na ekranu iscrta kvadrat sastavljen od zvjezdica čije se dimenzije zadaju kao parametar nema smisla ukoliko se kao parametar zada negativan broj). Stoga je prirodno postaviti pitanje *šta bi funkcija trebala da vrati kao rezultat* ukoliko joj se kao parametri proslijede takve vrijednosti da funkcija nije definirana, ili općenito, *šta bi funkcija trebala da uradi* ukoliko su joj proslijedeni takvi parametri za koje ona nema smisla. Sve do nedavno, ovaj problem se rješavao na razne načine koji su bili više improvizacije nego prava rješenja. U posljednje vrijeme, standard jezika C++ je predvio način za rješavanje ovog problema zasnovan na tzv. *izuzecima* (engl. *exceptions*). Međutim, prije nego što razmotrimo izuzetke i tehnike njihove obrade, recimo nešto i o starijim tehnikama kako bi se moglo pristupiti ovom problemu. Na taj način ćemo bolje shvatiti prednosti koje oni donose.

Uzmimo kao primjer funkciju za računanje faktorijela. Ova funkcija nije definirana za slučaj kada je argument negativan. Posmatrajmo međutim kako će reagirati standardne izvedbe iterativne i rekursivne verzije funkcije za računanje faktorijela ukoliko im se kao argument ponudi negativan broj:

```
long int Faktorijel(int n) {
    long int p(1);
    for(int i = 1; i <= n; i++) p *= i;
    return p;
}

long int FaktorijelRekursivni(int n) {
    if(n == 0) return 1;
    else return n * FaktorijelRekursivni(n - 1);
}
```

Nije teško zaključiti da će iterativna verzija funkcije za računanje faktorijela za negativne argumente vratiti rezultat 1 (jer se “**for**“ petlja neće izvršiti niti jedanput), dok će rekursivna verzija “srušiti” program zagušenjem mašinskog steka (s obzirom da iz rekurzije neće biti izlaza). Očigledno, niti jedno niti drugo rješenje nije zadovoljavajuće. Ono što nipošto ne smijemo napraviti je ubacivanje ispisu poruka upozorenja unutar same funkcije, kao npr. u sljedećoj funkciji:

```
long int Faktorijel(int n) {
    if(n < 0)
        cout << "Faktorijel nije definiran za negativne argumente!\n";
    else {
        long int p = 1;
        for(int i = 1; i <= p; i++) p *= i;
        return p;
    }
}
```

Naime, u ovom slučaju onaj ko poziva funkciju ne bi imao nikakvu kontrolu nad ispisom poruke o greški: ona bi se ispisivala *uvijek na isti način* (kad god je argument negativan), i *pri svakom pozivu funkcije* sa neispravnim argumentima. Na primjer, u sljedećem izrazu

```
rezultat = Faktorijel(a) + Faktorijel(b) - Faktorijel(c);
```

u slučaju da sva tri argumenta “a”, “b” i “c” imaju negativne vrijednosti, poruka o greški bi se ispisala tri puta, što vjerovatno nije ono što bismo željeli. Može se postaviti pitanje zašto bismo uopće provjeru ispravnosti argumenata prepuštali funkciji, a ne onome ko *poziva funkciju*, kao npr. u sljedećem programskom isječku:

```
int n;
cin >> n;
if(n < 0)
    cout << "Faktorijel nije definiran za negativne argumente!\n";
else
    cout << "Faktorijel od " << n << " iznosi " << Faktorijel(n) << endl;
```

U navedenom primjeru, ovakvo rješenje u potpunosti zadovoljava. Međutim, ovo je ipak posve elementaran primjer. Generalno posmatrano, postoji barem nekoliko razloga zbog kojih bi u općem slučaju testiranje ispravnosti argumenata trebalo biti povjereni *funkciji*, a ne *onome ko funkciju poziva*. Prvo, uvjeti pod kojim funkcija nije definirana često nisu posve jednostavni. Na primjer, sljedeća funkcija, koja računa površinu trougla sa stranicama “a”, “b” i “c” korištenjem Heronovog obrasca, nije definirana ukoliko “a”, “b” i “c” ne predstavljaju stranice trougla, tj. ukoliko je ispunjen jedan od uvjeta “ $a > b + c$ ”, “ $b > a + c$ ” ili “ $c > a + b$ ”:

```
double PovrsinaTrougla(double a, double b, double c) {
    double s = (a + b + c) / 2;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}
```

Stoga, ukoliko bi se ovakva funkcija morala pozivati na mnogo mjesta u programu, provjera ispravnosti argumenata prilikom svakog poziva mogla bi biti veoma zamorna. Drugo, česti su slučajevima u kojima nije lako (ili čak nije ni moguće) prostim posmatranjem argumenata utvrditi da li je funkcija definirana ili ne, pogotovo ukoliko funkcija nije data jednostavnom formulom, nego se njena vrijednost izračunava prema nekom složenijem algoritmu. Posmatrajmo, na primjer, sljedeću funkciju dva realna argumenta “a” i “b”:

```
double Tajanstvena(double a, double b) {
    double s(0);
    for(int i = 0; i < a; i++) s += (a + i) / (a + s - b * i);
    return s;
}
```

Ova funkcija očigledno nije definirana za one vrijednosti argumenata “a” i “b” kod kojih će se u toku računanja u nazivniku izraza unutar “**for**“ petlje pojaviti nula. Postoji beskonačno mnogo parova argumenata “a” i “b” za koje se to može dogoditi (npr. za $a = 3$ i $b = 4$ do dijeljenja nulom dolazi u drugom prolasku kroz petlju), ali je ovakve situacije nemoguće uočiti unaprijed, već će one biti otkrivene tek u toku izračunavanja. Stoga je za ovakve funkcije testiranje ispravnosti argumenata *nemoguće* prepustiti pozivaocu funkcije, već to mora da uradi sama funkcija. Na kraju, formalno gledano, definiciono područje bilo koje funkcije je *svojstvo same funkcije*, a ne *onoga ko funkciju poziva*, stoga je i logično da se o ispravnosti argumenata brine upravo sama funkcija. Ipak, treba napomenuti da je u svakom slučaju onaj ko poziva funkciju odgovoran da utvrdi da li je funkcija ispravno obavila svoj zadatak i da ispiše obavještenje u slučaju da nije (to ne treba i ne smije da bude zadatak same funkcije, jer pozivaoc funkcije treba imati punu mogućnost odlučivanja kako reagirati u slučaju greške).

Jedna mogućnost prepuštanja brige o argumentima funkciji je uvođenje konvencije po kojoj funkcija u slučaju da nije definirana treba kao rezultat vratiti neku *unaprijed dogovorenu vrijednost* koja služi kao indikator greške. Na primjer, kako je površina trougla uvijek pozitivna, možemo se dogovoriti da funkcija

“PovrsinaTrougla“ vrati kao rezultat vrijednost “-1” u slučaju da argumenti ne predstavljaju stranice trougla, kao u sljedećoj definiciji:

```
double PovrsinaTrougla(double a, double b, double c) {
    if(a > b + c || b > a + c || c > a + b) return -1;
    double s = (a + b + c) / 2;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}
```

U ovom slučaju, onaj ko poziva funkciju može znati da li je funkcija odradila svoj posao ispravno tako što će provjeriti rezultat funkcije. Ukoliko je rezultat -1, to je sigurna indikacija da nešto nije u redu. Na primjer:

```
double a, b, c;
cout << "Unesite stranice trougla: ";
cin >> a >> b >> c;
double p = PovrsinaTrougla(a, b, c);
if(p == -1) cout << "Uneseni podaci ne obrazuju stranice trougla!\n";
else cout << "Povrsina trougla je " << p << endl;
```

Ovakav pristup može se koristiti u jednostavnijim primjerima, ali on posjeduje nekoliko ozbiljnih nedostataka. Prvo, često se javljaju funkcije koje kao rezultat mogu vratiti praktično *svaku realnu vrijednost*, tako da ne postoji nikakva specijalna “nemoguća” vrijednost koju bismo mogli vratiti kao indikator greške. Npr funkcija “Tajanstvena“ iz maloprije navedenog primjera je upravo takva. Jedno moguće rješenje je u slučaju greške vratiti neku *neočekivanu i veoma malo vjerovatnu* vrijednost, poput npr. 10^{100} . Na taj način bi funkcija “Tajanstvena“ mogla izgledati recimo ovako:

```
double Tajanstvena(double a, double b) {
    double s(0);
    for(int i = 0; i < a; i++)
        if(a + s == b * i) return 1e100;
        else s += (a + i) / (a + s - b * i);
    return s;
}
```

Testni primjer za ovu funkciju mogao bi izgledati ovako:

```
double a, b;
cin >> a >> b;
double f = Tajanstvena(a, b);
if(f == 1e100) cout << "Funkcija nije definirana!\n";
else cout << "Vrijednost funkcije je " << f << endl;
```

S obzirom da je zaista vrlo nevjerovatno (mada teoretski ne i nemoguće) da ova funkcija u normalnim okolnostima vrati 10^{100} kao rezultat, ovo polurješenje uglavnom zadovoljava. Međutim, rješenja poput ovog nisu pogodna u situaciji kada pozvanu funkciju treba upotrijebiti unutar nekog složenijeg izraza. Pretpostavimo, na primjer, da smo funkciju “Faktorijel“ modificirali tako da vraća rezultat -1 u slučaju da faktorijel nije definiran. Pretpostavimo dalje da ovakvu funkciju želimo da iskoristimo u programu koji računa broj mogućih kombinacija od k elemenata iz skupa koji ima n elemenata. Kao što je poznato, broj ovakvih kombinacija računa se po formuli $n!/[k!(n-k)!]$, što znači da bismo mogli koristiti sljedeći programski isječak:

```
int n, k;
cin >> n >> k;
cout << Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
```

Međutim, u ovom primjeru za potrebe testiranja grešaka ne bismo imali nikakve koristi od toga što funkcija “Faktorijel” vraća -1 u slučaju greške, jer se njeni pozivi nalaze kao sastavni dio složenijeg izraza. Da bismo imali ikakve koristi od ovakve povratne vrijednosti, morali bismo izraz u kojem se javlja funkcija “Faktorijel” razbiti na sastavne dijelove, kao u sljedećem primjeru:

```
int n, k;
long int f1, f2, f3;
cin >> n >> k;
f1 = Faktorijel(n); f2 = Faktorijel(k); f3 = Faktorijel(n - k);
if(f1 == -1 || f2 == -1 || f3 == -1) cout << "Nešto nije u redu!\n";
else cout << f1 / (f2 * f3);
```

Neelegantnost ovog primjera je više nego očigledna. Također, još jedan nedostatak rješenja zasnovanih na vraćanju specijalnih vrijednosti u slučaju grešaka je što su oni neprimjenljivi u funkcijama koje *ne vraćaju nikakvu vrijednost*. Ovaj problem rješava se lako prepravkom funkcija koje ne vraćaju vrijednost u funkcije koje vraćaju logičku vrijednost “**true**” ili “**false**” u zavisnosti da li je funkcija uspješno odradila svoj posao ili nije (ovakva rješenja smo do sada u više navrata koristili).

Zbog mnoštva pomenutih nedostataka, strategija vraćanja specijalnih vrijednosti kao rezultata u slučaju greške može se koristiti samo u jednostavnim slučajevima. Pogodnija strategija je *upotreba globalnih promjenljivih kao indikatora grešaka*. Sve do uvođenja koncepta *izuzetaka* (ili *iznimaka*) u jezik C++, ovo je bila najviše upotrebljavana metoda za obradu grešaka, i općenito neočekivanih situacija (pored toga, ovo je ujedno bio i jedini slučaj u kojem se preporučivala upotreba globalnih promjenljivih). Ideja ove metode je da se deklarira neka globalna promjenljiva, koja u normalnim okolnostima uvijek ima vrijednost “**false**” (ili 0), a da se u slučaju greške njena vrijednost postavi na “**true**” (ili neku nenultu vrijednost, čija tačna vrijednost može zavisiti od vrste greške). Kako je ta promjenljiva globalna, ona se može postaviti unutar funkcije, a čitati izvan nje, što znači da sama funkcija može indicirati da je došlo do greške, dok pozivaoc funkcije može lako provjeriti i utvrditi tu činjenicu. Na primjer, neka je data sljedeća globalna promjenljiva:

```
bool greska(false);
```

Tada bismo funkciju “Faktorijel” mogli prepraviti da izgleda ovako:

```
long int Faktorijel(int n) {
    long int p(1);
    if(n < 0) greska = true;
    else for(int i = 1; i <= p; i++) p *= i;
    return p;
}
```

Uz ovaku prepravku, imali bismo i sljedeći primjer za računanje broja kombinacija:

```
int n, k, komb;
cin >> n >> k;
komb = Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
if(greska) cout << "Nešto nije u redu!\n";
else cout << komb << endl;
```

Primijetimo da bez obzira na to što za indikaciju greške koristimo globalnu promjenljivu “greska”, funkcija *mora nešto vratiti i u slučaju greške* (u našem primjeru nije teško vidjeti da vraćena vrijednost u slučaju greške iznosi 1). Međutim, u ovom slučaju povratna vrijednost u slučaju greške *uopće nije bitna*, s obzirom da za testiranje da li je došlo do greške ne koristimo povratnu vrijednost, nego vrijednost

globalne promjenljive "greska". Doduše, u našem primjeru ne bi bilo dobro da povratna vrijednost funkcije u slučaju greške bude nula, jer bi u tom slučaju greška u funkciji "Faktorijel" doveća do novog problema – dijeljenja nulom. Mnoge funkcije koje su koristile ovu tehniku detekcije grešaka poštovale su konvenciju da u slučaju greške povratna vrijednost bude neki ogroman broj, poput 10^{100} (pored postavljanja globalnog indikatora greške).

Važno je napomenuti da je, prilikom korištenja ovog metoda za signalizaciju greške, onaj ko pozove funkciju dužan da, nakon što obradi grešku i preduzme odgovarajuću akciju (npr. zatraži novi unos podataka), vrati vrijednost globalne promjenljive "greska" na vrijednost "**false**" (ili 0 ukoliko se koristi cjelobrojni tip indikatora greške), jer će u suprotnom svaki sljedeći poziv funkcije biti protumačen kao neispravan (s obzirom da sama funkcija nikada ne vraća ovu promjenljivu na vrijednost "**false**" odnosno 0). Ova situacija je potpuno analogna situaciji da korisnik nakon obrade greške prilikom unosa podataka sa ulaznog toka mora pozvati funkciju "`cin.clear`" da obriše indikator greške ulaznog toka, o čemu smo ranije govorili.

Opisani metod, ukoliko se dosljedno poštuje, u praksi može funkcionirati prilično dobro. Međutim, osnovni nedostatak ovog koncepta je upotreba globalnih promjenljivih, koje su po svojoj prirodi vidljive i kome treba da budu, i kome ne treba da budu vidljive, čime se narušava koncept modularnosti, po kojem bi svaki identifikator trebao da bude vidljiv samo onome ko taj identifikator treba da koristi. Stoga, postoji velika mogućnost da dođe do nehotične pogrešne upotrebe promjenljive koja služi kao indikator greške. Dalje, lako se može desiti da pozivalac funkcije zaboravi da obriše indikator greške nakon obrade greške, što lako može dovesti do kasnijih problema. Konačno, ovaj metod, isto kao i prethodni metod, posjeduje veoma ozbiljan problem koji se ne smije zanemariti. Naime, opisani metodi *ne prisiljavaju korisnika funkcije da testira da li je došlo do greške!* Drugim riječima, ukoliko programer nakon poziva funkcije ne ispita da li je došlo do greške (bilo zbog previda, bilo zbog ljenosti, smatrajući da je vjerovatno sve u redu), u slučaju greške program će se nastaviti kao da se ništa nije dogodilo, ali prihvatajući neku besmislenu vrijednost kao rezultat. Posljedice naravno mogu biti posve nepredvidljive i mogu se manifestirati tek nakon mnogo vremena od nastanka greške. Pomenuti faktor ljenosti ni u kom slučaju nije zanemarljiv. Naime, poznato je da veliki broj standardnih funkcija iz jezika C koristi opisane metode za indikaciju greške (bilo putem specijalne povratne vrijednosti, bilo putem globalne varijable za indiciranje greške, koja se tipično zove "`errno`"), ali je također poznato da veoma mali broj programera *testira* indikatore greške nakon pozivanja tih funkcija, smatrajući, često posve neopravdano, da nema nikakvog razloga da dođe do greške prilikom izvršavanja pozvane funkcije.

Praktično svi do sada opisani problemi riješeni su kada je u jezik C++ uveden koncept *izuzetaka* (ili *iznimaka*). Izuzeci (engl. *exceptions*) su način reagiranja funkcije na nepredviđene okolnosti, npr. na argumente za koje funkcija nije definirana. U slučaju da funkcija ustanovi da ne može odraditi svoj posao, ili da ne može vratiti nikakvu smislenu vrijednost (npr. stoga što nije definirana za navedene argumente), funkcija umjesto da vrati vrijednost treba da *baci izuzetak* (engl. *throw exception*). Obratimo pažnju na terminologiju: vrijednosti funkcije se *vraćaju* iz funkcije, a izuzeci se *bacaju*. Za bacanje izuzetaka koristi se naredba "**throw**", koja ima sličnu sintaksu kao naredba "**return**": iza nje slijedi izraz koji identificira izuzetak koji se baca (kasnije ćemo vidjeti smisao ove identifikacije). Na primjer, funkcija "Faktorijel" prilagođena za bacanje izuzetaka mogla bi izgledati ovako:

```
long int Faktorijel(int n) {
    if(n < 0) throw n;
    long int p(1);
    for(int i = 1; i <= p; i++) p *= i;
    return p;
}
```

Razmotrimo sada šta se dešava ukoliko pozvana funkcija baci izuzetak (npr. ukoliko ovako napisanu funkciju "Faktorijel" pozovemo sa negativnim argumentom). Ukoliko ne preduzmemu nikakve izmjene u ostatku programa, nailazak na naredbu "throw" prosto će prekinuti izvršavanje programa, odnosno prva nepredviđena situacija doveće do prekida programa. Ovo sigurno nije ono što želimo. Međutim, izuzeci se uvijek bacaju sa ciljem da budu uhvaćeni (engl. *catch*). Za "hvatanje" izuzetaka koriste se ključne riječi "try" i "catch" koje se uvijek koriste zajedno, na način opisan sljedećom blokovskom konstrukcijom, koja donekle podsjeća na "if" – "else" konstrukciju:

```
try {
    Pokušaj
}
catch(formalni_parametar) {
    Hvatanje_izuzetka
}
```

Pri nailasku na ovu konstrukciju, prvo se izvršava skupina naredbi označena sa "Pokušaj". Ukoliko prilikom izvršavanja ovih naredbi ne dođe ni do kakvog bacanja izuzetaka, tj. ukoliko se sve naredbe okončaju na regularan način, skupina naredbi označena sa "Hvatanje_izuzetka" neće se uopće izvršiti, odnosno program će nastaviti izvršavanje iza zatvorene vitičaste zagrade koja označava kraj ovog bloka (tj. iza cijele "try" – "catch" konstrukcije). Međutim, ukoliko prilikom izvršavanja naredbi u bloku "Pokušaj" dođe do bacanja izuzetka (npr. pozove se neka funkcija koja baci izuzetak), izvršavanje naredbi koje eventualno slijede u bloku "Pokušaj" se prekida, a izvršavanje programa se nastavlja od prve naredbe u skupini "Hvatanje_izuzetka". Iza naredbe "catch" u zagradama se mora nalaziti deklaracija jednog formalnog parametra, koji će prihvati vrijednost bačenu naredbom "throw". Na taj način, naredbom "throw" može se "baciti" vrijednost koja će biti "uhvaćena" u naredbi "catch". Ta vrijednost može npr. označavati šifru greške (u slučaju da je moguće više različitih vrsta grešaka), a naredbe u bloku "Hvatanje_izuzetka" mogu u slučaju potrebe iskoristiti vrijednost ovog parametra da saznaju šifru greške, i da u zavisnosti od šifre preduzmu različite akcije. Sljedeći primjer ilustrira kako se izuzetak bačen iz funkcije "Faktorijel" može uhvatiti na mjestu poziva (npr. iz glavnog programa):

```
try {
    int n, k;
    cin >> n >> k;
    cout << Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
}
catch(int e) {
    cout << "Greška: faktorijel od " << e << " nije definiran!\n";
}
```

U razmotrenom primjeru treba obratiti pažnju na nekoliko činjenica. Prvo, promjenljive "n" i "k" su definirane kao *lokalne promjenljive* u "try" bloku, i kao takve, ona ne postoje izvan tog bloka. Stoga se one ne mogu koristiti niti unutar "catch" bloka. S druge strane, funkcija "Faktorijel" je, prilikom bacanja izuzetka, kao identifikaciju izuzetka iskoristila upravo svoj formalni parametar "n", tako da naredbe u "catch" bloku mogu lako saznati problematičnu vrijednost argumenta koja je dovela do greške. Na primjer, ukoliko pri testiranju ovog programskog fragmenta zadamo ulazne podatke $n=4$ i $k=6$, na ekranu ćemo dobiti ispis poruke "Greška: faktorijel od -2 nije definiran!". Zaista, pri računanju broja kombinacija za $n=4$ i $k=6$, pojavljuje se potreba za računanjem izraza $4!/[6! \cdot (-2)!]$ u kojem je problematičan upravo faktorijel od -2.

Prilikom bacanja izuzetaka, iza naredbe "throw" može se naći izraz *bilo kojeg tipa*. Pri tome, da bi izuzetak bio uhvaćen, tip bačenog izuzetka treba *striktno odgovarati* tipu formalnog parametra navedenog unutar "catch" naredbe, odnosno ne vrše se automatske pretvorbe tipa (npr. izuzetak tipa "int" neće biti

uhvaćen u “**catch**“ naredbi čiji je formalni parametar tipa “**double**“). Jedini izuzetak od ovog pravila su pretvorbe pokazivača i pretvorbe vezane za nasljeđivanje, o čemu ćemo govoriti u kasnijim poglavljima. Sljedeći primjer demonstrira funkciju “Faktorijel“ koja baca izuzetak tipa stringovne konstante:

```
long int Faktorijel(int n) {
    if(n < 0) throw "Faktorijel negativnih brojeva nije definiran";
    long int p(1);
    for(int i = 1; i <= p; i++) p *= i;
    return p;
}
```

Za hvatanje takvih izuzetaka, odgovarajući tip formalnog parametra u naredbi “**catch**“ mora biti “**const char []**“ (kvalifikator “**const**“ je, pri tome, *obavezan*). Stoga, eventualno bačeni izuzetak iz prethodnog primjera možemo uhvatiti kao u sljedećem primjeru:

```
try {
    int n;
    cin >> n;
    cout << "Faktorijel od " << n << " iznosi " << Faktorijel(n) << endl;
}
catch(const char poruka[]) {
    cout << poruka << endl;
}
```

Često se dešava da jedna funkcija poziva drugu, druga treću, treća četvrtu itd. Prepostavimo na primjer, da neka od funkcija u lancu pozvanih funkcija (npr. treća) baci izuzetak. Prirodno je postaviti pitanje *gdje će takav izuzetak biti uhvaćen*. Da bismo odgovorili na ovo pitanje, razmotrimo *šta se tačno dešava* prilikom bacanja izuzetaka. Ukoliko se prilikom izvršavanja neke funkcije nađe na naredbu “**throw**”, njeno izvršavanje se prekida (osim u slučaju da se naredba “**throw**“ nalazila unutar “**try**“ bloka, što je specijalni slučaj koji ćemo razmotriti kasnije) i tok programa se vraća na mjesto poziva funkcije. Pri tome se dešavaju iste stvari kao i pri regularnom završetku funkcije (npr. sve lokalne promjenljive funkcije bivaju uništene). Dalji tok programa sada bitno ovisi da li je funkcija koja je bacila izuzetak pozvana iz unutrašnjosti nekog “**try**“ bloka ili nije. Ukoliko je funkcija pozvana iz “**try**“ bloka, izvođenje naredbe unutar koje se nalazi poziv funkcije i svih naredbi koje eventualno slijede iza nje u “**try**“ bloku se prekida, i tok programa se preusmjerava na početak odgovarajućeg “**catch**“ bloka, kao što je već opisano. Međutim, ukoliko funkcija koja je bacila izuzetak nije pozvana iz nekog “**try**“ bloka, izvršavanje funkcije iz koje je problematična funkcija pozvana biće također prekinuto, kao da je upravo ona bacila izuzetak, i kontrola toka programa se prenosi na mjesto poziva te funkcije. U slučaju da je pozvana funkcija bila pozvana iz “**main**“ funkcije (izvan “**try**“ bloka), prekinuće se sam program, a ista stvar bi se desila i u slučaju da se izuzetak baci direktno iz “**main**“ funkcije. Postupak se dalje ponavlja sve dok izuzetak ne bude uhvaćen u nekoj od funkcija u lancu pozvanih funkcija, nakon čega se tok programa nastavlja u pripadnom “**catch**“ bloku (koji odgovara mjestu gdje je izuzetak uhvaćen). Ukoliko se u toku ovog “razmotavanja” poziva dode i do same “**main**“ funkcije, a izuzetak ne bude uhvaćen ni u njoj, program će biti prekinut.

Iz gore izloženog vidimo da se svaki izuzetak koji ne bude uhvaćen unutar neke funkcije prosljeđuje dalje na mjesto poziva funkcije, sve dok ne bude eventualno uhvaćen. Po tome se mehanizam bacanja izuzetaka bitno razlikuje od mehanizma vraćanja vrijednosti iz funkcija. Opisani mehanizam prosljeđivanja izuzetaka je veoma prirodan i logičan, mada pri prvom čitanju može djelovati pomalo zbunjujuće. Situacija postaje mnogo jasnija nakon što razmotrimo sljedeći primjer, u kojem “**main**“ funkcija poziva funkciju “**Kombinacije**”, koja opet poziva funkciju “**Faktorijel**“ koja može eventualno baciti izuzetak:

```

#include <iostream>
using namespace std;

long int Faktorijel(int n) {
    if(n < 0) throw n;
    long int p(1);
    for(int i = 1; i <= p; i++) p *= i;
    return p;
}

int Kombinacije(int n, int k) {
    return Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
}

int main() {
    try {
        int n, k;
        cin >> n >> k;
        cout << Kombinacije(n, k);
    }
    catch(int e) {
        cout << "Greška: Faktorijel od " << e << " nije definiran!";
    }
    return 0;
}

```

Pretpostavimo da smo pokrenuli ovaj program i zadali ulazne podatke $n=4$ i $k=6$. Funkcija “main” će prvo pozvati funkciju “Kombinacije” sa parametrima 4 i 6, koja s druge strane tri puta poziva funkciju “Faktorijel” sa parametrima 4, 6 i –2 respektivno. Prva dva poziva završavaju uspješno. Međutim, u trećem pozivu dolazi do bacanja izuzetka, i tok programa se iz funkcije “Faktorijel” vraća u funkciju “Kombinacije”. Kako unutar funkcije “Kombinacije” funkcija “Faktorijel” nije pozvana iz “try” bloka, prekida se i izvršavanje funkcije “Kombinacije”, i tok programa se vraća u “main” funkciju. Kako je u “main” funkciji funkcija “Kombinacije” pozvana iz “try” bloka, izvršavanje se nastavlja u odgovarajućem “catch” bloku, u kojem se prihvata izuzetak bačen iz funkcije “Faktorijel”, i ispisuje poruka “Greška: faktorijel od –2 nije definiran”. Drugim riječima, izuzetak bačen iz funkcije “Faktorijel” uhvaćen je tek u funkciji “main”, nakon što je prethodno “prošao” neuhvaćen kroz funkciju “Kombinacije”. Sasvim drugu situaciju imamo u sljedećem primjeru, u kojem se izuzetak hvata već u funkciji “Kombinacije”:

```

#include <iostream>
using namespace std;

long int Faktorijel(int n) {
    if(n < 0) throw n;
    long int p(1);
    for(int i = 1; i <= p; i++) p *= i;
    return p;
}

int Kombinacije(int n, int k) {
    try {
        return Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
    }
    catch(int e) {
        return 0;
    }
}

```

```

}

int main() {
    int n, k;
    cin >> n >> k;
    cout << Kombinacije(n, k);
    return 0;
}

```

Pretpostavimo ponovo isti početni scenario, tj. da je program pokrenut sa ulaznim podacima $n=4$ i $k=6$. Funkcija "main" ponovo poziva funkciju "Kombinacije" koja dalje tri puta poziva funkciju "Faktorijel", pri čemu prilikom trećeg poziva dolazi do bacanja izuzetka. Tok programa se iz funkcije "Faktorijel" vraća u funkciju "Kombinacije". Međutim, u ovom primjeru funkcija "Faktorijel" je pozvana iz "try" bloka unutar funkcije "Kombinacije", tako da izuzetak biva uhvaćen već u funkciji "Kombinacije". Stoga se tok programa prenosi u odgovarajući "catch" blok unutar funkcije "Kombinacije". Tamo se nalazi naredba "return" kojom se vrši *regularni povratak* iz funkcije "Kombinacije", pri čemu se kao njen rezultat vraća vrijednost 0 (navедена kao argument naredbe "return"). Stoga će u glavnom programu kao rezultat biti ispisana upravo nula (ovakvo ponašanje ima smisla izvesti, jer je razumljivo da je broj kombinacija od šest elemenata koje se mogu napraviti izborom iz četveročlanog skupa upravo nula). Dakle, funkcija "Kombinacije" se završava *normalno*, bez obzira što je ona uhvatila izuzetak koji je bačen iz funkcije "Faktorijel" (možemo smatrati da uhvaćeni izuzetak prestaje biti izuzetak). Također, primijetimo da formalni parametar "e" u "catch" bloku nije iskorišten ni za šta, tj. prosto je *ignoriran*. To smijemo uraditi kada god nas ne zanima kako je došlo do izuzetka, već samo da je do njega *došlo*. Jezik C++ dozvoljava da se u takvim slučajevima umjesto formalnog argumenta pišu tri tačke, kao u sljedećem isječku:

```

try {
    return Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
}
catch(...) {
    return 0;
}

```

Ovakav "catch" blok uhvatiće bilo koji izuzetak, bez obzira na njegov tip. Također je dozvoljeno unutar zagrada iza naredbe "catch" pisati samo *ime tipa*, bez navođenja imena formalnog parametra (npr. "catch(int)"). Ovu varijantu možemo koristiti ukoliko želimo hvatati *tačno određenog tipa*, ali nas ne zanima *vrijednost bačenog izuzetka*.

Iz izloženog razmatranja lako se može zaključiti da se izuzetak koji bude uhvaćen u nekoj od funkcija ne prosljeđuje dalje iz te funkcije, nego se smatra da se funkcija koja je uhvatila izuzetak završava regularno. Međutim, sasvim je legalno da funkcija koja uhvati izuzetak, nakon što eventualno izvrši neke specifične akcije u skladu sa nastalom situacijom, ponovo baci izuzetak (isti ili neki drugi), kao u sljedećem primjeru:

```

#include <iostream>
using namespace std;

long int Faktorijel(int n) {
    if(n < 0) throw "Pogrešan argument u funkciji Faktorijel";
    long int p(1);
    for(int i = 1; i <= p; i++) p *= i;
    return p;
}

int Kombinacije(int n, int k) {
    try {

```

```

        return Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
    }
    catch(...) {
        throw "Pogrešni argumenti u funkciji Kombinacije";
    }
}

int main() {
    try {
        int n, k;
        cin >> n >> k;
        cout << Kombinacije(n, k);
    }
    catch(const char poruka[]) {
        cout << poruka << endl;
    }
    return 0;
}

```

U ovom primjeru, u slučaju pogrešnih argumenata, funkcija “Faktorijel” baca izuzetak koji biva uhvaćen u funkciji “Kombinacije”. Međutim, ova funkcija u svom “**catch**“ bloku baca novi izuzetak, koji biva uhvaćen unutar funkcije “main”. Krajnja posljedica će biti da će biti ispisana poruka “*Pogrešni argumenti u funkciji Kombinacije*”, na osnovu čega se jasno vidi da je u “*main*“ funkciji uhvaćen (novi) izuzetak koji je bacila funkcija “Kombinacije”, a ne prvobitno bačeni izuzetak koji je bacila funkcija “Faktorijel” (i koji je uhvaćen u funkciji “Kombinacije”).

Nekada se može desiti da je potrebno da funkcija uhvati izuzetak, izvrši neke akcije koje su neophodne za obradu izuzetka, a da nakon toga isti izuzetak proslijedi dalje (funkciji koja ju je pozvala). Naravno, nikakav problem nije da funkcija baci isti izuzetak koji je i uhvatila. Međutim, C++ za tu svrhu dozvoljava i korištenje naredbe “**throw**“ bez navođenja nikakvog izraza iza nje. U tom slučaju se podrazumijeva da funkcija proslijedi dalje (tj. ponovo baca) isti izuzetak koji je upravo uhvatila. Ovo je jedini slučaj u kojem se naredba “**throw**“ može koristiti bez svog argumenta. Naravno, takva “**throw**“ naredba može se nalaziti samo unutar nekog “**catch**“ bloka.

Interesantno je da formalni parametar u “**catch**“ naredbi može biti *i referencia* (tj. izuzeci se također mogu prenositi po referenci). U tom slučaju, “**catch**“ blok može *izmjeniti* prihvaćeni izuzetak, prije nego što ga proslijedi dalje korištenjem naredbe “**throw**“ bez argumenta (u slučaju da nije korištena referenca, dalje bi bio proslijeden isti izuzetak koji je uhvaćen bez obzira na eventualne promjene formalnog parametra “**catch**“ bloka, jer bi formalni parametar predstavljao *kopiju* bačenog izuzetka a ne njega samog). Ovo je jedan od mogućih (i ne osobito često korištenih) razloga za korištenje referenci kao formalnih parametara u “**catch**“ bloku. Drugi mogući razlozi su sprečavanje kopiranja koja mogu nastati ukoliko se kao izuzeci bacaju masivni objekti (npr. vektori ili dinamički stringovi), kao i za podršku polimorfne obrade izuzetaka (o polimorfizmu ćemo govoriti u kasnijim poglavljima).

Mada se izuzeci obično bacaju iz neke od funkcija koje se pozivaju iz nekog od “**try**“ blokova bilo direktno ili indirektno (tj. posredstvom drugih funkcija), dozvoljeno je i da se izuzeci bacaju nevezano od poziva funkcija. Tako je, na primjer, sljedeća konstrukcija savršeno legalna:

```

try {
    double x;
    cin >> x;
    if(x == 0) throw "Nula nema recipročnu vrijednost!\n";
    cout << "Recipročna vrijednost broja " << x << " glasi " << 1/x;
}

```

```

catch(const char poruka[]) {
    cout << poruka;
}

```

Ipak, ovu mogućnost treba izbjegavati, jer se ovako pisane konstrukcije uvijek mogu napisati korištenjem obične “**if**” – “**else**“ konstrukcije, a bacanje izuzetaka je prilično zahtjevno po pitanju računarskih resursa. Na primjer, prethodni isječak se mogao mnogo prirodnije napisati na sljedeći način;

```

double x;
cin >> x;
if(x == 0) cout << "Nula nema recipročnu vrijednost!\n";
else cout << "Recipročna vrijednost broja " << x << " glasi " << 1/x;

```

Bacanje izuzetka direktno iz “**try**“ konstrukcije može eventualno imati smisla u primjerima poput sljedećeg, u kojem se izuzetak baca u slučaju da se ispostavi da prilično složeni račun koji se provodi (za ovaj primjer je posve nebitno šta ovaj račun radi) ne može da se obavi zbog dijeljenja nulom:

```

try {
    double p(0);
    for(int i = 0; i < a; i++) {
        double q(0);
        for(int j = 0; j < b; j++) {
            if(a + s == b * i) throw 0;
            q += (a + j) / (a + s - b * i);
        }
        p += q * i;
    }
    cout << "Rezultat je " << p << endl;
}
catch(...) {
    cout << "Problem: došlo je do dijeljenja nulom!\n";
}

```

Naravno, i ovaj primjer bi se mogao da prepravi da ne koristi bacanje izuzetaka, samo što bismo u tom slučaju imali malo više problema da se “ispetljamo” iz dvostruko ugnježdene petlje u slučaju da detektiramo dijeljenje nulom. Ovako, pomoću bacanja izuzetka, čim uočimo problem, tok programa se odmah nastavlja unutar “**catch**“ bloka. Bez obzira na sve pogodnosti koje nude izuzeci, u posljednje vrijeme se sve više susreću programi koji sa upotrebom izuzetaka pretjeruju, i koriste bacanje izuzetaka za rješavanje situacija koje su se mogle prirodnije riješiti običnom upotrebom “**if**” – “**else**“ ili “**switch**” – “**case**“ konstrukcija. Kao što je već rečeno, takva metodologija, mada izgleda da mnogima djeluje “modernom” i “savremenom”, ne smatra se dobrom programskom praksom.

Treba napomenuti da za izuzetke vrijedi pravilo “bolje spriječiti nego liječiti”, s obzirom da postupak bacanja izuzetaka opterećuje računarske resurse (tj. izuzeci su “bolest” koju, ukoliko je moguće, treba “spriječiti”, iako postoje i dosta kvalitetne metode “liječenja”). Na primjer, bez obzira što smo funkciju “Faktorijel” napravili tako da baca izuzetak u slučaju neispravnog argumenta, u slučaju da apriori znamo da argument nije ispravan, bolje je funkciju uopće ne pozivati, nego pozvati je, i hvatati bačeni izuzetak. Na primjer, uvijek je bolje pisati konstrukciju poput

```

int n;
cin >> n;
if(n < 0) cout << "Faktorijel nije definiran!\n";
else cout << n << "!" << Faktorijel(n) << endl;

```

nego konstrukciju

```

try {
    int n;
    cin >> n;
    cout << n << " ! = " << Faktorijel(n) << endl;
}
catch(...) {
    cout << "Faktorijel nije definiran!\n";
}

```

To ne znači da funkciju “Faktorijel” ne treba praviti tako da baca izuzetak. Dobro napisana funkcija uvijek treba da zna kako da se “odbrani” od nepredviđenih situacija. Međutim, to što ona zna kako da se “odbrani”, ne treba značiti da korisnik funkcije treba da je “provocira” dovodeći je u situaciju u kojoj se unaprijed zna da će morati da se “brani”. Izuzeci treba da služe za reagiranje u nepredvidljivim situacijama. Kad god je posve jasno da će do izuzetka doći, bolje je preduzeti mjere da do njega ne dođe, nego čekati njegovu pojavu i hvatati ga.

Bilo koji “**try**” blok može imati više pridruženih “**catch**” blokova, koji hvataju različite tipove izuzetaka. Na primjer, konstrukcija poput sljedeće je posve ispravna:

```

try {
    Neke_naredbe
}
catch(int broj) {
    Obrada_1
}
catch(const char tekst[]) {
    Obrada_2
}
catch(...) {
    Obrada_3
}

```

Ukoliko prilikom izvršavanja skupine naredbi “*Neke_naredbe*” dođe do bacanja izuzetka cjelobrojnog tipa, on će biti uhvaćen u prvom “**catch**” bloku, odnosno izvršiće se skupina naredbi “*Obrada_1*”. Ukoliko se baci izuzetak tipa stringovne konstante, on će biti uhvaćen u drugom “**catch**” bloku, dok će treći “**catch**” blok uhvatiti eventualno bačene izuzetke nekog drugog tipa. Kada se baci izuzetak, “**catch**” blokovi se ispituju u redoslijedu kako su navedeni, i prvi blok koji može uhvatiti izuzetak čiji se tip slaže sa navedenim tipom preuzima njegovu obradu, dok se ostali “**catch**” blokovi ne izvršavaju. Zbog toga, se eventualni “**catch**” blok sa tri tačkice umjesto formalnog parametra smije nalaziti samo na posljednjem mjestu u nizu ovakvih blokova, jer u suprotnom ni jedan “**catch**” blok koji slijedi ne bi nikada bio izvršen (s obzirom da “**catch**” blok sa tri tačkice hvata izuzetke bilo kojeg tipa). Ovo pravilo je čak regulirano sintaksom jezika C++, tako da će kompjajler prijaviti sintaksnu grešku ukoliko “**catch**” blok sa tri tačkice umjesto formalnog parametra nije na posljednjem mjestu u nizu “**catch**” blokova.

Kao što je moguće za funkciju specificirati tip povratne vrijednosti, tako je moguće (ali ne i obavezno) specificirati tipove izuzetaka koje neka funkcija eventualno baca (što korisniku funkcije omogućava da bez analize funkcije zna koje tipove izuzetaka eventualno treba hvatati). Tako, na primjer, sljedeća definicija funkcije

```

int NekakvaFunkcija(int n) throw(int, const char []) {
    if(n < 0) throw n;
    if(n % 2 == 0) throw "Argument ne smije biti paran!";
    return (x + 1) % 2;
}

```

eksplicitno naglašava da funkcija “NekakvaFunkcija” može baciti isključivo izuzetke tipa “**int**” ili “**const char []**” (pokušaj bacanja izuzetaka drugačijeg tipa smatra se fatalnom greškom koja dovodi do prekida rada programa). Specifikacija tipova izuzetaka može se navesti i u prototipu funkcije.

Izlaganje o izuzecima završićemo napomenom da mnoge ugrađene funkcije jezika C++ (naročito one koje su u standard jezika C++ ušle relativno skoro) također mogu bacati izuzetke, koji se hvataju isto kao i izuzeci bačeni iz korisnički definiranih funkcija. Nažalost, praktično sve starije funkcije koje su u jeziku C++ naslijedene iz jezika C, uključujući i gotovo sve klasične matematičke funkcije (poput “sqrt”, “log”, “pow” itd.) *ne bacaju izuzetke* u slučaju grešaka, bar ne po standardu jezika C++ (iako ima kompjajlera u kojima i ove funkcije bacaju izuzetke). Prema standardu, ove funkcije signaliziraju grešku postavljanjem globalne promjenljive “**errno**”. Ovo je šteta, jer greške koje mogu nastati uslijed pogrešnih parametara ovih funkcija ne mogu biti tretirane na isti način kao greške koje nastaju u korisnički definiranim funkcijama. Ovaj problem se lako može riješiti tako što ćemo za svaku takvu funkciju koju namjeravamo koristiti napraviti vlastitu verziju, koja u slučaju greške baca izuzetak, a u suprotnom samo poziva ugrađenu funkciju. Na primjer, možemo napisati vlastitu funkciju “**Sqrt**“ (sa velikim “S”) koja za negativne argumente baca izuzetak, a inače poziva ugrađenu funkciju “sqrt”:

```
double Sqrt(double x) {
    if(x < 0) throw "Greška: Korijen iz negativnog broja!";
    return sqrt(x);
}
```

Ako nakon toga u programu umjesto funkcije “sqrt“ budemo uvijek koristili isključivo funkciju “**Sqrt**”, moći ćemo za kontrolu i “hvatanje” grešaka i drugih izuzetnih situacija koristiti sve tehnike koje pruža mehanizam bacanja i hvatanja izuzetaka opisan u ovom poglavlju. Inače, funkcije koje u suštini samo pozivaju neku drugu funkciju sa ciljem ostvarivanja neke minorne izmjene njenog ponašanja nazivaju se *funkcije omotači* (engl. *function wrappers*). Tako je, u ovom primjeru, funkcija “**Sqrt**“ *omotač za ugrađenu funkciju “sqrt“*.

25. Pokazivači i pokazivačka aritmetika

U ranijim poglavljima upoznali smo se sa *referencama* koje predstavljaju objekte koji upućuju na neke druge objekte, i preko kojih se može ostvariti *indirektni pristup* objektima na koje oni upućuju. Pored referenci, koje su karakteristične isključivo za jezik C++, u jeziku C++ je podržana još jedna vrsta objekata za indirektan pristup drugim objektima naslijedena iz jezika C, koji se nazivaju *pokazivači* ili *pointeri*. Interna struktura reference i pokazivača je identična. I reference i pokazivači zapravo u sebi interno sadrže adresu objekta na koji upućuju (odnosno *pokazuju* kako se to kaže u slučaju kada koristimo pokazivače). Međutim, suštinska razlika između referenci i pokazivača sastoji se u činjenici da su reference izvedene na takav način da *u potpunosti oponašaju objekat na koji upućuju* (odnosno, svaki pokušaj pristupa referenci se automatski preusmjerava na objekat na koji referencia upućuje, dok je sam pristup internoj strukturi reference nemoguć). S druge strane, kod pokazivača je napravljena striktna razlika između pristupa *samom pokazivaču* (odnosno *adresi* koja je u njemu pohranjena) i pristupa *objektu na koji pokazivač pokazuje*. Drugim riječima, za pristup objektu na koji pokazivač pokazuje koristi se *drugačija sintaksa* u odnosu na pristup internoj strukturi pokazivača.

Treba napomenuti da su reference jednostavnije za upotrebu od pokazivača, ali su i podložne mnogo većim ograničenjima. Na primjer, reference od trenutka stvaranja uvijek upućuju na jedan te isti objekat, što nije moguće promijeniti sve dok referencia ne prestane postojati. S druge strane, pokazivači tokom svog života mogu pokazivati na različite objekte, što u kombinaciji sa tzv. *pokazivačkom aritmetikom*, koja će biti opisana nešto kasnije, omogućava vrlo efikasne manipulacije nad elementima nizova. Pokazivači također ne moraju pokazivati isključivo na već postojeće objekte, nego mogu pokazivati na proizvoljan dio memorije, što se obilato koristi za potrebe tzv. *dinamičke alokacije memorije* (o kojoj ćemo govoriti u narednim poglavljima). Dalje, nije moguće napraviti nizove referenci, ali je moguće imati nizove pokazivača, itd. Dakle, pokazivači pružaju veću fleksibilnost, ali je u radu sa njima mnogo lakše “zabrljati” nego pri radu sa referencama (s obzirom da imamo mogućnost direktnog pristupa njihovoj internoj strukturi). Stoga u jeziku C++ reference treba koristiti kad je god to moguće, a pokazivače samo u slučajevima kada nam je neophodna puna fleksibilnost koju oni pružaju. Vidjećemo da postoji zaista mnogo stvari koje je moguće uraditi samo pomoću pokazivača, a koje je nemoguće realizirati niti pomoću referenci, niti na bilo koji drugi način (koji ne koristi pokazivače).

Za razliku od referenci, pokazivače nije moguće objasniti bez nešto detaljnijeg objašnjenja o tome kako se promjenljive čuvaju u memoriji računara (između ostalog i zbog toga što kod pokazivača možemo *direktno pristupiti* informaciji o tome gdje se u memoriji čuva objekat na koji pokazivač pokazuje). Zbog toga ćemo prvo u osnovnim crtama razmotriti način smještanja promjenljivih u računarskoj memoriji. Poznato je da svaka promjenljiva koja se deklarira u programu tokom svog života zauzima određeni prostor u memoriji. Mjesto u memoriji koje je dodijeljeno nekoj promjenljivoj određeno je njenom *adresom*, koja je fiksna sve dok posmatrana promjenljiva postoji (ako zamislimo da je čitava memorija poput nekog niza, tada adrese možemo zamisliti kao indekse elemenata tog niza). Na primjer, pretpostavimo da imamo sljedeću deklaraciju:

```
int broj;
```

Prilikom nailaska na ovu deklaraciju, rezervira se odgovarajući prostor u memoriji za potrebe smještanja sadržaja promjenljive “broj”, koji naravno ima svoju adresu (koju ćemo nazvati prosto *adresa promjenljive “broj”*). Prepostavimo, na primjer, da je ovoj promjenljivoj dodijeljena adresa 4325. Tada imamo sljedeću memorijsku sliku:

Adrese	...	4322	4323	4324	4325	4326	4327	4328	4329	...
--------	-----	------	------	------	------	------	------	------	------	-----

--	--	--	--	--	--	--	--	--	--

broj

Ukoliko nakon ove deklaracije izvršimo dodjelu neke vrijednosti promjenljivoj “broj”, npr. dodjelom

```
broj = 56;
```

efekat ove dodjele biće smještanje vrijednosti 56 u memoriju na adresu 4325 dodijeljenu promjenljivoj “broj”, kao što je prikazano na sljedećoj slici:

Adrese	...	4322	4323	4324	4325	4326	4327	4328	4329	...
					56					

broj

U stvarnosti je situacija nešto složenija nego na prikazanoj slici. Naime, memorija se obično adresira po *bajtima*, tako da jedna memorijska lokacija može primiti samo jednobajtnu vrijednost, dok cijelobrojne promjenljive obično zahtijevaju više bajta za memoriranje sadržaja (danas najčešće 4). Stoga će promjenljiva “broj” u stvarnosti zauzeti ne samo jednu, nego više susjednih lokacija (npr. 4325, 4326, 4327 i 4328), pri čemu pod adresom promjenljive smatramo adresu *prve od zauzetih lokacija* (tj. 4325). Međutim, radi pojednostavljenja diskusije koja slijedi, mi ćemo sadržaj memorije crtati kao da jedna promjenljiva uvijek zauzima *jednu* memorijsku lokaciju. Ovo pojednostavljenje neće se bitno odraziti na suštinu izlaganja. Također, postoje računarske arhitekture kod kojih organizacija memorije nije *linearna* (tj. nalik na niz) nego ima složeniju strukturu, npr. poput strukture *matrice*, tako da adresa nije običan *broj* nego *par brojeva* u obliku (*red, kolona*). Ovakvu situaciju imamo, na primjer, kod Intel-ovih procesora pod starijim operativnim sistemima poput MS DOS-a i Windows-a 3.11, kod kojih je adresa zaista zadana kao par brojeva (koji se redom nazivaju *segment* i *offset*). U nastavku izlaganja ćemo uglavnom zanemariti eventualne komplikacije ove prirode i smatraćemo da je memorija *linearna*, tj. usvojićemo *linearni memorijski model* (pri čemu ćemo istaći eventualne poteškoće koje mogu nastati ukoliko to nije slučaj).

Razumije se da za pristup promjenljivoj “broj” nije uopće potrebno poznavati na kojoj se adresi u memoriji ona nalazi, jer njenom sadržaju uvijek možemo pristupiti koristeći njeno simboličko ime “broj” (kao što smo uvijek i činili). Međutim, reference i pokazivači kao svoj *sadržaj* imaju *adresu neke druge promjenljive* (ili općenito, neke druge l-vrijednosti). I dok je kod referenci ta činjenica potpuno zamaskirana i nevidljiva za korisnika (informacija o adresi vezanog objekta se implicitno koristi za automatsko preusmjeravanje pristupa referenci u pristup objektu na koji referencia upućuje), kod pokazivača je adresa objekta na koji pokazivač pokazuje direktno dostupna, dok se eventualno preusmjeravanje na objekat na koji pokazivač pokazuje ne vrši *automatski*, već ga moramo *eksplicitno zatražiti*.

Pokazivačke promjenljive se deklariraju tako da se ispred njihovog imena stavi znak “*” (zvjezdica), slično kao što se znak “&” koristi za deklaraciju referenci. Na primjer, sljedeća deklaracija deklarira promjenljivu “pokazivac” koja nije tipa “cijeli broj”, nego “pokazivač na cijeli broj”:

```
int *pokazivac;
```

Kao i svaka druga promjenljiva, i pokazivačka promjenljiva zauzima određeni prostor u memoriji. Ukoliko pretpostavimo da je promjenljivoj “pokazivac” dodijeljena npr. adresa 4329, dobijamo sljedeću memorijsku sliku:

<i>Adrese</i>	...	4322	4323	4324	4325	4326	4327	4328	4329	...
					56					

broj

pokazivac

Naravno, ovom deklaracijom još uvijek nije dodijeljena nikakva vrijednost promjenljivoj “pokazivac” (vidimo da za razliku od referenci, pokazivači mogu biti neinicijalizirani). Međutim, pokazivačkim promjenljivim ne smiju se neposredno dodjeljivati *brojčane vrijednosti*. Stoga komajler *neće dozvoliti* dodjelu poput sljedeće:

```
pokazivac = 4325;
```

Umjesto toga, pokazivačkim promjenljivim se mogu dodjeljivati samo *vrijednosti pokazivačke prirode* (tj. vrijednosti koje garantirano predstavljaju adrese nekih drugih objekata). Na primjer, pokazivaču je moguće dodjeliti adresu nekog objekta (npr. promjenljive) uz pomoć unarnog prefiksнog operatora “&”, koji možemo čitati “adresa od”. Tako, u sljedećem primjeru, promjenljivoj “pokazivac” dodjeljujemo adresu promjenljive “broj” (odnosno vrijednost “4325” u konkretnom primjeru situacije u memoriji):

```
pokazivac = &broj;
```

Nakon ove dodjele, situacija u memoriji je kao na sljedećoj slici:

<i>Adrese</i>	...	4322	4323	4324	4325	4326	4327	4328	4329	...
					56				4325	

broj

pokazivac

Treba napomenuti da bismo *identičnu situaciju u memoriji* imali u slučaju da je promjenljiva “pokazivac” referenca vezana na promjenljivu “broj”, ali korisnik toga ne bi bio svjestan. Na primjer, u tom slučaju pokušaj ispisu promjenljive “pokazivac” ne bi ispisao vrijednost “4325” nego “56” (adresa 4325 bila bi iskorištena samo kao informacija odakle treba uzeti podatak kojem se pristupa), odnosno ispis bi bio isti kao da smo ispisali direktno vrijednost promjenljive “broj”. Međutim, sa pokazivačima je situacija drugačija. Ukoliko bismo pokušali da ispišemo sadržaj maloprije deklarirane promjenljive “pokazivac” (koja je zaista deklarirana kao pokazivač), dobili bismo ispis *njenog pravog sadržaja*, odnosno adrese na kojoj je smještena promjenljiva “broj”. U navedenom primjeru, to bi bio broj 4325. Strogo gledano, ispis neće glasiti “4325” nego “0x10e5” zbog konvencije da se adrese uvijek ispisuju u *heksadekadnom brojnom sistemu*, a “0x” je prefiks za heksadekadni brojni sistem u jeziku C++ (dekadni prikaz bismo mogli dobiti eksplicitnom konverzijom u tip “**int**”, odnosno ispisom izraza poput “(**int**)pokazivac”). Naravno, tačan ispis striktno zavisi od toga gdje je promjenljiva “broj” zaista smještena u memoriji što se ne može unaprijed znati (značenje ispisu postaje još zapetljanije kod računarskih arhitektura kod kojih memorija nije linearna, u šta ovdje nećemo ulaziti). Zbog toga sama vrijednost pokazivačke promjenljive najčešće nije od neposredne koristi. Mnogo češće nam je potreban ne sam sadržaj pokazivačke promjenljive, već *sadržaj memorijske lokacije na koju pokazivačka promjenljiva pokazuje*. Ovaj sadržaj možemo dobiti primjenom unarnog operatora “*”, koji možemo čitati “sadržaj lokacije smještene u” (ili pojednostavljeni samo “sadržaj od”). Na primjer, sljedeća naredba

```
cout << *pokazivac;
```

ispisaće vrijednost “56”, jer je sadržaj lokacije na koju pokazuje promjenljiva “pokazivac” upravo 56.

Operator “`&`” naziva se još i operator *referenciranja*, dok se operator “`*`” naziva i operator *derefenciranja*. Operator referenciranja može se primijeniti isključivo na l-vrijednosti (jer samo l-vrijednosti imaju adrese). Ako je “`x`” objekat tipa “`Tip`”, tada je tip izraza “`&x`” tipa “pokazivač na tip `Tip`”. S druge strane, ukoliko je “`p`” pokazivač tipa “pokazivač na tip `Tip`”, tada je tip izraza “`*p`” prosti tipa “`Tip`”.

Derefencirani pokazivač može se koristiti u bilo kojem kontekstu u kojem bi se mogao koristiti bilo koji izraz čiji tip odgovara tipu na koji pokazivač pokazuje. Stoga je sljedeća naredba potpuno legalna, i ispisuje brojeve 57 i 112:

```
cout << *pokazivac + 1 << endl << 2 * *pokazivac;
```

U ovoj naredbi trebamo obratiti pažnju na dva detalja. Prvo, operator derefenciranja ima veći prioritet od aritmetičkih operatorka (sabiranja, množenja, itd.) tako da se izraz “`*pokazivac + 1`” ne interpretira kao “`(*pokazivac) + 1`” već kao “`(*pokazivac) + 1`” (kasnije ćemo vidjeti kakvo bi tačno značenje imala prva interpretacija, koja je također smislena). Drugo, u izrazu “`2 * *pokazivac`” prva zvjezdica predstavlja operator množenja, dok druga zvjezdica predstavlja operator derefenciranja (odnosno, postoje dva različita operatorka istog imena “`*`”, pri čemu je jedan binarni a drugi unarni). Kako se razmaci mogu izostaviti, istu naredbu smo mogli napisati i ovako:

```
cout << *pokazivac + 1 << endl << 2 * *pokazivac;
```

Međutim, na ovaj način je veoma nejasno šta koja zvjezdica predstavlja, pa ovakvo pisanje treba izbjegavati. Vjerovatno najjasnije značenje dobijamo ukoliko upotrijebimo zagrade:

```
cout << *pokazivac + 1 << endl << 2 * (*pokazivac);
```

Veoma važna činjenica je da derefencirani pokazivač predstavlja *l-vrijednost*, odnosno može se naći sa *lijeve strane znaka dodjeljivanja*, i na njega se mogu primijeniti operatori poput “`++`” i “`--`”. Stoga su dozvoljene naredbe poput sljedećih:

```
*pokazivac = 100;  
(*pokazivac) ++;
```

Prva naredba postavlja sadržaj lokacije na koju ukazuje promjenljiva “`pokazivac`” na vrijednost 100, dok druga naredba uvećava sadržaj lokacije na koju ukazuje promjenljiva “`pokazivac`” za jedinicu. U konkretnom primjeru, kada promjenljiva “`pokazivac`” pokazuje na promjenljivu “`broj`”, efekat ovih naredbi je isti kao da smo neposredno pisali

```
broj = 100;  
broj ++;
```

Drugim riječima, *derefencirani pokazivač ponaša se kao referenca koja upućuje na promjenljivu na koju pokazivač pokazuje*. Treba još napomenuti da su u izrazu “`(*Pokazivac) ++`” zagrade *obavezne*. Naime, operator “`++`” ima veći prioritet u odnosu na operator derefenciranja. Stoga bi se izraz “`*Pokazivac++`” interpretirao kao “`(*Pokazivac++)`”, koja je također smislena. Uskoro ćemo razjasniti i sadržaj ove druge interpretacije.

Prije nego što se upoznamo sa primjenama pokazivača, bitno je napomenuti da bez obzira što pokazivači i dalje imaju veliku primjenu u jeziku C++, oni se mnogo manje koriste nego što su se koristili u jeziku C. Naime, kako jezik C ne poznaje reference, pokazivači su u jeziku C bili jedini način da se ostvari indirektni pristup nekoj promjenljivoj. Reference koje su uvedene u jezik C++ u mnogim slučajevima pružaju jednostavniji način da se ta indirekcija ostvari. Na primjer, ukoliko u jeziku C++

želimo da napišemo funkciju koja će razmijeniti vrijednost dvije promjenljive koje joj se prenose kao parametri, možemo napisati funkciju poput sljedeće, u kojoj su formalni parametri reference:

```
void Razmijeni(int &prva, int &druga) {
    int pomocna = prva;
    prva = druga; druga = pomocna;
}
```

Ukoliko bismo sada željeli da razmijenimo vrijednost dvije cjelobrojne promjenljive “a” i “b”, to bismo mogli uraditi prostim pozivom poput

```
Razmijeni(a, b);
```

Međutim, u jeziku C ne postoje reference, tako da u jeziku C funkcije ne mogu promijeniti vrijednosti svojih stvarnih parametara (s obzirom da se bez korištenja referenci, formalni parametri ne mogu poistovijetiti sa stvarnim parametrima). Stoga je u jeziku C jedini način da se ostvari sličan efekat eksplisitno prenijeti funkciji *adrese* gdje se nalaze odgovarajući stvarni parametri. To zahtijeva da poziv funkcije izgleda poput sljedećeg:

```
Razmijeni(&a, &b);
```

Da bi ovakav poziv bio legalan, potrebno je promijeniti i samu funkciju “Razmijeni” tako da umjesto cijelih brojeva prihvata *pokazivače na cijele brojeve*. Tada je moguće upotrijebiti operator dereferenciranja da se pomoću njega *indirektno* pristupi sadržaju memorijskih lokacija gdje se nalaze stvarni parametri i da se na taj način izvrši razmjena. Na taj način funkcija “Razmijeni”, napisana u duhu jezika C, mogla bi izgledati ovako:

```
void Razmijeni(int *pok1, int *pok2) {
    int pomocna = *pok1;
    *pok1 = *pok2; *pok2 = pomocna;
}
```

Mada napisana funkcija izgleda dosta slično kao i srodnja funkcija napisana uz upotrebu prenosa parametara po referenci, početniku nije posve lako shvatiti kako ona radi. Neka su, na primjer, vrijednosti promjenljivih “a” i “b” respektivno “5” i “8”, i neka je funkcija “Razmijeni” pozvana na opisani način (tj. uz eksplisitno navođenje operatora referenciranja “&” pri pozivu funkcije). Sljedeća slika ilustrira situaciju nakon izvršavanja svake od naredbi u funkciji “Razmijeni” (navedene adrese i raspored u memoriji uzeti su kao čisto hipotetički primjer):

```
Razmijeni(&a, &b);
```

Adrese	...	3144	3145	3146	3147	3148	3149	3150	3151	...
		5	8		3144	3145				

a b pok1 pok2

```
int pomocna = *pok1;
```

Adrese	...	3144	3145	3146	3147	3148	3149	3150	3151	...
		5	8		3144	3145				5

a b pok1 pok2 pomocna

```
*pok1 = *pok2;
```

Adrese	...	3144	3145	3146	3147	3148	3149	3150	3151	...
		8	8		3144	3145			5	

a b

pok1 pok2

pomocna

```
*pok2 = pomocna;
```

Adrese	...	3144	3145	3146	3147	3148	3149	3150	3151	...
		8	5		3144	3145			5	

a b

pok1 pok2

pomocna

Vidimo da su promjenljive "a" i "b" zaista razmijenile svoje vrijednosti. Naravno, rješenje napisano u duhu jezika C++ korištenjem prenosa parametara po referenci je mnogo jednostavnije i jasnije. Bez obzira na to, ovaj primjer veoma ilustrativno demonstrira na koji način djeluju pokazivači, a ujedno i demonstrira kako se pokazivači koriste kao parametri funkcija. Treba naglasiti da se potpuno ista situacija u memoriji poput situacije prikazane na prethodnim slikama dešava i ukoliko se koristi rješenje sa referencama, samo što toga programer nije svjestan. Naime, prilikom inicijalizacije referenci automatski se uzima adresa referenciranog objekta, bez potrebe za navođenjem operatora "&" (i to se ne može izmijeniti). Također, reference se prilikom upotrebe automatski dereferenciraju, bez potrebe za navođenjem operatora "*" (i to se također ne može izmijeniti). Inače, po internoj strukturi, pokazivači i reference su potpuno identični (doduše, standard jezika C++ ne zahtijeva da mora biti tako, ali jeste tako u praktično svim postojećim implementacijama prevodilaca za C++). Interesantno je napomenuti da primjenom operatora "&" na referencu nećemo dobiti adresu same reference, već adresu objekta na koji je referenca vezana. Dakle, interna struktura reference je potpuno nedostupna, tako da je referencu jednostavno *nemoguće* razdvojiti od objekta na koji je vezana!

Dereferencirani pokazivač se, poput reference, u potpunosti ponaša kao *promjenljiva* smještena na lokaciju na koju pokazivač pokazuje, te se može upotrijebiti u bilo kojem kontekstu u kojem i bilo koja druga promjenljiva. Tako, na primjer, ukoliko imamo funkciju

```
void Obrisni(int &promjenljiva) {
    promjenljiva = 0;
}
```

koja postavlja promjenljivu koju joj je proslijedena kao argument na vrijednost nula, sasvim je moguće koristiti poziv funkcije poput

```
Obrisni(*pokazivac);
```

Ovaj poziv će postaviti na nulu lokaciju na koju pokazuje promjenljiva "pokazivac".

Jednom pokazivaču se može dodijeliti drugi pokazivač samo ukoliko *oba pokazuju na posve isti tip* (uz neke vrlo rijetke iznimke, na koje ćemo ukazati kada se za to ukaže potreba). Na primjer, ukoliko imamo deklaraciju

```
int *pok1, *pok2;
```

tada je dodjela "pok1 = pok2" posve legalna, i nakon nje "pok1" i "pok2" pokazuju na istu lokaciju u memoriji (da bi ova dodjela imala smisla, prethodno je potrebno pokazivaču "pok2" dodijeliti adresu

nekog drugog objekta). Slično vrijedi za dodjelu “`pok2 = pok1`”. S druge strane, ukoliko imamo deklaracije

```
int *pok1;
double *pok2;
```

tada niti dodjela “`pok1 = pok2`” niti dodjela “`pok2 = pok1`” *nisu dozvoljene*. Treba razlikovati ove dodjele od dodjela “`*pok1 = *pok2`” i “`*pok2 = *pok1`” koje su *potpuno legalne*, jer se dereferencirani pokazivači “`pok1`” i “`pok2`” ponašaju kao cjelobrojna odnosno realna promjenljiva, a međusobne dodjele između cjelobrojnih i realnih promjenljivih su dozvoljene (uz automatsku konverziju tipa). Također treba primijetiti da se pri deklaraciji zvjezdica treba stavljati ispred *svake promjenljive za koju želimo da bude pokazivačka promjenljiva*. Tako je u deklaraciji

```
int *a, b;
```

samo promjenljiva “`a`” pokazivačka promjenljiva (tipa “pokazivač na cijeli broj”, koji obilježavamo i kao anonimni tip “`int *`”), dok je “`b`” obična cjelobrojna promjenljiva (tipa “`int`”). Pomoću naredbe “`typedef`” moguće je deklarirati odnosno imenovati poseban *pokazivački tip*. Na primjer, deklaracijom

```
typedef int *PokNaCijeli;
```

uveli smo novi tip “`PokNaCijeli`” (kao imenovani sinonim za anonimni tip “`int *`”) koji predstavlja *tip pokazivača na cijele brojeve*. Stoga, deklaracijom

```
PokNaCijeli p1, p2;
```

deklariramo dvije promjenljive “`p1`” i “`p2`” koji su pokazivači na cijele brojeve.

Mada u svojoj osnovi možemo reći da pokazivačke promjenljive najčešće u sebi interno sadrže također *brojeve* koji predstavljaju *adrese nekih objekata* (ovo nije posve tačno na arhitekturama kod kojih organizacija memorije nije linearна), C++ pokazivačke promjenljive tretira posve drugačije nego obične brojčane promjenljive (između ostalog, i zbog činjenice da nije dobro prepostavljati ništa o tome kako je memorija zaista organizirana, s obzirom da bi smisao jezika trebala da što manje ovisi od svojstava hardvera na kojem se program izvršava). Stoga, mnoge aritmetičke operacije nad pokazivačkim promjenljivim *nisu dozvoljene*, a i one koje jesu dozvoljene *interpretiraju se bitno drugačije* nego sa brojčanim promjenljivim (na način koji *ne ovisi* od same arhitekture računara). Pokazivačke tipove treba shvatiti kao *sasvim posebne tipove*, bitno različite od cjelobrojnih tipova (bez obzira što su oni najčešće interno realizirani upravo kao cijeli brojevi). Na primjer, *nije dozvoljeno* sabrati, pomnožiti ili podijeliti dva pokazivača (razlog za ovu zabranu je činjenica da se ne vidi na šta bi smisleno mogao pokazivati pokazivač koji bi se dobio recimo množenjem adresa dva druga objekta, čak i uz pretpostavku da su te adrese obični brojevi). Slično, nije dozvoljeno pomnožiti ili podijeliti pokazivač sa brojem, ili obrnuto. Također, mada se interna vrijednost pokazivača može ispisati (u heksadekadnom obliku) na izlazni tok, *nije dozvoljeno njihovo unošenje* preko ulaznog toka (npr. tastature). Međutim, dozvoljeno je *sabrati pokazivač i cijeli broj* (i obratno), zatim *oduzeti cijeli broj od pokazivača*, kao i *oduzeti jedan pokazivač od drugog*. U nastavku ćemo razmotriti smisao ovih operacija.

Smisao pokazivačke aritmetike očitava se u slučaju kada pokazivač pokazuje na neki element niza (kako operator referenciranja “`&`” kao svoj argument prihvata bilo kakvu *l-vrijednost*, stoga njegov argument može biti i indeksirani element niza, za razliku od recimo izraza poput “`&2`” ili “`&(a + 2)`” koji su besmisleni). Pretpostavimo da imamo sljedeće deklaracije:

```
double niz[5] = {4.23, 7, 3.441, -12.9, 6.03};
double *p = &niz[2];
```

Ovim smo postavili pokazivač “`p`“ da pokazuje na element niza “`niz[2]`”. Naravno, poput svih drugih promjenljivih, inicijalizacija pokazivačke promjenljive se može (ali, za razliku od referenci, ne mora) obaviti odmah prilikom njene deklaracije. Međutim, razmotrimo sada činjenicu da elementi niza tipa “`double`“ sigurno ne mogu stati u jednu jednobajtnu memoriju lokaciju. Pretpostavimo npr. da jedan element ovakvog niza zauzima 8 bajta (u skladu sa IEEE 754 standardom). Podrazumijevajući linearni model memorije, kao i do sada, memorija slika nakon ovih deklaracija mogla bi izgledati recimo ovako (adrese su ponovo izabrane posve nasumično):

<i>Adrese</i>	...	$370 \div 377$	$378 \div 385$	$386 \div 393$	$394 \div 401$	$402 \div 409$...	550	...
		4.23	7	3.441	-12.9	6.03		386	

Niz

`p`

Pokazivač “`p`“ pokazuje na element “`niz[2]`”, a njegov stvarni sadržaj je adresa 386 (naravno, u skladu sa pretpostavljenim primjerom memorijске slike). Razmotrimo sada izraz “`p + 1`“. Ovaj izraz ponovo predstavlja pokazivač, ali koji pokazuje na element “`niz[3]`”. Stoga njegova stvarna vrijednost nije 387 ($386+1$) nego 394. Dakle, dodavanje jedinice na neki pokazivač daje kao rezultat novi pokazivač koji pokazuje na sljedeći element niza u odnosu na element na koji pokazuje izvorni pokazivač. Pri tome, stvarna adresa ne mora nužno biti povećana za 1. Zapravo, možemo zaključiti da se stvarna adresa koju sadrži pokazivač (za slučaj linearog memorijskog modela) uvećava za *veličinu objekta na koju pokazivač pokazuje* (izraženu u bajtima). Međutim, ovakva konvencija omogućila je da programer *ne mora da vodi brigu* o tome koliko zaista neki element niza zauzima prostora: ako “`p`“ pokazuje na neki element niza, “`p + 1`“ pokazuje na sljedeći element, ma gdje da se on nalazio. Što je još značajnije, isto vrijedi čak i u slučaju da memorija nije organizirana na linearan način. Izraz “`p + 1`“ uvijek pokazuje na element niza koji neposredno slijedi elementu na koji pokazuje pokazivač “`p`“, ma kako da je memorija organizirana, i ma kakav da je interni sadržaj pokazivača “`p`“. To i jeste osnovni smisao pokazivačke aritmetike. Analogno tome, izraz “`p - 1`“ predstavlja pokazivač koji pokazuje na prethodni element niza u odnosu na element na koji pokazuje pokazivač “`p`“ (u navedenom primjeru na element “`niz[1]`”). Generalno, izraz oblika “`p + n`“ odnosno “`p - n`“ gdje je “`p`“ pokazivač a “`n`“ cijeli broj (odnosno ma kakav cjelobrojni izraz) predstavlja novi pokazivač koji pokazuje na element niza koji se nalazi “`n`“ elemenata ispred odnosno iza elementa niza na koji pokazuje pokazivač “`p`“. Stoga će sljedeća naredba, za niz iz razmatranog primjera, ispisati vrijednosti “3.441”, “-12.9” i “6.03”:

```
for(int i = 0; i <= 2; i++) cout << *(p + i);
```

Uočite razliku između izraza “`* (p + i)`“ i izraza “`*p + i`“ (odnosno “`(*p) + i`“) koji, kao što smo ranije vidjeli, imaju posve drugačije značenje. Također, kako je svaki dereferencirani pokazivač *l-vrijednost*, to su izrazi poput izraza

`* (p - 1) = 8.5`

sasvim smisleni (ovaj izraz će postaviti sadržaj elementa niza “`niz[1]`“ na vrijednost “8.5”).

Bitno je napomenuti da je smisao pokazivačke aritmetike preciziran standardom jezika C++ samo za slučaj kada pokazivač pokazuje na neki element nekog niza. Za slučaj kada pokazivač pokazuje na neki drugi objekat (npr. na običnu promjenljivu), smisao pokazivačke aritmetike *nije definiran*. Na primjer, za ranije definiran pokazivač “`pokazivac`“ koji je pokazivao na cjelobrojnu promjenljivu “`broj`“, smisao izraza “`pokazivac + 1`“ i “`pokazivac - 1`“ nije preciziran standardom (tj. oni su *sintaksno ispravni*, ali nije precizirano šta im je vrijednost, stoga se ne smiju koristiti). Za slučaj linearnih memorijskih modela,

ovi izrazi bi se gotovo sigurno interpretirali kao pokazivači koji pokazuju na mesta u memoriji čije su adrese veće odnosno manje od adrese zapisane u pokazivaču "pokazivac" za onoliko bajtova koliko zauzima promjenljiva "broj". Međutim, ako memorijski model nije linearan, interpretacije zaista mogu biti svakakve. Stoga se trebamo pridržavati standarda koji strogo naglašava da pokazivačku aritmetiku treba koristiti samo ukoliko pokazivač pokazuje na neki element niza. Ovog pravila se zaista moramo pridržavati strogo: u nekim slučajevima operativni sistem ima pravo da prekine izvršavanje programa u kojem je pokazivač "odlutowao" zbog pogrešne primjene pokazivačke aritmetike, čak i ako se "zalutali" pokazivač uopće ne dereferencira (ovo se dešava iznimno rijetko, ali se dešava).

Pokazivačka aritmetika nije uvijek definirana čak ni za slučaj pokazivača koji pokazuju na elemente niza. Postoji još pravilo koje kaže da su izrazi oblika " $p + n$ " odnosno " $p - n$ " gdje je " p " pokazivač koji pokazuje na neki element niza a " n " cjelobrojni izraz definirani samo ukoliko novodobijeni pokazivač ponovo pokazuje na neki element niza koji *zaista postoji* u skladu sa specificiranom dimenzijom niza, ili eventualno na mjesto koje se nalazi neposredno *iza posljednjeg elementa niza*. Drugim riječima, vrijednost izraza " $p - n$ " postaje nedefinirana ukoliko novodobijeni pokazivač "odluta" ispred prvog elementa niza, a vrijednost izraza " $p + n$ " postaje nedefinirana ukoliko novodobijeni pokazivač "odluta" daleko iza posljednjeg elementa niza. Za konkretan primjer niza "niz" iz maloprije navedenog primjera koji sadrži 5 elemenata i pokazivača " p " koji je postavljen da pokazuje na element "niz[2]", izrazi " $p + 1$ ", " $p + 2$ ", " $p + 3$ ", " $p - 1$ " i " $p - 2$ " su precizno definirani, za razliku od izraza " $p + 4$ " ili " $p - 3$ " koji nisu, jer su "odlutili" u "nepoznate vode" (njihov sadržaj je nepredvidljiv, posebno u slučajevima kada memorijski model nije linearan).

Nad pokazivačkim promjenljivim se mogu primjenjivati i operatori "+=", "-=", "++" i "--" čije je značenje analogno kao za obične promjenljive, samo što se koristi pokazivačka aritmetika. Na primjer, izraz " $p += 2$ " će promijeniti sadržaj pokazivačke promjenljive " p " tako da pokazuje na element niza koji se nalazi dva mesta ispred elementa niza na koji " p " trenutno pokazuje, dok će izraz " $--p$ " promijeniti sadržaj pokazivačke promjenljive tako da pokazuje na prethodni element niza u odnosu na element na koji " p " trenutno pokazuje (u oba slučaja, " p " postaje nedefiniran u slučaju da odlutamo izvan prostora koji niz zauzima; eventualno, smijemo se pozicionirati tik iza kraja niza). Na taj način moguće je koristiti pokazivače čija vrijednost tokom izvršavanja programa pokazuje na razne elemente niza, kao da se pokazivač "kreće" odnosno "klizi" kroz elemente niza. Ovakve pokazivače obično nazivamo *klizeći pokazivači* (engl. *sliding pointers*), kao što je ilustrirano u sljedećem primjeru (u kojem prepostavljamo da su "niz" i " p " deklarirani kao u ranijim primjerima):

```
p = &niz[0];
for(int i = 0; i < 5; i++) {
    cout << *p;
    p++;
}
```

Dereferenciranje i inkrementiranje (odnosno dekrementiranje) pokazivača se često obavlja u istom izrazu, što je ilustrirano u sljedećem primjeru (koji je ekvivalentan prethodnom):

```
p = &niz[0];
for(int i = 0; i < 5; i++) cout << *p++;
```

U posljednjem primjeru, izraz " $*p++$ " interpretira se kao " $*(p++)$ " a ne kao " $(*p)++$ " koji ima drugačije, ranije objašnjeno značenje.

Oba navedena primjera su posve legalna i korektna. Međutim, sljedeći primjer, koji bi trebao da ispiše elemente niza u obrnutom poretku, *nije legalan* (iako je sintaksno ispravan), bez obzira što u većini

slučajeva *raditi ispravno*:

```
p = &niz[4];
for(int i = 0; i < 5; i++) {
    cout << *p;
    p--;
}
```

Isto vrijedi i za sažetu varijantu ovog primjera:

```
p = &niz[4];
for(int i = 0; i < 5; i++) cout << *p--;
```

U čemu je problem? Poteškoća leži u činjenici da nakon završetka ovih sekvenci, pokazivač “*p*” pokazuje *ispred prvog elementa niza*, što po standardu nije legalno. Operativni sistem ima pravo da prekine program u kojem se generira takav pokazivač ukoliko mu njegovo postojanje ugrožava rad (to se vrlo vjerovatno neće desiti, ali se *može desiti*). Rješenje ovog problema je posve jednostavno – nemojmo generirati takve pokazivače. Na primjer, možemo “*p*” na početku postaviti da pokazuje tačno *iza* posljednjeg elementa niza, tj. na nepostojeći element niza “*niz[5]*” (što je, začudo, legalno – na ovom mjestu nećemo ulaziti u razloge zašto) i umanjivati pokazivač *prije nego što ga dereferenciramo*, kao u sljedećem (legalnom) primjeru:

```
p = &niz[5];
for(int i = 0; i < 5; i++) {
    p--;
    cout << *p;
}
```

ili u njegovoj kompaktnijoj varijanti:

```
p = &niz[5];
for(int i = 0; i < 5; i++) cout << *--p;
```

Pokazivači se mogu *porediti* pomoću operatora “<”, “>”, “<=”, “>=”, “==” i “!=” pod uvjetom da *pokazuju na isti tip*. Dva pokazivača na različite tipove (npr. pokazivači na “**int**” i “**double**”) ne mogu se poređiti, čak i ukoliko su tipovi na koje pokazivači pokazuju uporedivi. Pokušaj poređenja pokazivača koji pokazuju na različite tipove prijavljuje se kao sintaksna greška od strane kompjerala. Zbog toga, pretpostavimo da imamo dva pokazivača koji pokazuju na objekte istog tipa. Rezultat poređenja primjenom operatora “==” i “!=” je uvijek jasno definiran: dva pokazivača su jednaka ako i samo ako pokazuju na *isti objekat*. Međutim, rezultat poređenja dva pokazivača primjenom ostalih relacionih operatora je precizno definiran standardnom samo za slučaj *kada oba pokazivača pokazuju na elemente jednog te istog niza*. Na primjer, neka su “*p1*” i “*p2*” dva pokazivača koji pokazuju na neke elemente istog niza (eventualno, oni smiju pokazivati i na fiktivni prostor iza posljednjeg elementa niza). Pokazivač “*p1*” je *manji* od pokazivača “*p2*” ukoliko “*p1*” pokazuje na element niza koji je *ispred* elementa na koji pokazuje “*p2*”, a *veći* ukoliko pokazuje na element niza koji je *iza* elementa na koji pokazuje “*p2*”. S druge strane, ukoliko pokazivači “*p1*” i “*p2*” ne pokazuju na dva elementa istog niza, pojmovi “*veći*” i “*manji*” nisu definirani, tako da rezultat poređenja *zavisi od implementacije*. Stoga se takva poređenja ne smiju koristiti. Ukoliko je memorijski model linearan, sva poređenja pokazivača se obično interno realiziraju prostim poređenjem *brojčanih vrijednosti adresu* pohranjenih u pokazivaču, tako da su rezultati poređenja praktično uvijek u skladu sa intuicijom, čak i ukoliko kršimo pravila kada je poređenje pokazivača legalno (npr. kod linearnih modela uvijek je manji onaj pokazivač koji pokazuje na nižu adresu). Međutim, kod memorijskih modela koji nisu linearni, svakakva iznenadenja su moguća. Da biste spriječili moguća iznenadenja, samo poštujte pravilo: *nemojte porediti pokazivače koji ne pokazuju na*

elemente istog niza. Također, bitno je uočiti razliku između poređenja poput “ $p1 == p2$ “ koje poredi *dva pokazivača* (koji mogu biti različiti čak i ukoliko su sadržaji lokacija na koje oni pokazuju isti) i poređenja poput “ $*p1 == *p2$ “ koje poredi *sadržaje lokacija na koje pokazuju dva pokazivača* (koji mogu biti isti čak i ukoliko su pokazivači različiti).

Mada nije moguće sabrati, pomnožiti ili podijeliti dva pokazivača, moguće je *oduzeti* jedan pokazivač od drugog, pod uvjetom da su oba pokazivača *istog tipa*. Pri tome, rezultat oduzimanja je, slično kao za slučaj poređenja, precizno definiran *samo za slučaj kada oba pokazivača pokazuju na elemente istog niza*. Po definiciji, rezultat oduzimanja takva dva pokazivača je *cijeli broj* (ne pokazivač) koji je jednak *razlici indeksa elemenata niza na koje pokazivači pokazuju*. Na primjer, ukoliko pokazivač “ $p1$ ” pokazuje na element niza “ $niz[4]$ ” a pokazivač “ $p2$ ” na element niza “ $niz[1]$ ”, tada je vrijednost izraza “ $p1 - p2$ ” cijeli broj 3 (tj. 4–1). Ovakva definicija obezbjeđuje da vrijedi pravilo da je vrijednost izraza “ $p1 + (p2 - p1)$ ” jednaka “ $p2$ ”, što je u skladu sa intuicijom.

Zahvaljujući mogućnosti poređenja pokazivača, prethodni primjeri za ispis elemenata niza uz korištenje klizećih pokazivača mogu se napisati tako da se uopće ne koristi pomoćna promjenljiva “ i ”, već samo klizeći pokazivač. Na primjer, elemente niza u standardnom poretku možemo ispisati ovako:

```
for(double *p = &niz[0]; p < &niz[5]; p++) cout << *p;
```

Pri ispisu unazad ponovo treba biti oprezniji, zbog mogućnosti da pokazivač “*odluta*” ispred prvog elementa niza. Na primjer, sljedeći primjer nije korektan:

```
for(double *p = &niz[4]; p >= &niz[0]; p--) cout << *p;
```

Naime, kada “ p ” dostigne prvi element niza, vrijednost izraza “ $p--$ ” postaje nedefinirana, pa je pogotovo nedefinirano šta je rezultat poređenja “ $p >= &niz[0]$ ”. Stoga je potpuno upitno kada će napisana petlja uopće završiti (i da li će uopće završiti). Ovo nije samo prazno filozofiranje: napisani primjer *provjereno* neće raditi sa C++ kompjuterima za MS DOS operativni sistem (kod kojeg memorijski model *nije linearan*) u slučajevima kada je niz “ niz ” deklariran kao statički ili globalni. Sigurno je moguće navesti još mnoštvo situacija u kojima ovaj primjer ne radi kako treba. Rješenje ovog problema je isto kao u ranijim primjerima: ne smijemo dozvoliti da pokazivačka aritmetika dovede do izlaska pokazivača izvan dozvoljenog opsega. Sljedeća varijanta je, na primjer, korektna (sjetimo se da se u “**for**” petlji bilo koji od 3 njena argumenta mogu izostaviti):

```
for(double *p = &niz[5]; p > &niz[0];) cout << *--p;
```

Treba naglasiti da su veoma rijetke knjige o programskom jeziku C++ u kojima se upozorava na opisanu opasnost pokazivačke aritmetike, koja posebno dolazi do izražaja kada klizeće pokazivače koristimo za kretanje kroz niz *unazad*.

U poglavljima o nizovima, informativno smo govorili o tome da se ime niza *automatski konvertira u pokazivač (odgovarajućeg tipa)* koji pokazuje na prvi element niza (preciznije, u *konstantni pokazivač*, što ćemo precizirati malo kasnije). Drugim riječima, ukoliko je “ niz ” neki niz, tada je izraz “ niz ” potpuno ekvivalentan izrazu “ $&niz[0]$ ”. Zbog automatske konverzije nizova u pokazivače, slijedi da se pokazivačka aritmetika *može primijeniti i na nizove*. Razmotrimo, na primjer, na prvi pogled besmislen izraz “ $niz + 2$ ”. Međutim, on je ekvivalentan izrazu “ $&niz[0] + 2$ ” koji je, zahvaljujući pokazivačkoj aritmetici (ne zaboravimo da je izraz “ $&niz[0]$ ” pokazivač) ekvivalentan izrazu “ $&niz[2]$ ”, što je lako provjeriti. Dakle, izrazi “ $niz + 2$ ” i “ $&niz[2]$ ” su *ekvivalentni*. Slijedi da elemente niza “ niz ” možemo ispisati i ovako:

```
for(double *p = niz; p < niz + 5; p++) cout << *p;
```

Naravno, iz istog razloga, sasvim je legalna i sljedeća konstrukcija:

```
for(int i = 0; i < 5; i++) cout << *(niz + i);
```

Iz ovoga vidimo da su, zahvaljujući pokazivačkoj aritmetici i automatskoj konverziji nizova u pokazivače, izrazi “niz[i]”, i “*(niz + i)” gdje je “niz” ma kakva nizovna promjenljiva zapravo sinonimi. Ovim je u jezicima C i C++ uspostavljena veoma tijesna veza između pokazivača i nizova. Ovi jezici su pomenutu vezu učinili još tijesnjom (na način kakav ne postoji niti u jednom drugom programskom jeziku) uvođenjem konvencije da izrazi oblika “a[n]” i “*(a + n)” uvijek budu potpuni sinonimi, bez obzira da li je “a” ime niza ili pokazivač. Drugim riječima, indeksiranje se može primijeniti i na pokazivače! Jednostavno, izraz oblika “a[n]” u slučaju kada “a” nije ime niza, po definiciji se interpretira kao “*(a + n)”, bez obzira kojeg je tipa “a”. Stoga je izraz “a[n]” ispravan kad god se zbir “a + n” može interpretirati kao pokazivač (jedna pomalo čudna posljedica ove činjenice je da su sinonimi i izrazi poput “niz[3]” i “3[niz]” mada se ovo svojstvo teško može iskoristiti za nešto pametno). Indeksiranje primjenjeno na pokazivače ilustriraćemo sljedećim primjerom koji ispisuje elemente 4.23, 7, 3.441 i -12.9 (uz pretpostavku da je niz “niz” deklariran kao u prethodnim primjerima) i koji pokazuje kako se u jeziku C++ mogu simulirati nizovi sa *negativnim indeksima* kakvi postoje u nekim drugim programskim jezicima (npr. u Pascalu i FORTRAN-u):

```
double *pok = niz + 2;  
for(int i = -2; i <= 1; i++) cout << pok[i];
```

Podsetimo se da izraz “niz + 2” ima isto značenje kao i izraz “&niz[2]”. Kasnije ćemo vidjeti i mnoge druge korisne primjene činjenice da se indeksiranje može primijeniti na pokazivače. Međutim, treba napomenuti da sljedeći primjer, koji se čak može naći u nekim knjigama o jezicima C i C++ nije legalan. Programer je htio da iskoristi indeksiranje pokazivača sa ciljem simulacije nizova čiji indeksi počinju od *jedinice* umjesto od nule:

```
double *pok = niz - 1;  
for(int i = 1; i <= 5; i++) cout << pok[i];
```

Problem sa ovim primjerom je u tome što izraz “niz - 1” nije legalan, u skladu sa već opisanim pravilima vezanim za pokazivačku aritmetiku (isto vrijedi i za njemu ekvivalentan izraz “&niz[-1]”). Stoga ovaj primjer može ali i ne mora raditi, ovisno od konkretnog sistema. Postoji velika vjerovatnoća da će on raditi ispravno (inače ga ne bi citirali u mnogim knjigama), ali postoji vjerovatnoća i da neće. To je, bez sumnje, sasvim dovoljan razlog da ne koristimo ovakva rješenja.

Bez obzira na tjesnu vezu između pokazivača i nizova, na imena nizova ne mogu se primijeniti operatori “+=”, “-=”, “++” i “--” koji mijenjaju sadržaj pokazivačke promjenljive, odnosno lokaciju na koju oni pokazuju. Imena nizova (sama za sebe, bez indeksiranja) uvijek se tretiraju kao pokazivači na *prvi element niza*, i to se *ne može primijeniti*. Oni su uvijek vezani za *prvi element niza*, slično kao što su reference čitavo vrijeme svog postojanja vezane za jedan te isti objekat. Preciznije, ime niza upotrijebljeno samo za sebe konvertira se u *konstantni (nepromjenljivi)* pokazivač na prvi element niza. Stoga se imena nizova *ne mogu koristiti kao klizeći pokazivači*.

Treba napomenuti da su jedini izuzeci u kojima se ime niza upotrijebljeno samo za sebe ne konvertira u odgovarajući pokazivač slučajevi kada se ime niza upotrijebi kao argument operatora “**sizeof**” ili “&”. Tako, npr. “**sizeof** niz” daje veličinu čitavog niza a ne veličinu pokazivača. Također, izraz “&niz” ne predstavlja adresu pokazivača, već adresu prvog elementa niza. Naravno, izrazi “niz” ili “&niz[0]” također predstavljaju adresu prvog elementa niza, ali oni nisu ekvivalentni izrazu “&niz”. Naime, tip

izraza "niz" ili "&niz[0]" je "pokazivač na element niza" (npr. pokazivač na cijeli broj). S druge strane, tip izraza "&niz" (koji se inače vrlo rijetko koristi) je "pokazivač na čitav niz". O takvim pokazivačima govorićemo kasnije. Drugim riječima, *tipovi izraza "niz"* (ili "&niz[0]") i "&niz" se *razlikuju* (iako im je vrijednost ista).

Kako se nizovi po potrebi automatski konvertiraju u pokazivače kada se upotrijebe bez indeksa, to će svaka funkcija koja kao svoj formalni parametar očekuje pokazivač na neki tip kao stvarni parametar prihvati niz elemenata tog tipa (naravno, formalni parametar će pri tome pokazivati na prvi element niza). Međutim, interesantno je da vrijedi i obrnuta situacija: funkciji koja kao formalni parametar očekuje niz možemo kao stvarni parametar proslijediti pokazivač odgovarajućeg tipa. Zapravo, jezici C i C++ uopće ne prave razliku između formalnog parametra tipa "pokazivač na tip Tip" i formalnog parametra tipa "Niz elemenata tipa Tip" (u oba slučaja funkcija dobija samo adresu prvog elementa niza, što zapravo objašnjava zbog čega izgleda da se nizovni parametri prenose u funkcije po referenci). Drugim riječima, formalni parametri nizovnog tipa, ne samo da se mogu koristiti kao pokazivači, već oni zaista i jesu pokazivači, bez obzira što u općem slučaju, nizovi i pokazivači nisu u potpunosti ekvivalentni, kao što smo već naglasili. To je ujedno razlog zbog čega operator "**sizeof**" ne daje ispravnu veličinu niza ukoliko se primjeni na formalni argument nizovnog tipa, o čemu smo već ranije govorili (naime, kao rezultat se dobija veličina *pokazivača*).

Činjenica da su formalni argumenti nizovnog tipa zapravo pokazivači, omogućava nam da funkcije koje obrađuju nizove možemo realizirati i preko klizećih pokazivača. Na primjer, funkciju koja će ispisati na ekranu elemente niza realnih brojeva možemo umjesto uobičajenog načina realizirati i na sljedeći način:

```
void IspisiNiz(double *pokazivac, int n) {
    for(int i = 0; i < n; i++) cout << *pokazivac++ << endl;
}
```

Ovu funkciju možemo pozvati na posve uobičajeni način:

```
IspisiNiz(niz, 5);
```

Nikakve razlike ne bi bilo ni da smo formalni parametar deklarirali kao "**double pokazivac []**". U oba slučaja on bi se mogao koristiti kao klizeći pokazivač. Dakle, za formalne parametre nizovnog tipa ne vrijedi pravilo da su uvijek vezani isključivo za prvi element niza. Oni se mogu koristiti u potpunosti kao i pravi pokazivači (iz prostog razloga što *oni to i jesu*). Zapravo, mogućnost da se formalni parametri deklariraju kao da su nizovnog tipa podržana je samo sa ciljem da se jasnije izrazi namjera da funkcija kao stvarni parametar očekuje *niz*. Deklaracija formalnog parametra kao pokazivača početniku bi svakako djelovala mnogo nejasnije.

Moguće je deklarirati i *pokazivače na konstantne objekte*. Na primjer, deklaracijom

```
const double *p;
```

deklariramo pokazivač "p" na *konstantnu realnu vrijednost* (tip ovog pokazivača je "**const double ***", za razliku od pokazivača na nekonstantnu realnu vrijednost, čiji je tip prosto "**double ***"). Ovakvi pokazivači omogućavaju da se sadržaj lokacije na koju oni pokazuju *čita*, ali ne i da se *mijenja*. Na primjer, sa ovakvim pokazivačem izrazi "cout << *p" i "broj = p[2]" su dozvoljeni (pod uvjetom da je promjenljiva "broj" deklarirana kao realna promjenljiva), ali izrazi "*p = 2.5" i "p[2] = 0" nisu. Odavde odmah vidimo da se formalni parametar "pokazivac" u prethodnoj funkciji mogao deklarirati kao "**const double *pokazivac**", što bi čak bilo veoma preporučljivo. Naime, na taj način bismo spriječili eventualne nehotične promjene objekta na koji pokazivač "pokazivac" pokazuje.

Na ovom mjestu potrebno je razjasniti neke česte zablude vezane za pokazivače na konstantne objekte. Prvo, pokazivač na konstantni objekat neće automatski učiniti objekat na koji on pokazuje konstantom. Na primjer, ako je “`p`” pokazivač na konstantnu realnu vrijednost, a “`broj`” neka realna (nekonstantna) promjenljiva, dodjela “`p = &broj`” neće učiniti *promjenljivu* “`broj`” *konstantom*. Promjenljiva “`broj`” će se i dalje moći mijenjati, ali ne preko pokazivača “`p`”! Ipak, adresu *konstantnog objekta* (npr. adresu neke konstante) moguće je dodijeliti samo pokazivaču na konstantni objekat. Također, pokazivaču na nekonstantni objekat nije moguće (bez eksplisitne primjene operatora za konverziju tipa) dodijeliti pokazivač na konstantni objekat (čak i ukoliko su tipovi na koje oba pokazivača pokazuju isti), jer bi nakon takve dodjele preko pokazivača na nekonstantni objekat mogli promijeniti sadržaj konstantnog objekta, što je svakako nekonzistentno. Obrnuta dodjela je legalna, tj. pokazivaču na konstantni objekat moguće je dodijeliti pokazivač na nekonstantni objekat istog tipa, s obzirom da takva dodjela ne može imati nikakve štetne posljedice.

Treba primijetiti da se sadržaj pokazivača na konstantne objekte može mijenjati, tj. sam pokazivač *nije konstantan*. To treba razlikovati od *konstantnih pokazivača*, čija se vrijednost ne može mijenjati, ali se sadržaj objekata na koji oni pokazuju može mijenjati (npr. imena nizova upotrijebljena sama za sebe konvertiraju se upravo u konstantne pokazivače). Konstantni pokazivači mogu se deklarirati na sljedeći način (primijetimo da se kvalifikator “`const`” ovdje piše iza zvjezdice):

```
double *const p = &niz[2];
```

Primijetimo da se konstantni pokazivač *mora odmah inicijalizirati*, jer mu se naknadno vrijednost više ne može mijenjati. Ovim su konstantni pokazivači veoma slični referencama, jer su i oni čitavo vrijeme vezani za jedan te isti objekat (jedina razlika između referenci i konstantnih pokazivača je u tome što se reference automatski dereferenciraju, dok konstantne pokazivače i dalje treba eksplisitno dereferencirati pomoću operatora “`*`”). Naravno, operacije poput “`p++`” ili “`p += 2`” koje bi *promijenile* sadržaj pokazivača “`p`” nisu dozvoljene (s obzirom da je “`p`” konstantan pokazivač), ali je izraz “`p + 2`” posve legalan (i predstavlja konstantni pokazivač koji pokazuje na element niza “`niz[4]`”). Dakle, bez obzira na sličnost, konstantni pokazivači su i dalje fleksibilniji od referenci, jer se na njih može u određenoj mjeri primjenjivati i pokazivačka aritmetika. Konačno, moguće je deklarirati i *konstantni pokazivač na konstantni objekat*, npr. deklaracijom poput sljedeće:

```
const double *const p = &Niz[2];
```

Može se postaviti pitanje zašto se uopće patiti sa pokazivačima i koristiti pokazivačku aritmetiku ukoliko se ista stvar može obaviti i uz pomoć indeksiranja. Ako zanemarimo dinamičku alokaciju memorije i dinamičke strukture podataka, o kojima će kasnije biti govora i koji se ne mogu izvesti bez upotrebe pokazivača, osnovni razlog je *efikasnost* i *fleksibilnost*. Naime, mnoge radnje se mogu neposrednije obaviti upotrebom klizećih pokazivača nego pomoću indeksiranja, naročito u slučajevima kada se elementi niza obrađuju sekvencijalno, jedan za drugim. Pored toga, pokazivač koji pokazuje na neki element niza sadrži u sebi cijelokupnu informaciju koja je potrebna da se tom elementu pristupi. Kod upotrebe indeksiranja ta informacija je razbijena u dvije neovisne cjeline: ime niza (odnosno adresa prvog elementa niza) i indeks posmatranog elementa.

Navedimo jedan klasični školski primjer koji se navodi u gotovo svim udžbenicima za programski jezik C (nešto rijede za C++, jer se C++ ne hvali toliko pokazivačima koliko C), koji ilustrira efikasnost upotrebe klizećih pokazivača. Naime, razmotrimo kako bi se funkcija “`KopirajString`” (odnosno njoj ekvivalentna funkcija “`strcpy`” iz biblioteke “`cstring`”), koju smo razmatrali u poglavljju o stringovima, mogla realizirati uz pomoć pokazivačke aritmetike. Izvorna verzija funkcije “`KopirajString`” glasila je ovako:

```

void KopirajString(char odredisni[], const char izvorni[]) {
    int i(0);
    while(izvorni[i] != 0) {
        odredisni[i] = izvorni[i];
        i++;
    }
    odredisni[i] = 0;
}

```

Zamijenimo sada parametre “odredisni” i “izvorni” klizećim pokazivačima (strogog uzevši, oni to već i sada jesu, ali ćemo ipak izmijeniti i njihovu deklaraciju, da jasnije istaknemo namjeru da ih želimo koristiti kao klizeće pokazivače). Odmah vidimo da nam indeks “i” više nije potreban:

```

void KopirajString(char *odredisni, const char *izvorni) {
    while(*izvorni != 0) {
        *odredisni = *izvorni;
        odredisni++; izvorni++;
    }
    *odredisni = 0;
}

```

Dalje, oba inkrementiranja možemo obaviti u istom izrazu u kojem vršimo dodjelu. Također, test različitosti od 0 možemo izostaviti (zbog načina kako se uvjeti tretiraju u jeziku C++, odnosno automatske konverzije u tip “**bool**”) čime dobijamo sljedeću verziju funkcije:

```

void KopirajString(char *odredisni, const char *izvorni) {
    while(*izvorni) *odredisni++ = *izvorni++;
    *odredisni = 0;
}

```

Konačno, kako je u jeziku C++ dodjela također izraz, rezultat izvršene dodjele je zapravo rezultat izraza “*izvorni” (inkrementiranje se obavlja naknadno) koji je identičan sa izrazom u uvjetu “**while**” petlje. Stoga se funkcija može svesti na sljedeći oblik:

```

void KopirajString(char *odredisni, const char *izvorni) {
    while(*odredisni++ = *izvorni++);
}

```

Djeluje nevjerojatno, zar ne!? Primijetimo da je nestala potreba čak i za dodjelom “*Odredisni = 0”, jer je lako uočiti da će ovako napisana petlja iskopirati i “NUL” graničnik. Ovaj primjer veoma ilustrativno pokazuje u kojoj mjeri se klizeći pokazivači mogu iskoristiti za optimizaciju kôda (doduše, po cijenu čitljivosti i razumljivosti, mada se na pokazivačku aritmetiku programeri brzo naviknu kada je jednom počnu koristiti). Funkcija “**strcpy**” iz biblioteke “**cstring**” implementirana je upravo ovako. Odnosno, ne baš u potpunosti: prava funkcija “**strcpy**” iz biblioteke “**cstring**” implementirana je tačno ovako (ako zanemarimo da su imena promjenljivih u stvarnoj implementaciji vjerovatno na engleskom jeziku, što je naravno posve nebitno):

```

char *strcpy(char *odredisni, const char *izvorni) {
    char *pocetak_odredista = odredisni;
    while(*odredisni++ = *izvorni++);
    return pocetak_odredista;
}

```

Praktično jedina razlika je u tome što (prava) funkcija “**strcpy**”, pored toga što obavlja kopiranje,

vraća kao rezultat pokazivač na prvi element niza u koji se vrši kopiranje (isto svojstvo posjeduje i funkcija “`strcat`”). Ovaj primjer ujedno demonstrira da funkcija može kao rezultat vratiti pokazivač. Povratna vrijednost koju vraćaju funkcije “`strcpy`” i “`strcat`” najčešće se ignorira, kao što smo i mi radili u svim dosadašnjim primjerima. Međutim, ova povratna vrijednost može se nekada korisno upotrijebiti za ulančavanje funkcija “`strcpy`” i/ili “`strcat`”. Na primjer, posmatrajmo sljedeću sekvencu naredbi (uz pretpostavku da su “`recenica`”, “`rijec1`”, “`rijec2`” i “`rijec3`” propisno deklarirani nizovi znakova):

```
strcpy(recenica, rijec1);
strcat(recenica, rijec2);
strcat(recenica, rijec3);
cout << recenica;
```

Zahvaljujući činjenici da funkcije “`strcpy`” i “`strcat`” vraćaju pokazivač na odredišni niz, ove četiri naredbe možemo kompaktnije zapisati u vidu sljedeće naredbe:

```
cout << strcat(strcat(strcpy(recenica, rijec1), rijec2), rijec3);
```

Treba napomenuti da pomenuto svojstvo funkcija “`strcat`” i “`strcpy`” može početnika da dovede u zabludu. Naime, mnogi početnici često isprobaju naredbu poput sljedeće:

```
cout << strcat("Dobar ", "dan!");
```

Postoji velika vjerovatnoća da će ova naredba zaista ispisati pozdrav “Dobar dan!” na ekran (uskoro ćemo vidjeti zašto), iz čega početnik može zaključiti da je ona ispravna. Međutim, ovakva naredba je strogo zabranjena, samim tim što konstrukcija “`strcat("Dobar ", "dan!")`” nije dozvoljena zbog razloga o kojima smo govorili u poglavlju o stringovima (kopiranje će se obaviti u nedozvoljenu zonu u memoriji). Posljedice ovakve “zabranjene” naredbe mogu se odraziti znatno kasnije u programu, što može biti veoma frustrirajuće. Ako nam operativni sistem odmah prekine izvođenje ovakvog programa zbog zabranjenih akcija, još možemo smatrati da smo imali sreće.

Pokazivačka aritmetika može se veoma lijepo iskoristiti zajedno sa funkcijama koje barataju sa klasičnim nul-terminiranim stringovima (“`strcpy`”, “`strcat`”, “`strcmp`” itd.). Na primjer, ukoliko je potrebno iz niza znakova “`recenica`” izdvajiti sve znakove od desetog nadalje u drugi niz znakova “`recenica2`”, to najlakše možemo uraditi na jedan od sljedeća dva načina (oba su ravnopravna):

```
strcpy(recenica2, &recenica[9]);
strcpy(recenica2, recenica + 9);
```

Ista tehnika može se iskoristiti i sa funkcijama koje prihvataju proizvoljne vrste nizova kao parametre, odnosno koje prihvataju pokazivače na proizvoljne tipove. Na primjer, pretpostavimo da želimo iz ranije deklariranog niza realnih brojeva “`niz`” ispisati samo drugi, treći i četvrti element (tj. elemente sa indeksima 1, 2 i 3). Nema potrebe da prepravljamo već napisanu funkciju “`IspisiNiz`”, s obzirom da ovaj cilj možemo veoma jednostavno ostvariti jednim od sljedeća dva poziva, bez obzira da li je funkcija napisana da prihvata pokazivače ili nizove kao svoj prvi argument, i bez obzira da li je napisana korištenjem indeksiranja ili klizećih pokazivača:

```
IspisiNiz(&niz[1], 3);
IspisiNiz(niz + 1, 3);
```

Ova univerzalnost ostvarena je automatskom konverzijom nizova u pokazivače, kao i mogućnošću da se indeksiranje primjenjuje na pokazivače kao da se radi o nizovima.

Tjesna veza između nizova i pokazivača i činjenica da se nizovi znakova tretiraju donekle različito od ostalih nizova ima kao posljedicu da se i pokazivači na znakove tretiraju neznatno drugačije nego pokazivači na druge objekte. Tako, ukoliko je “`p`” pokazivač na znakove (tj. na tip “`char`”), izraz “`cout << p`” neće ispisati *adresu* smještenu u pokazivač “`p`” (kao što bi vrijedilo za sve druge tipove pokazivača), nego redom sve znakove iz memorije počev od adrese smještene u pokazivač “`p`”, pa sve do “NUL” graničnika. Na ovaj način je, kao prvo, ostvarena konzistencija sa tretmanom znakovnih nizova od strane objekta izlaznog toka “`cout`”, a kao drugo, omogućeni su neki interesantni efekti. Na primjer, sljedeća skupina naredbi

```
char recenica[] = "Principi programiranja";
char *p = &recenica[9];
cout << p;
```

ispisće tekst “programiranja” na ekran. Naime, nakon izvršene dodjele, “`p`” pokazuje na deseti znak rečenice smještene u niz “`recenica`” (indeksiranje počinje od nule), a to je upravo prvi znak druge riječi u nizu “`recenica`”, odnosno početak dijela teksta koji glasi “`programiranja`”. Naravno, umjesto izraza “`&recenica[9]`” mogli smo pisati i “`recenica + 9`” (s obzirom da bi se u ovom primjeru upotreba imena “`recenica`” bez indeksa automatski konvertirala u pokazivač na prvi element niza, nakon čega bi pokazivačka aritmetika odradila ostatak posla). Također, uopće nismo ni morali uvoditi pokazivačku promjenljivu “`p`”: mogli smo prosto pisati

```
char recenica[] = "Principi programiranja";
cout << &recenica[9];
```

odnosno

```
char recenica[] = "Principi programiranja";
cout << recenica + 9;
```

Isti efekat proizvela bi čak i sljedeća naredba (razmislite zašto):

```
cout << "Principi programiranja" + 9;
```

Pokazivači na znakove su jedini pokazivači koji se mogu koristiti sa ulaznim tokom “`cin`”. Tako, ako je “`p`” pokazivač na znakove, izvršavanjem izraza “`cin >> p`” znakovi pristigli sa ulaznog toka (do prvog razmaka) smjestiće se u memoriju počev od adrese smještene u pokazivaču “`p`”. Slično vrijedi i za konstrukciju poput “`cin.getline(p, 100)`”. Zajedno sa pokazivačkom aritmetikom, ovo se može iskoristiti za razne interesantne efekte. Na primjer, pomoću sekvene naredbi

```
char recenica[100] = "Početak: ", *p = &recenica[9];
cin >> p;
```

sa tastature će se unijeti *jedna riječ*, i smjestiti u niz `recenica` počev od desetog znaka, odnosno tačno iza teksta “Početak:”. I ovdje smo mogli izbjegći uvođenje promjenljive “`p`” i pisati direktno izraz poput “`cin >> &recenica[9]`” ili “`cin >> recenica + 9`”. Ukoliko bismo umjesto jedne riječi željeli unijeti *čitavu liniju teksta*, koristili bismo konstrukciju “`cin.getline(p, 91)`” odnosno, bez uvođenja pomoćne promjenljive “`p`”, konstrukcije poput “`cin.getline(&recenica[9], 91)`” odnosno “`cin.getline(recenica + 9, 91)`”. Maksimalna dužina 91 određena je tako što je od maksimalnog kapaciteta od 100 znakova odbijen prostor koji je već rezerviran za tekst “Početak:”. Naravno, umjesto pisanja “magičnog broja” 91, mogli smo pisati izraz “`sizeof recenica - 9`”.

Za razliku od nizova, kao što smo to već jednom prilikom istakli, vektori (odnosno promjenljive tipa “vector”) ne konvertiraju se automatski u pokazivače, čak i ukoliko se upotrijebe bez indeksa. To znači da na vektore pokazivačka aritmetika nije neposredno primjenljiva. Međutim, sasvim je moguće pomoći operatoru “&” uzeti adresu nekog elementa vektora i na tako dobijeni pokazivač primjenjivati pokazivačku aritmetiku. S obzirom da standard jezika C++ garantira da su elementi vektora također u memoriji smješteni kontinuirano, jedan za drugim, pokazivačka aritmetika nad tako dobijenim pokazivačem legalna je pod istim uvjetima koji vrijede za pokazivače na elemente nizova (odnosno, legalna je sve dok novoformirani pokazivač također pokazuje na neki element istog vektora, ili eventualno na prostor iza posljednjeg elementa vektora). Na primjer, neka imamo deklaraciju

```
vector<double> v(10);
```

i neka je vektor “v” napunjen sa 10 realnih brojeva. Tada je konstrukcija

```
for(double *p = &v[0]; p < &v[10]; p++) cout << *p;
```

koja vrši ispis elemenata vektora posve legalna, dok sljedeća konstrukcija nije (zbog nepostojanja automatske konverzije vektora u pokazivače):

```
for(double *p = v; p < v + 10; p++) cout << *p;
```

Treba napomenuti da postoje i kontejnerski tipovi podataka koji su *veoma slični* tipu “vector” ali za koje ne postoje garancije da su im elementi smješteni kontinuirano u memoriji (npr. takav tip podataka je tip “deque”). Za pokazivače na elemente takvih tipova podataka *nije legalno* koristiti pokazivačku aritmetiku, već se moraju koristiti općenitiji objekti od pokazivača, koji se nazivaju *iteratori*. O takvima strukturama podataka i iteratorima govorićemo kasnije. Za sada treba zapamtiti da je pokazivačka aritmetika valjana samo za pokazivače koji pokazuju na elemente *nizova ili vektora*.

Pokazivači i pokazivačka aritmetika omogućavaju prilično moćne trikove, koje bi inače bilo znatno teže realizirati. S druge strane, pokazivači mogu biti i *veoma opasni*, jer lako mogu “odlутati” u neko nepredviđeno područje memorije, što omogućava da pomoći njih indirektno promijenimo praktički bilo koji dio memorije u kojem su dopuštene izmjene (posljedice ovakvih intervencija su pretežno nepredvidljive i često fatalne). Na primjer, već smo rekli da je smisao pokazivačke aritmetike precizno definiran samo za pokazivače koji pokazuju na neki element niza, i to samo pod uvjetom da novodobijeni pokazivač ponovo pokazuje na neki element niza, ili eventualno tačno iza posljednjeg elementa niza. Ukoliko se ne pridržavamo ovih pravila, pokazivači mogu “odlутati” u nedozvoljena područja memorije. Razmotrimo, na primjer, sljedeću sekvencu naredbi:

```
int a = 5, *p = &a;
p++;
*p = 3;
```

Ovdje smo prvo deklarirali cjelobrojnu promjenljivu “a” koju smo inicijalizirali na vrijednost “5”, a zatim pokazivačku promjenljivu “p” koju smo inicijalizirali tako da pokazuje na promjenljivu “a” (odnosno, ona sada sadrži adresu promjenljive “a”). Nakon toga je na promjenljivu “p” primijenjen operator “++”. Na šta sada pokazuje promjenljiva “p”? Strogo uzevši, odgovor nije definiran u skladu sa pokazivačkom aritmetikom, ali u skladu sa načinom kako se pokazivačka aritmetika implementira u većini kompjlera, “p” će gotovo sigurno pokazivati na adresu neposredno *iza* prostora koji zauzima promjenljiva “a”. Nakon toga će dodjela “*p = 3” *na to mjesto u memoriji* upisati vrijednost “3”. Međutim, šta se nalazi “na tom mjestu u memoriji”? Odgovor je nepredvidljiv: možda neka druga promjenljiva (ova varijanta je najvjerojatnija), možda ništa pametno, a možda neki podatak od vitalne

važnosti za rad operativnog sistema! Samim tim, posljedica ove dodjele je nepredvidljiva. U svakom slučaju, dodjelom “ $*p = 3$ “ mi smo zapravo “napali” memoriju koju svrhu ne znamo, što je strogo zabranjeno. Za pokazivač koji je “odlutaš” tako da više ne znamo na šta pokazuje kažemo da je *divlji pokazivač* (engl. *wild pointer*). To ne mora značiti da ne znamo koja je adresa u njemu, nego samo da ne znamo kome pripada adresa na koju on pokazuje. Divlji pokazivači nikada se ne bi trebali pojaviti u programu. Nažalost, greške koje nastaju uslijed divljih pokazivača veoma se teško otkrivaju, a prilično ih je lako napraviti. Zbog toga se u literaturi često citira da su “pokazivači zločestii” (engl. *pointers are evil*), i oni predstavljaju drugi “zločestii” koncept koji smo upoznali u jeziku C++ (prije su bili nizovi). Smatra se da oko 90% svih fatalnih grešaka u današnjim profesionalnim programima nastaje upravo zbog ilegalnog pristupa memoriji preko divljih pokazivača!

Treba primijetiti da je i pokazivač koji pokazuje tačno iza posljednjeg elementa niza također divlji pokazivač, jer i on pokazuje na lokaciju koja ne pripada nizu (niti se zna kome ona zapravo pripada). Međutim, standard jezika C++ kaže da je takav pokazivač *legalan*. Nije li ovo kontradikcija? Stvar je u tome što je dozvoljeno *generirati* odnosno *kreirati* pokazivač koji pokazuje iza posljednjeg elementa niza. Pored toga, garantira se da je on *veći* od svih pokazivača koji pokazuju na elemente tog niza. Ipak, to još uvijek ne znači da takav pokazivač smijemo *derefencirati*. Pokušaj njegovog dereferenciranja također može imati nepredvidljive posljedice. Međutim, njega bar smijemo *kreirati* bez opasnosti. Druge vrste divljih pokazivača ponekad ne smijemo ni *kreirati* – samo njihovo *kreiranje* može dovesti do kraha čak i ukoliko se oni uopće ne *derefenciraju* (to se, u nekim rijetkim slučajevima, može desiti npr. ukoliko primjenom pokazivačke aritmetike pokazivač odluta ispred prvog elementa niza). Dalje, sve operacije sa njima (npr. poređenje, itd.) potpuno su nedefinirane. U tom smislu, pokazivač koji pokazuje neposredno iza elementa niza legalan je u smislu da ga smijemo *kreirati* (tj. njegovo *kreiranje* garantirano neće izazvati krah) i smijemo ga porebiti sa drugim pokazivačima, jedino ga ne smijemo *derefencirati*. Sa drugim divljim pokazivačima *ne smijemo raditi apsolutno ništa*. Čak i sam pokušaj njihovog *kreiranja* (koji tipično nastaje neispravnom upotrebo pokazivačke aritmetike) može biti fatalan po program!

U vezi sa divljim pokazivačima neophodno je naglasiti jednu nezgodnu osobinu: svi pokazivači su neposredno nakon deklaracije divlji, sve dok im se eksplicitno ne dodijeli vrijednost! Naime, kao i sve ostale promjenljive, njihova vrijednost je *nedefinirana* neposredno nakon deklaracije, sve do prve dodjele (osim u slučajevima kada se uporedo sa dodjelom vrši i inicijalizacija). Tako je i sa pokazivačima: njihova početna vrijednost je nedefinirana, tako da se sve do prve dodjele ne zna ni na šta pokazuju! Na primjer, sljedeća sekvenca naredbi ima nepredvidljive posljedice jer “napada” nepoznati dio memorije, s obzirom da se ne zna na šta pokazivač “ p “ pokazuje:

```
int *p;  
*p = 10;
```

Sličnu situaciju imamo u sljedećem izrazito neispravnom primjeru, koji se veoma često može sresti kod početnika:

```
char *p;  
cin >> p;  
cout << p;
```

Mada onome ko isproba navedeni primjer može izgledati da radi korektno (odnosno da ispisuje na ekran riječ prethodno unesenu sa tastature), on sadrži katastrofalan grešku. Naime, kako je “ p “ *neinicijaliziran* pokazivač, njegov sadržaj je nepredvidljiv, pa je i potpuno nepredvidljivo gdje će se uopće u memoriji smjestiti znakovi pročitani sa tastature! Može se desiti da se program koji sadrži ovakve naredbe “sruši” tek nakon više sati, ukoliko se ispostavi da su smješteni znakovi “upropastili” neke bitne podatke u memoriji. Može se desiti da u startu “ p “ pokazuje na neki dio memorije zabranjen za upis od strane operativnog sistema. U tom slučaju program će se srušiti odmah (zapravo, operativni sistem će

prekinuti njegovo izvršavanje). U tom slučaju čak možemo reći da smo imali sreću!

Treba napomenuti da pokazivač koji pokazuje na neki element vektora (osnosno objekta tipa “vector”) može postati divlji ukoliko se nad vektorom izvrši neka operacija koja mijenja broj elemenata vektora, kao što je “push_back” ili “resize”. Naime, promjena veličine vektora može dovesti do pomjeranja pozicija elemenata vektora u memoriji, tako da nakon pomjeranja adrese svih elemenata ne moraju nužno ostati iste, odnosno pokazivači koji su pokazivali na elemente vektora mogu, nakon obavljenog pomjeranja, pokazivati na nešto sasvim drugo. Stoga je, nakon izvršavanja takvih operacija, uvijek potrebno izvršiti ponovnu dodjelu svim pokazivačima koji su eventualno pokazivali na neke od elemenata vektora. Greške ovog tipa rijetko se opisuju u literaturi. One doduše nisu osobito česte, ali se veoma teško otkrivaju ukoliko se pojave. Stoga je potrebno ukazati na njihovu mogućnost.

Već smo rekli da pokazivačima nije dozvoljeno eksplisitno dodjeljivati brojeve. Međutim, postoji jedan broj koji se smije dodjeliti svakom tipu pokazivača: broj *nula* (0). Pokazivač kojem je dodijeljena vrijednost 0 naziva se *nul-pokazivač* i često se obilježava sa “NULL” (ne “NUL”), tako da mnoge biblioteke funkcija definiraju simboličku konstantu “NULL” koja ima vrijednost 0. Po konvenciji, smatra se da nul-pokazivač ne pokazuje *ni na šta*. Obično se pokazivač inicijalizira na nulu kada eksplisitno želimo da naglasimo da pokazivač još uvijek ne pokazuje ni na šta konkretno. Na primjer:

```
int *p(0);
```

Naravno, istu stvar smo mogli uraditi i ovako:

```
int *p = 0;
```

Na taj način lako možemo kasnije ispitati u programu da li pokazivač pokazuje na nešto smisleno tako što ćemo prosto testirati da li je vrijednost pokazivača nula. Pored očiglednih testova tipa “`p == 0`” odnosno “`p != 0`”, sintaksa jezika C++ dozvoljava i samo navođenje izraza “`!p`” odnosno “`p`” unutar uvjeta “`if`” naredbe (svi pokazivači upotrijebljeni unutar uvjeta automatski se konvertiraju u logičku vrijednost “`true`”, osim nul-pokazivača koji se konvertira u logičku vrijednost “`false`”). Naravno, vrijednost “0” možemo pokazivaču *dodjeliti* bilo kada, a ne samo prilikom inicijalizacije. To obično radimo kada želimo da kažemo da pokazivač *više ne pokazuje* na neki konkretni objekat.

Konvencija da se pokazivač koji ne pokazuje ni na šta eksplisitno inicijalizira na nulu može učiniti divlje pokazivače “manje divljim”. Oni su i dalje divlji u smislu da ne pokazuju ni na šta smisleno, tako da ih ne smijemo dereferencirati. Međutim, njihov sadržaj nije nepredvidljiv, nego je *tačno određen*, tako da programski možemo testirati (poređenjem sa nulom) da li on pokazuje na nešto smisleno ili ne. Interesantno je da standard jezika C++ ne insistira da nul-pokazivači budu implementirani tako da im sadržaj zaista bude nula (tj. da čuvaju u sebi adresu nula). Postoje jaki razlozi (tehničke prirode) zbog kojeg se ovo ne insistira. Međutim, kako god da su oni interno implementirani, dodjela nule pokazivaču i njihovo poređenje sa nulom mora da radi tačno onako kako treba (tako da korisnik ne treba da brine o implementaciji)!

Nul-pokazivači se također često vraćaju kao rezultati iz funkcija koje kao rezultat vraćaju pokazivače, u slučaju da nije moguće vratiti neku drugu smislenu vrijednost. Na primjer, biblioteka “`cstring`” sadrži veoma korisnu funkciju “`strstr`” koja prihvata dva nul-terminalirana stringa kao parametre. Ova funkcija ispituje da li se drugi string sadrži u prvom stringu (npr. string “je lijep” sadrži se u stringu “dan je lijep dan”). Ukoliko se nalazi, funkcija vraća kao rezultat *pokazivač na znak* unutar prvog stringa od kojeg počinje tekst identičan drugom stringu (u navedenom primjeru, to bi bio pokazivač na prvu pojavu slova “j” u stringu “dan je lijep dan”). U suprotnom, funkcija kao rezultat vraća *nul-pokazivač*. Sljedeći primjer demonstrira kako se ova funkcija može iskoristiti (obratite pažnju na igre sa pokazivačkom

aritmetikom):

```
char recenica[100], fraza[100], *pozicija;
cin.getline(recenica, 99);
cin.getline(fraza, 99);
pozicija = strstr(recenica, fraza);
if(pozicija != 0)
    cout << "Navedena fraza se nalazi u rečenici počev od "
        << pozicija - recenica + 1 << ". znaka\n";
else
    cout << "Navedena fraza se ne nalazi u rečenici\n";
```

Pored funkcije “`strstr`”, biblioteka “`cstring`” sadrži i sličnu funkciju “`strchr`”, samo što je njen drugi parametar `znak` a ne string (prihvata se i brojčana vrijednost, sa značenjem ASCII šifre). Ona vraća pokazivač na prvu pojavu navedenog znaka u stringu, ili nul-pokazivač ukoliko takvog znaka nema. Ova funkcija se može zgodno iskoristiti da se dobije pokazivač na “`NUL`” graničnik stringa (za tu svrhu, funkciji “`strchr`” treba proslijediti nulu kao drugi parametar). Biblioteka “`cstring`” sadrži još mnoštvo korisnih funkcija koje kao rezultat vraćaju pokazivače, koje ovdje nećemo opisivati s obzirom da bi njihov opis zauzeo previše prostora. Napomenimo da su sve ove funkcije namijenjene isključivo za rad sa klasičnim, nul-terminaliranim stringovima. Funkcije slične namjene postoje i za dinamičke stringove, samo što se koriste na nešto drugačiji način i one nikada ne vraćaju pokazivače kao rezultate (pokazivače i dinamičke stringove nije dobro koristiti u istom kontekstu, jer su oni konceptualno različiti). Zainteresirani čitatelji i čitateljke upućuju se na mnogobrojnu literaturu koja detaljno obraduje standardnu biblioteku “`string`” i operacije sa dinamičkim stringovima. Vrijedi još napomenuti da je, u slučaju potrebe, legalno uzeti adresu nekog elementa dinamičkog stringa i sa tako dobijenim pokazivačem (koji je tipa `pokazivač na znak`, isto kao i u slučaju klasičnih nul-terminaliranih stringova) koristiti pokazivačku aritmetiku (tj. garantira se da se elementi dinamičkih stringova također čuvaju u memoriji jedan za drugim). Tako dobijene adrese su garantirano ispravne sve dok se dužina stringa ne promjeni, nakon čega više ne moraju biti (s obzirom da promjena veličine stringa, kao i u slučaju vektora, može dovesti do pomjeranja njegovih elemenata u memoriji).

Pošto u ovom poglavlju imamo namjeru detaljno govoriti o pokazivačima, recimo još nekoliko riječi o pokazivačima na znakove. Interesantno je da se pokazivači na konstantne znakove mogu inicijalizirati tako da pokazuju na neku stringovnu konstantu, kao da se radi o konstantnim znakovnim nizovima, koji se mogu inicijalizirati na sličan način. Na primjer:

```
const char *p = "Ovo je neki string...";
cout << p;
```

Ovakvim pokazivačima je čak moguće i naknadno “dodijeliti” stringovne konstante, pri čemu ovakva “dodjela” naravno ne dodjeljuje samu stringovnu konstantu pokazivačkoj promjenljivoj (ona nema “prostora” da primi cijeli niz znakova u sebe), već *adresu stringovne konstante*. Međutim, bez obzira na to, ovakva “poludodjela” može, zajedno sa specijalnim tretmanom pokazivača na znakove, proizvesti interesantne efekte. Na primjer, sljedeći primjer je posve legalan:

```
const char *recenica;
recenica = "Ja sam prva rečenica...";
cout << recenica << endl;
recenica = "A ja sam druga rečenica...";
cout << recenica << endl;
```

Ranije smo vidjeli da slična konstrukcija *nije moguća* sa običnim nizovima znakova, nego da se moraju koristiti ili dinamički stringovi (tj. objekti tipa “`string`”) ili konstrukcije poput sljedeće:

```

char recenica[100];
strcpy(recenica, "Ja sam prva rečenica...");
cout << recenica << endl;
strcpy(recenica, "A ja sam druga rečenica...");
cout << recenica << endl;

```

Veoma je važno uočiti bitnu razliku između navedena dva primjera. U prvom primjeru, "recenica" je pokazivač, koji prvo *pokazuje* na jednu stringovnu konstantu, a zatim na drugu. S druge strane, u drugom primjeru, "recenica" je *niz znakova* (za koji je rezerviran *prostor* od 100 mesta za znakove), u koji se prvo *kopira* jedna stringovna konstanta, a zatim druga. Ova razlika je veoma važna, mada na prvi pogled djeluje nebitna, jer je krajnji efekat isti. Međutim, kako je u prvom primjeru "recenica" pokazivač na *konstantne znakove*, izmjena sadržaja na koji on pokazuje nije moguća. Stoga bi u primjeru poput sljedećeg prevodilac prijavio grešku:

```

const char *ime;
ime = "Elma";
ime[0] = 'A';           // Ilegalno: "ime" pokazuje na KONSTANTNE znakove!
cout << ime;

```

Na ovome bi sva priča o pokazivačima na znakove mogla da završi da nije jedne nevolje. Naime, po posljednjem standardu jezika C++ kompjajleri *ne bi trebali da dozvole* da se običnim pokazivačima na (nekonstantne) znakove dodjeljuju adrese stringovnih konstanti. Kažemo "ne bi trebali", jer gotovo svi raspoloživi kompjajleri za C++ *dozvoljavaju* takve dodjele (s obzirom da su one dugi niz godina *načelno* bile dozvoljene, bez obzira što nisu smjele da budu – po ranijim standardima stringovne konstante zapravo uopće nisu bile *konstante*). Stoga će sljedeći primjer "proći nekažnjeno" kroz većinu raspoloživih C++ kompjajlera (i praktično *sve* kompjajlere za C):

```

char *ime;
ime = "Elma";
ime[0] = 'A';
cout << "Elma";

```

"Repertoar" mogućih posljedica ove skupine naredbi je raznovrstan. Možda se program "sruši" zbog toga što "ime" pokazuje na stringovnu konstantu koja je možda smještena u dio memorije koji je zabranjen za upis. Ukoliko to nije slučaj, gotovo sigurno da ćemo na ekranu umjesto "Elma" imati ispisano "Alma", bez obzira što naredba ispisa eksplisitno ispisuje tekst "Elma". Naime, obje pojave stringovne konstante "Elma" gotovo sigurno se u memoriji čuvaju na *istom mjestu* (sjetimo se *stapanja stringova*), pa će se izmjena sadržaja stringovne "konstante" (izmjena konstante je već sama po sebi absurdna) odraziti na svaku pojavu te stringovne konstante. Uglavnom, ovakvom nekonzistentnom dodjelom adrese stringovne konstante pokazivaču na nekonstantne znakove u mogućnosti smo da proizvedemo praktično iste neželjene efekte kao kada stringovnu konstantu proslijedimo funkciji koja vrši izmjenu sadržaja niza, o čemu smo ranije govorili. Navedimo još jedan katastrofalan primjer koji se može sresti kod početnika (i koji kompjajler *ne bi trebao* uopće da dozvoli):

```

char *recenica;
recenica = "Ja sam recenica...";
...
cin >> recenica;

```

Postavlja se pitanje *gdje će se u memoriju smjestiti* niz znakova koji unesemo sa tastature? Odgovor je jasan: na najgore moguće mjesto – preko znakova stringovne konstante "Ja sam rečenica..."! Pored toga, ukoliko je unesen i niz znakova *duži* od dužine te stringovne konstante, višak znakova će "pojesti" i sadržaj

memorije koji se nalazi iza ove stringovne konstante! Ukoliko nam operativni sistem odmah prekine ovakav program, imamo mnogo sreće. Gora varijanta je da program počne neispravno raditi tek nakon nekoliko sati, kad mu zatreba sadržaj uništenog dijela memorije!

Iz svega što je rečeno bitno je shvatiti da se dodjela stringovnih konstanti pokazivačima na znakove nipošto ne smije shvatiti kao lijepu i elegantnu zamjenu za funkciju “`strcpy`” (kako se propagira u nekim udžbenicima za jezik C) nego kao nešto što treba koristiti sa izuzetnim oprezom, i to obavezno sa kvalifikatorom “`const`”. Na taj način će nas kompjuler upozoriti pokušamo li izvršiti nedozvoljeni “napad” na memoriju. Napomenimo da su ovi komentari više namijenjeni onima koji imaju određena predznanja u jeziku C i žele da ta znanja nastave koristiti u jeziku C++. Za one kojima je C++ prvi jezik, nudimo veoma elegantno rješenje: za ozbiljniji rad sa stringovima ne koristite niti nul-terminirane nizove znakova niti pokazivače na znakove, već dinamičke stringove, odnosno tip “`string`”!

Mada je rečeno da pokazivačima nije moguće eksplisitno dodjeljivati brojeve, niti je moguće pokazivaču na jedan tip dodijeliti pokazivač na drugi tip, postoji indirektna mogućnost da se ovakva dodjela ipak izvrši, pomoću operatora za *pretvorbu tipova* (*typecasting operatora*). Na primjer, neka imamo sljedeće deklaracije:

```
int *pok_na_int;  
double *pok_na_double;
```

Tada pretvorba tipova dozvoljava, da na sopstveni rizik, izvršimo i ovakve dodjele:

```
pok_na_int = (int *) 3446;  
pok_na_double = (double *) pok_na_int;
```

Ovakve dodjele su veoma opasne. Prvom dodjelom smo eksplisitno postavili pokazivač “`pok_na_int`” da pokazuje na adresu 3446, mada ne znamo šta se tamo nalazi. Drugom dodjelom smo postavili pokazivač “`pok_na_double`” da pokazuje na istu adresu na koju pokazuje “`pok_na_int`”, što je također veoma opasno. Naime, ako se na nekoj lokaciji nalazi cijeli broj, tamo ne može u isto vrijeme biti realni broj. Pored toga, na različite tipove pokazivača pokazivačka aritmetika ne djeluje isto. Stoga su ovakve pretvorbe ostavljene samo onima koji *veoma dobro znaju šta njima žele da postignu*. Treba napomenuti da se pretvorbom broja u pokazivač dobija *pokazivač*, koji se dalje može dereferencirati kao i svaki drugi pokazivač. Stoga je naredba poput sljedeće sasvim legalna:

```
* (int *) 3446 = 50;
```

Ova naredba će na adresu 3446 u memoriju smjestiti broj 50 (posljedice ovakvih akcija preuzima programer, i u “normalnim” programima ih ne treba koristiti). Jezici C i C++ su rijetki jezici koji omogućavaju ovakvu slobodu u pristupima resursima računara, što ih čini idealnim za pisanje sistemskog softvera (naravno, za tu svrhu treba izuzetno dobro poznavati arhitekturu računara i operativnog sistema za koji se piše program). Stoga, ukoliko želite koristiti ovakve konstrukcije, provjerite da li ste *sigurni* da znate šta radite. Ukoliko utvrdite da ste sigurni, provjerite da li ste *sigurno sigurni*. Na kraju, ukoliko ste sigurni da ste sigurno sigurni, opet *ne radite to ako zaista ne morate!*

Već smo vidjeli da je, zahvaljujući pokazivačkoj aritmetici, funkcije koje barataju sa nizovima, poput više puta napisane funkcije “`IspisiNiz`”, moguće iskoristiti tako da se izvrše samo nad *dijelom niza*, koji ne počinje nužno od prvog elementa. Međutim, sve takve funkcije bile su *heterogene* po pitanju svojih parametara, odnosno njihovi parametri bili su različitih tipova (na primjer, jedan parametar funkcije “`IspisiNiz`” bio je sam niz, dok je drugi parametar bio broj elemenata koje treba ispisati). Za mnoge primjene je praktičnije imati funkcije sa *homogenom strukturom parametara*, odnosno funkcije čiji su parametri *istog tipa*. Na primjer, funkcija “`IspisiNiz`” mogla bi se napisati tako da njeni parametri budu

pokazivači na *prvi* i *posljednji* element koji će se ispisati. Ovako napisane funkcije su često ne samo jednostavnije za korištenje, već i jednostavnije za implementaciju, jer se na taj način može efikasno koristiti pokazivačka aritmetika. Zbog nekih tehničkih razloga, bolje je umjesto pokazivača na posljednji element koristiti pokazivač *iza posljednjeg elementa*. Na primjer, takva je sljedeća verzija generičke funkcije “`IspisiNiz`”, u kojoj parametri “`pocetak`” i “`iza_kraja`” predstavljaju respektivno pokazivač na prvi element niza koji treba ispisati i pokazivač koji pokazuje *iza posljednjeg elementa* koji treba ispisati:

```
template <typename Tip>
void IspisiNiz(Tip *pocetak, Tip *iza_kraja) {
    for(Tip *p = pocetak; p < iza_kraja; p++) cout << *p << endl;
}
```

Recimo, ukoliko želimo ispisati elemente niza “`niz`” sa indeksima 3, 4, 5, 6 i 7, možemo koristiti sljedeći poziv:

```
IspisiNiz(niz + 3, niz + 8);
```

Također, ukoliko isti niz ima “`n`” elemenata, tada čitav niz možemo ispisati pozivom poput

```
IspisiNiz(niz, niz + n);
```

Iz posljednjeg primjera je ujedno vidljivo zbog čega je praktično da drugi parametar pokazuje *iza posljednjeg elementa*, a ne tačno na njega (ovo nije jedini razlog). Naravno, istu funkciju bismo mogli koristiti za ispis elemenata vektora, ukoliko eksplicitno uzmemos adrese pojedinih elemenata pomoću operatora “`&`”. Na primjer, sljedeća dva poziva su analogna prethodnim pozivima za slučaj kada umjesto niza “`niz`” treba ispisati elemente vektora “`v`” (u istom opsegu kao u prethodnim primjerima):

```
IspisiNiz(&v[3], &v[8]);
IspisiNiz(&v[0], &v[n]);
```

Standardna biblioteka “`algorithm`” posjeduje čitavo mnoštvo funkcija za manipulaciju nizovima i vektorima (i drugim kontejnerskim strukturama koje ćemo upoznati kasnije) koje koriste upravo ovakvu konvenciju o parametrima kao što je iskorištena u posljednjoj verziji funkcije “`IspisiNiz`”. U tabeli koja slijedi prikazane su neke od najčešće korištenih funkcija iz ove biblioteke (kasnije, kada budemo govorili o nekim specifičnim primjenama kao što su sortiranje i pretraživanje, upoznaćemo još korisnih funkcija iz ove biblioteke). Sve opisane funkcije su posve jednostavne, i čitatelj odnosno čitateljka bi ih vjerovatno mogli lako napisati sami (što je preporučljivo uraditi kao korisnu vježbu). Međutim, kako se radi o operacijama koje se izuzetno često koriste u praktičnim primjenama, lijepo je znati da takve funkcije već postoje napisane, tako da ih možemo koristiti bez potrebe da svaki put “ponovo otkrivamo topalu vodu”. Također, sve spomenute funkcije napisane su veoma optimalno, tako da će vjerovatno raditi efikasnije nego analogne funkcije koje bismo eventualno napisali sami. U svim opisanim funkcijama, kada govorimo o “*bloku elemenata između pokazivača `p1` i `p2`*” mislimo da pokazivač `p1` pokazuje na *prvi element bloka*, a pokazivač `p2` pokazuje *tačno iza posljednjeg elementa bloka*, tačno onako kao što je bilo u posljednjoj napisanoj verziji funkcije “`IspisiNiz`”:

`copy(p1, p2, p3)`

Kopira elemente između pokazivača `p1` i `p2` na lokaciju određenu pokazivačem `p3` (uspust vraća kao rezultat pokazivač na poziciju tačno *iza* odredišnog bloka).

`copy_backward(p1, p2, p3)`

Kao “`copy`”, ali kopira elemente *unazad* odnosno redom od *posljednjeg* elementa ka *prvom*. Razlika između “`copy`” i “`copy_backward`” može biti važna ukoliko se

<code>fill(p1, p2, v)</code>	izvorni i odredišni blok preklapaju.
<code>fill_n(p, n, v)</code>	Popunjava sve elemente između pokazivača $p1$ i $p2$ sa vrijednošću v .
<code>swap_ranges(p1, p2, p3)</code>	Popunjava n elemenata počev od pokazivača p nadalje sa vrijednošću v .
<code>reverse(p1, p2)</code>	Razmjenjuje blok elemenata između pokazivača $p1$ i $p2$ sa blokom elemenata na koji pokazuje pokazivač $p3$ (uspust vraća kao rezultat pokazivač na poziciju tačno iza bloka na koji pokazuje $p3$).
<code>reverse_copy(p1, p2, p3)</code>	Izvrće blok elemenata između pokazivača $p1$ i $p2$ u obrnuti poredak, tako da prvi element postane posljednji, itd.
<code>rotate(p1, p2, p3)</code>	Kopira blok elemenata između pokazivača $p1$ i $p2$ u obrnutom poretku na lokaciju određenu pokazivačem $p3$ (tako da će posljednji element u izvornom bloku biti prvi element u odredišnom bloku, itd.). Sam izvorni blok ostaje neizmijenjen.
<code>rotate_copy(p1, p2, p3, p4)</code>	“Rotira” blok elemenata između pokazivača $p1$ i $p3$ ulijevo onoliko puta koliko je potrebno da element na koji pokazuje pokazivač $p2$ dođe na mjesto elementa na koji pokazuje pokazivač $p1$ (jedan korak rotacije se obavlja tako da krajnji lijevi element prelazi na posljednju poziciju, a svi ostali se pomjeraju za jedno mjesto ulijevo).
<code>replace(p1, p2, v1, v2)</code>	Isto kao “rotate”, ali kopira rotiranu verziju bloka elemenata na poziciju određenu pokazivačem $p4$, dok sam izvorni blok ostaje neizmijenjen.
<code>replace_copy(p1, p2, p3, v1, v2)</code>	Zamjenjuje sve elemente između pokazivača $p1$ i $p2$ koji imaju vrijednost $v1$ sa elementima sa vrijednošću $v2$.
<code>count(p1, p2, v)</code>	Kopira blok elemenata između pokazivača $p1$ i $p2$ na lokaciju određenu pokazivačem $p3$ uz zamjenu svakog elementa koji ima vrijednost $v1$ sa elementom koji ima vrijednost $v2$.
<code>equal(p1, p2, p3)</code>	Vraća kao rezultat broj elemenata između pokazivača $p1$ i $p2$ koji imaju vrijednost v .
<code>min_element(p1, p2)</code>	Vraća kao rezultat logičku vrijednost “ true ” ukoliko je blok elemenata između pokazivača $p1$ i $p2$ identičan po sadržaju bloku elemenata na koji pokazuje pokazivač $p3$, a u suprotnom vraća logičku vrijednost “ false ”.
<code>max_element(p1, p2)</code>	Vraća kao rezultat pokazivač na najmanji element u bloku između pokazivača $p1$ i $p2$.
<code>find(p1, p2, v)</code>	Vraća kao rezultat pokazivač na najveći element u bloku između pokazivača $p1$ i $p2$ koji ima vrijednost v . Ukoliko takav element ne postoji, vraća $p2$ kao rezultat.

```
remove(p1, p2, v)
```

“Uklanja” sve elemente iz bloka između pokazivača $p1$ i $p2$ koji imaju vrijednost v tako što ih premješta na kraj bloka. Kao rezultat vraća pokazivač na dio bloka u koji su premješteni uklonjeni elementi (tako da, na kraju, niti jedan element u bloku između pokazivača $p1$ i vraćenog pokazivača neće imati vrijednost v). Ukoliko niti jedan element nije imao vrijednost v , funkcija vraća $p2$ kao rezultat.

```
remove_copy(p1, p2, p3, v)
```

Kopira blok elemenata između pokazivača $p1$ i $p2$ na lokaciju određenu pokazivačem $p3$, uz izbacivanje elemenata koji imaju vrijednost v . Kao rezultat, funkcija vraća pokazivač koji pokazuje tačno iza posljednjeg elementa odredišnog bloka.

```
unique(p1, p2)
```

“Uklanja” sve elemente koji su jednaki elementu koji im prethodi iz bloka između pokazivača $p1$ i $p2$ tako što ih premješta na kraj bloka. Kao rezultat vraća pokazivač na dio bloka u koji su premješteni uklonjeni elementi (tako da na kraju u bloku između pokazivača $p1$ i vraćenog pokazivača neće biti susjednih elemenata koji su međusobno jednaki). Ukoliko u nizu nije bilo susjednih elemenata koji su međusobno jednaki, funkcija vraća $p2$ kao rezultat.

```
unique_copy(p1, p2, p3)
```

Kopira blok elemenata između pokazivača $p1$ i $p2$ na lokaciju određenu pokazivačem $p3$, uz izbacivanje elemenata koji su jednaki elementu koji im prethodi. Kao rezultat, funkcija vraća pokazivač koji pokazuje tačno iza posljednjeg elementa odredišnog bloka.

Kao vrlo jednostavan primjer primjene ovih funkcija, pretpostavimo da imamo dva niza “ $niz1$ ” i “ $niz2$ ” od po 10 realnih elemenata, i da je potrebno iskopirati sve elemente niza “ $niz1$ ” u niz “ $niz2$ ”. Umjesto da vršimo kopiranje element po element pomoću “**for**” petlje, možemo prosto iskoristiti funkciju “**copy**”, na sljedeći način:

```
copy(niz1, niz1 + 10, niz2);
```

Dalje, neka je elemente niza “ $niz1$ ” sa indeksima u opsegu od 4 do 9 potrebno pomjeriti za jedno mjesto *unazad*, tako da element sa indeksom 4 dođe na mjesto elementa sa indeksom 3 (recimo, sa ciljem brisanja elementa sa indeksom 3). To možemo uraditi sljedećim pozivom:

```
copy(niz1 + 4, niz1 + 10, niz1 + 3);
```

S druge strane, želimo li sve elemente niza “ $niz1$ ” sa indeksima u opsegu od 5 do 8 pomjeriti za jedno mjesto *unaprijed*, tako da element sa indeksom 5 dođe na mjesto elementa sa indeksom 6 (recimo, sa ciljem umetanja novog elementa na mjesto elementa sa indeksom 5), to možemo uraditi sljedećim pozivom (razmislite zbog čega je ovdje potrebno koristiti funkciju “**copy_backward**” umjesto “**copy**”):

```
copy_backward(niz1 + 5, niz1 + 9, niz1 + 6);
```

Kod svih opisanih funkcija koje kopiraju elemente na neko odredište (takve su sve navedene funkcije koje u svom imenu imaju riječ “**copy**”), isključiva je odgovornost programera da odredište sadrži dovoljno prostora da može primiti elemente koji se kopiraju. Ukoliko ovo nije ispunjeno, posljedice su

nepredvidljive i obično fatalne po program. Još treba napomenuti da su sve opisane funkcije napisane u izrazito generičkom stilu, tako da kao parametre ne primaju samo pokazivače, već i neke općenitije objekte od pokazivača kao što su tzv. *iteratori*, o kojima ćemo govoriti kasnije. Stoga je potrebno voditi računa da parametri koje prosljeđujemo ovim funkcijama imaju smisla. U protivnom, moguće su prijave vrlo čudnih grešaka od strane kompjajlera, a može se desiti i da ne bude prijavljena nikakva greška, ali da funkcija ne radi u skladu sa očekivanjima.

Interesantno je napomenuti da postoje ne samo pokazivači na jednostavne objekte, kao što su cijeli i realni brojevi ili znakovi, već je moguće napraviti pokazivače na praktično *bilo koje objekte*. Tako npr. postoje *pokazivači na nizove*, pa i *pokazivači na pokazivače* (tzv. *dvojni pokazivači*). Također se mogu napraviti *nizovi pokazivača*, *reference na pokazivače* i druge komplikirane strukture. Posebno su interesantni *pokazivači na funkcije* pomoću kojih se, u izvjesnom smislu, nekoj funkciji može druga funkcija proslijediti kao parametar, tako da je moguće napisati funkciju koja će pozvati neku drugu funkciju koja joj je proslijedena kao parametar. Drugim riječima, napisana funkcija do trenutka samog izvršavanja ne mora znati koju tačno funkciju treba da pozove. Zbog brojnih primjena ovakvih složenih pokazivačkih tipova, njima će biti posvećeno posebno poglavlje.

Priču o prostim pokazivačkim tipovima završićemo opisom tzv. *generičkih pokazivača*. Ovi pokazivači su se jako mnogo koristili u jeziku C, dok je njihova upotreba u jeziku C++ danas svedena na najnužniji minimum. Ipak, njihov opis može pomoći razumijevanju prave prirode pokazivačkih tipova. Generički pokazivači se definiraju kao pokazivači na “**void**”, i za njih se smatra da je *nepoznato* na koji tip tačno pokazuju. Stoga se oni ne mogu dereferencirati, niti se sa njima može vršiti pokazivačka aritmetika. Zapravo, ovi pokazivači su još ograničeniji: sa njima se ne može raditi *praktično ništa* sve dok se pomoću operatora za pretvorbu tipova ne pretvore u pokazivač u neki konkretni tip. Međutim, generičkom pokazivaču se može dodijeliti *ma koji drugi pokazivač*, i funkcija koja prima kao formalni parametar generički pokazivač prihvatiće kao stvarni parametar *bilo koji pokazivač*. Stoga su se generički pokazivači intenzivno koristili prije nego što su u jezik C++ uvedene generičke (šablonske) funkcije pomoću ključne riječi “**template**”. Naime, oni su ranije bili *jedini način* da se naprave funkcije koje mogu prihvati nizove *različitih tipova* kao parametre (u jeziku C oni su i dalje jedini način da se naprave funkcije koje, makar u izvjesnoj mjeri, posjeduju neka svojstva generičkih funkcija). Razmotrimo jedan konkretni primjer. Neka je potrebno napisati funkciju “*KopirajNiz*” koja posjeduje tri parametra “*odredisni*”, “*izvorni*” i “*br_elemenata*” i koja kopira “*br_elemenata*” elemenata iz niza “*izvorni*” u niz “*odredisni*”. Neka je dalje tu funkciju potrebno napisati tako da radi sa nizovima čiji su elementi proizvoljnog tipa. Uz pomoć mehanizma generičkih funkcija, kakve danas postoje u jeziku C++, ovaku funkciju je posve lako napisati. Ona bi, recimo, mogla izgledati poput sljedeće funkcije:

```
template <typename Tip>
void KopirajNiz(Tip odredisni[], Tip izvorni[], int br_elemenata) {
    for(int i = 0; i < br_elemenata; i++)
        odredisni[i] = izvorni[i];
}
```

Razmotrimo sada kako bi se isti problem mogao riješiti bez upotrebe generičkih funkcija, već samo upotrebom generičkih pokazivača. Ako bismo formalne parametre “*odredisni*” i “*izvorni*” modificirali tako da budu generički pokazivači (odnosno, pokazivači na “**void**”), takva funkcija bi kao odgovarajuće stvarne parametre prihvatala bilo koji pokazivač, pa samim tim (zbog automatske konverzije nizova u pokazivače) i bilo koji niz. Dakle, takva funkcija bi i dalje prihvatala proizvoljne nizove kao stvarne parametre. Međutim, sa generičkim pokazivačima se ne može raditi ništa dok se ne izvrši njihova pretvorba u pokazivač na neki konkretni tip. Sad se javlja prirodno pitanje u koji tip izvršiti njihovu pretvorbu. Naime, obična funkcija (tj. funkcija koja *nije generička*) nema nikakvog načina da sazna kojeg su tipa njeni stvarni parametri! Stoga je najprirodnije izvršiti njihovu konverziju u pokazivače na tip

“**char**”, s obzirom da je tip “**char**” *najelementarniji tip podataka* (svaki podatak tipa “**char**” uvijek zauzima *tačno jedan bajt*), a nakon toga obaviti kopiranje niza kao da se radi o običnom *nizu bajtova* (odnosno, obaviti kopiranje nizova *bajt po bajt*, onako kako su oni pohranjeni u memoriji). Međutim, tada se umjesto *broja elemenata* koje treba kopirati, kao parametar mora zadavati *broj bajtova* koje želimo kopirati. Tako napisana funkcija “**KopirajNiz**” mogla bi izgledati recimo ovako:

```
void KopirajNiz(void *odredisni, void *izvorni, int br_bajtova) {
    for(int i = 0; i < br_bajtova; i++)
        ((char *)odredisni)[i] = ((char *)izvorni)[i];
}
```

Sad, ako na primjer želimo kopirati 10 elemenata niza realnih brojeva “niz1” u niz “niz2”, mogli bismo koristiti sljedeći poziv:

```
KopirajNiz(niz2, niz1, 10 * sizeof(double));
```

Primijetimo kako je ovdje upotrijebljen operator “**sizeof**”, sa ciljem pretvaranja broja elemenata u broj bajtova. Naravno, umjesto izraza “**sizeof(double)**” mogli smo koristiti i izraz “**sizeof niz1[0]**” čime bismo poziv funkcije učinili neovisnim od stvarnog tipa elemenata niza. Vidimo da smo, na izvjestan način, uz pomoć generičkih pokazivača uspjeli napraviti funkciju koja se ponaša slično kao generičke funkcije, ali po cijenu da umjesto broja elemenata koji se kopiraju moramo zadavati broj bajtova koji se kopiraju. Uskoro ćemo vidjeti da to nije jedini nedostatak ovakvog pristupa.

Radi boljeg razumijevanja, navedimo i još jedan nešto složeniji primjer koji koristi generičke pokazivače. Pretpostavimo da treba napisati funkciju “**IzvrniNiz**” sa dva parametra “*niz*” i “*br_elemenata*” koja izvrće “*br_elemenata*” elemenata niza “*niz*”, čiji su elementi proizvoljnog tipa, u obrnuti poredak. Uz pomoć generičkih funkcija, rješenje ovog problema moglo bi se napisati na sljedeći način:

```
template <typename Tip>
void IzvrniNiz(Tip niz[], int br_elemenata) {
    for(int i = 0; i < br_elemenata / 2; i++) {
        Tip pomocna = niz[i];
        niz[i] = niz[br_elemenata - i - 1];
        niz[br_elemenata - i - 1] = pomocna;
    }
}
```

Napišimo sada funkciju koja obavlja sličan zadatak uz pomoć generičkih pokazivača. Jasno je da će sada parametar “*niz*” biti generički pokazivač. Međutim, ovdje se javlja dodatni problem u odnosu na funkciju iz prethodnog primjera. Da bismo izvrnuli elemente niza u obrnuti poredak, ne smijemo prosto izvrnuti *bajtove* od kojih je niz formiran u obrnuti poredak. Naime, ukoliko to učinimo, izvrnućemo i bajtovsku strukturu svakog individualnog elementa niza, što naravno ne smijemo uraditi. Dakle, bajtovska struktura svakog individualnog elementa niza ne smije se narušiti. Međutim, funkcija ne može ni na kakav način saznati tip elemenata niza, odakle slijedi da ne može saznati ni *koliko bajtova zauzima svaki od elemenata niza*. Ukoliko malo bolje razmislimo o svemu, zaključićemo da se problem ne može riješiti ukoliko u funkciju ne uvedemo *dodatajni parametar* (nazvan recimo “*vel_elementa*”) koji govori upravo koliko bajtova zauzima svaki od elemenata niza. Uz pomoć ovakvog dopunskog parametra, tražena funkcija se može napraviti uz malo “igranja” sa pokazivačkom aritmetikom. Jedno od mogućih rješenja moglo bi izgledati recimo ovako:

```
void IzvrniNiz(void *niz, int br_elemenata, int vel_elementa) {
    for(int i = 0; i < br_elemenata / 2; i++) {
```

```
char *p1 = (char *)niz + i * vel_elementa;
char *p2 = (char *)niz + (br_elemenata - i - 1) * vel_elementa;
for(int j = 0; j < vel_elementa; j++) {
    char pomocna = p1[j];
    p1[j] = p2[j]; p2[j] = pomocna;
}
```

Ukoliko sada želimo izvrnuti elemente niza “niz1” od 10 elemenata u obrnuti poredak, mogli bismo koristiti poziv poput sljedećeg:

```
IzvrniNiz(niz1, 10, sizeof niz1[0]);
```

Opisani primjeri nisu jednostavni, i traže izvjesno poznавање организације podataka u računarskoj memoriji. Čitatelju odnosno čitateljki se savjetuje da probaju ručno pratiti izvršavanje napisanih funkcija na nekom konkretnom primjeru. Napomenimo da su ovi primjeri ubačeni čisto sa ciljem pojašnjenja prave prirode pokazivača. Naime, u jeziku C++ ovakve primjere ne treba pisati, s obzirom da se razmotreni problemi mnogo lakše rješavaju pomoću *pravih* generičkih funkcija. Ovakve kvazi-generičke funkcije tipično zahtijevaju čudne ili dopunske parametre (poput parametra “*vel_elementa*” u posljednjem primjeru). Pored toga, nemoguće je na ovaj način napraviti funkciju koja bi, recimo, našla *najveći element niza*, ili *sumu svih elemenata u nizu*, s obzirom da na ovaj način mi uopće ne baratamo sa elementima niza, već sa *bajtovima* koje tvore elementi niza. Stoga je takav pristup moguć jedino za primjene u kojima tačna priroda elemenata niza nije ni bitna (kakva je npr. kopiranje).

Bez obzira na brojna ograničenja funkcija koje koriste generičke pokazivače, one su se mnogo koristile u jeziku C (jer bolja alternativa nije ni postojala). Stoga su mnoge funkcije iz biblioteka koje dolaze uz jezik C koristile ovaj pristup. Kako je jezik C++ naslijedio sve biblioteke koje su postojale u jeziku C, takve funkcije postoje i u jeziku C++. Na primjer, funkcija “*memcpy*” iz biblioteke “*cstring*” identična je posljednjoj napisanoj verziji funkcije “*KopirajNiz*”. Drugim riječima, kopiranje niza “niz1” od 10 realnih brojeva u niz “niz2” može se, u principu, obaviti i na sljedeći način:

```
memcpy(niz2, niz1, 10 * sizeof niz1[0]);
```

Također, ni funkcije koje kao jedan od parametara zahtijevaju veličinu elemenata niza u bajtovima (poput posljednje verzije funkcije “*IzvrniNiz*”) nisu rijetkost u standardnim bibliotekama. Takva je, na primjer, funkcija “*qsort*” iz biblioteke “*cstdlib*” koja služi za sortiranje elemenata niza u određeni poredak. Programeri u jeziku C intenzivno koriste ovakve funkcije. Mada se one, u načelu, mogu koristiti i u jeziku C++, postoje jaki razlozi da u jeziku C++ ove funkcije *ne koristimo* (ova napomena namijenjena je najviše onima koji imaju prethodno programersko iskustvo u jeziku C). Naime, sve funkcije zasnovane na generičkim pokazivačima svoj rad baziraju na činjenici da je manipulacija sa svim tipovima podataka moguće izvoditi *bajt po bajt*. Ta prepostavka je tačna za sve tipove podataka koji postoje u jeziku C, ali ne i za neke novije tipove podataka koji su uvedeni u jeziku C++ (kao što su svi tipovi podataka koji koriste tzv. *vlastiti konstruktor kopije*). Na primjer, mada će funkcija “*memcpy*” besprijekorno raditi za kopiranje nizova čiji su elementi tipa “**double**”, njena primjena na nizove čiji su elementi tipa “**string**” može imati fatalne posljedice. Isto vrijedi i za funkciju “*qsort*”, koja može napraviti pravu zbrku ukoliko se pokuša primijeniti za sortiranje nizova čiji su elementi dinamički stringovi. Tipovi podataka sa kojima se može sigurno manipulirati pristupanjem individualnim bajtima koje tvore njihovu strukturu nazivaju se *POD tipovi* (od engl. *Plain Old Data*). Na primjer, prosti tipovi kao što su “**int**”, “**double**” i klasični nizovi jesu POD tipovi, dok vektori i dinamički stringovi *isu*.

Opisani problemi sa korištenjem funkcija iz biblioteka naslijedjenih iz jezika C koje koriste generičke

pokazivače ne treba mnogo da nas zabrinjava, jer za svaku takvu funkciju u jeziku C++ postoje bolje alternative, koje garantirano rade u svim slučajevima. Tako, na primjer, u jeziku C++ umjesto funkcije “`memcpy`” treba koristiti već opisanu funkciju “`copy`” iz biblioteke “`algorithm`”, dok umjesto funkcije “`qsort`” treba koristiti funkciju “`sort`” (također iz biblioteke “`algorithm`”), o kojoj ćemo govoriti kasnije. Ove funkcije, ne samo da su univerzalnije od srodnih funkcija naslijedenih iz jezika C, već su i jednostavnije za upotrebu. Na kraju, moramo dati još samo jednu napomenu. Programeri koji posjeduju iskustvo u jeziku C teško se odriču upotrebe funkcije “`memcpy`” i njoj srodnih funkcija (poput “`memset`”), jer im je poznato da su one, zahvaljujući raznim trikovima na kojima su zasnovane, *vrlo efikasne* (mnogo efikasnije nego upotreba obične “`for`” petlje). Međutim, bitno je znati da odgovarajuće funkcije iz biblioteke “`algorithm`”, poput “`copy`” (ili “`fill`”, koju treba koristiti umjesto “`memset`”) tipično nisu *ništa manje efikasne*. Zapravo, u mnogim implementacijama biblioteke “`algorithm`” se poziv funkcija poput “`copy`” automatski prevodi u poziv funkcija poput “`memcpy`” ukoliko se ustanovi da je takav poziv *siguran*, odnosno da tipovi podataka sa kojima se barata predstavljaju POD tipove. Stoga, programeri koji sa jezika C prelaze na C++ ne trebaju osjećati nelagodu prilikom prelaska sa upotrebe funkcija poput “`memcpy`” na funkcije poput “`copy`”. Jednostavno, treba prihvatići činjenicu: funkcije poput “`memcpy`”, “`memset`”, “`qsort`” i njima slične *zaista više ne treba koristiti* u C++ programima (osim u vrlo iznimnim slučajevima). One su zadržane pretežno sa ciljem da omoguće kompatibilnost sa već napisanim programima koji su bili na njima zasnovani!

26. Dinamička alokacija memorije

Do sada smo pokazivače koristili isključivo tako što smo im dodjeljivali adrese objekata za koje je već *odranje postojao zauzet memorjski prostor*, na primjer, adresu neke druge promjenljive, nekog elementa niza ili stringovne konstantne. Na taj način, pokazivači su djelovali samo kao neko egzotično sredstvo pomoću kojeg su se doduše mogli postići interesantni efekti i ostvariti prilična optimizacija kôda, ali koji ne donose ništa radikalno novo. Zaista, sve što je demonstrirano u prethodnom poglavlju o pokazivačima, moglo bi se u načelu u jeziku C++ ostvariti i bez njih, eventualno na neki duži način, pogotovo nakon što je u jezik C++ uveden prenos parametara po referenci (prije uvođenja referenci, sličan efekat nije bio izvodiv bez upotrebe pokazivača) i generičke (šablonske) funkcije (koje su se ranije simulirale preko generičkih pokazivača, uz upotrebu veoma rogobatnih konstrukcija).

Pravu snagu pokazivači dobijaju tek u kombinaciji sa *dinamičkom alokacijom (dodjelom) memorije*. Pod dinamičkom alokacijom memorije podrazumijevamo mogućnost da program *u toku izvršavanja* zatraži da mu se dodijeli određena količina memorije koja ranije nije bila zauzeta, i kojom on može raspolagati sve dok eventualno ne zatraži njenoslobađanje. Takvim, dinamički dodijeljenim blokovima memorije, može se pristupati *isključivo pomoću pokazivača*. Dinamička alokacija memorije postojala je još u jeziku C, i ostvarivala se pozivom funkcije “`malloc`“ ili neke srodne funkcije iz biblioteke “`cstdlib`“ (zaglavljene ove biblioteke u starijim verzijama kompjajlera za C++ zvalo se “`stdlib.h`“, što je konvencija koja još uvijek vrijedi u jeziku C). Mada ovaj način za dinamičku alokaciju načelno radi i u jeziku C++, C++ nudi mnogo fleksibilnije načine, tako da u C++ programima po svaku cijenu treba izbjegavati načine za dinamičku dodjelu memorije naslijedene iz jezika C.

U jeziku C++, preporučeni način za dinamičku alokaciju memorije je pomoću operatora “`new`“. Ovaj operator ima mnogo različitih oblika. U svom osnovnom obliku, iza operatorka “`new`“ treba da slijedi *ime nekog tipa*. Operator “`new`“ će tada potražiti u memoriji slobodno mjesto u koje bi se mogao smjestiti podatak navedenog tipa. Ukoliko se takvo mjesto *pronade*, operator “`new`“ će kao rezultat vratiti *pokazivač na pronađeno mjesto u memoriji*. Pored toga, pronađeno mjesto u memoriji će biti označeno kao *zauzeto*, tako da se kasnije neće moći desiti da isti prostor memorije slučajno bude upotrijebљen za neku drugu namjenu. Ukoliko je sva memorija već zauzeta, operator “`new`“ će *baciti izuzetak*, koji možemo “uhvatiti”. Raniji standardi jezika C++ predviđali su da operator “`new`“ u slučaju neuspjeha vrati kao rezultat *nul-pokazivač*, ali je ISO C++ 98 standard precizirao da u slučaju neuspjeha operator “`new`“ baca izuzetak (razloge nije teško naslutiti: programeri su zaboravljali ili su bili lijeni da nakon svake upotrebe operatorka “`new`“ provjeravaju da li je vraćena vrijednost možda nul-pokazivač, što je, u slučaju da alokacija ne uspije, moglo imati ozbiljne posljedice).

Osnovnu upotrebu operatorka “`new`“ najbolje je ilustrirati na konkretnom primjeru. Prepostavimo da nam je data sljedeća deklaracija:

```
int *pokazivac;
```

Ovom deklaracijom definirana je pokazivačka promjenljiva “`pokazivac`”, koja ne pokazuje ni na šta konkretno (divlji pokazivač), ali koja, u načelu, treba da pokazuje na *cijeli broj*. Stanje memorije nakon ove deklaracije možemo predstaviti sljedećom slikom (adrese su prepostavljene proizvoljno):

Adrese	...	3758	3759	3760	3761	3762	3763	3764	3765	...
				???						

pokazivac

Ukoliko sad izvršimo naredbu

```
pokazivac = new int;
```

operator “**new**“ će potražiti slobodno mjesto u memoriji koje je dovoljno veliko da prihvati *jedan cijeli broj*. Ukoliko se takvo mjesto pronađe, njegova adresa će biti vraćena kao rezultat operatora “**new**”, i dodijeljena pokazivaču “pokazivac”. Pretpostavimo, na primjer, da je slobodno mjesto pronađeno na adresi 3764. Tada će rezultat operatora “**new**“ biti *pokazivač na cijeli broj koji pokazuje na adresu 3764* (ovaj rezultat bismo mogli zapisati kao izraz “(**int ***) 3764”), koji se može dodijeliti pokazivaču “pokazivac”, tako da će on pokazivati na adresu 3764. Stanje u memoriji će sada izgledati ovako:

Adrese	...	3758	3759	3760	3761	3762	3763	3764	3765	...
				3764						

pokazivac

Pored toga, lokacija 3764 postaje *zauzeta*, u smislu da će biti evidentirano da je ova lokacija rezervirana za upotrebu od strane programa, i ona do daljnog sigurno neće biti iskorištena za neku drugu namjenu. Stoga je sasvim sigurno pristupiti njenom sadržaju putem pokazivača “pokazivac” (on više nije divlji pokazivač). Stoga se dereferencirani pokazivač “pokazivac” u potpunosti ponaša *kao promjenljiva*, iako lokacija na koju on pokazuje *nema svoje vlastito simboličko ime*. Stoga su naredbe poput sljedećih posve korektne (i ispisaće redom vrijednosti “5” i “18”):

```
*pokazivac = 5;
cout << *pokazivac << endl;
*pokazivac = 3 * *pokazivac + 2;
(*pokazivac)++;
cout << *pokazivac << endl;
```

Kako se lokacija rezervirana pomoću operatora “**new**” u potpunosti ponaša kao promjenljiva (osim što nema svoje vlastito ime), kažemo da ona predstavlja *dinamičku promjenljivu*. Dakle, dinamičke promjenljive se *stvaraju* primjenom operatora “**new**”, i njihovom sadržaju se može pristupiti *isključivo putem pokazivača* koji na njih pokazuju. Pored *automatskih promjenljivih*, koje se automatski stvaraju na mjestu deklaracije i automatski uništavaju na kraju bloka u kojem su deklarirane (takve su sve *lokalne promjenljive* osim onih koje su deklarirane sa atributom “**static**”), i *statičkih promjenljivih* koje “žive” cijelo vrijeme dok se program izvršava (takve su sve *globalne promjenljive* i lokalne promjenljive deklarirane sa atributom “**static**”), dinamičke promjenljive se mogu smatrati za treću vrstu promjenljivih koje postoje. One se *stvaraju na zahtjev*, i kao što ćemo uskoro vidjeti, *uništavaju na zahtjev*. Samo, za razliku od automatskih i statičkih promjenljivih, dinamičke promjenljive nemaju imena (stoga im se može pristupiti samo pomoću pokazivača).

Sadržaj dinamičkih promjenljivih je nakon njihovog stvaranja tipično *nedefiniran*, sve dok im se ne izvrši prva dodjela vrijednosti (putem pokazivača). Preciznije, sadržaj zauzete lokacije zadržava svoj raniji sadržaj, jedino što se lokacija označava kao zauzeta. Međutim, moguće je izvršiti inicijalizaciju dinamičke promjenljive *odmah po njenom stvaranju*, tako što iza oznake tipa u operatoru “**new**” u zagradama navedemo izraz koji predstavlja željenu *inicijalnu vrijenost promjenljive*. Na primjer, sljedeća naredba će stvoriti novu cjelobrojnu dinamičku promjenljivu, inicijalizirati njen sadržaj na vrijednost “5”, i postaviti pokazivač “pokazivac” da pokazuje na nju:

```
pokazivac = new int(5);
```

Sada će stanje u memoriji biti kao na sljedećoj slici:

Adrese	...	3758	3759	3760	3761	3762	3763	3764	3765	...
				3764				5		

pokazivac

Razumije se da se rezultat operatora “**new**“ mogao odmah iskoristiti za inicijalizaciju pokazivača “pokazivac“ već prilikom njegove deklaracije. Drugim riječima, sljedeća konstrukcija je posve legalna:

```
int *pokazivac = new int(5);
```

Isto vrijedi i za sljedeću konstrukciju, u kojoj je iskorištena druga (konstruktorska) sintaksa za inicijalizaciju promjenljive:

```
int *pokazivac(new int(5));
```

Zapravo, postoji jedan indirektan način na koji možemo dati *ime* novostvorenoj dinamičkoj promjenljivoj. Naime, za dinamičku promjenljivu je, kao i za svaku drugu promjenljivu, moguće vezati *referencu*. Tako, ukoliko je “pokazivac” pokazivačka promjenljiva koja pokazuje na novostvorenu dinamičku promjenljivu, kao u prethodnom primjeru, moguće je izvršiti sljedeću deklaraciju:

```
int &dinamicka = *pokazivac;
```

Na ovaj način smo kreirali referencu “*dinamicka*” koja je vezana za novostvorenu dinamičku promjenljivu, i koju, prema tome, možemo smatrati kao alternativno ime te dinamičke promjenljive (zapravo, *jedino ime*, s obzirom da ona svog vlastitog imena i nema). U suštini, isti efekat smo mogli postići i vezivanjem reference direktno na dereferencirani rezultat operatora “**new**” (bez posredstva pokazivačke promjenljive). Naime, rezultat operatora “**new**” je *pokazivač*, dereferencirani pokazivač je *l-vrijednost*, a na svaku l-vrijednost se može vezati referenca odgovarajućeg tipa:

```
int &dinamicka = *new int(5);
```

Ovakve konstrukcije se ne koriste osobito često, mada se ponekad mogu korisno upotrijebiti (npr. dereferencirani rezultat operatora “**new**” može se prenijeti u funkciju kao parametar po referenci). U svakom slučaju, opis ovih konstrukcija pomaže da se shvati prava suština pokazivača i referenci.

Prilikom korištenja operatora “**new**”, treba voditi računa da uvijek postoji mogućnost da on *baci izuzetak*. Doduše, vjerovatnoća da u memoriji neće biti pronađen prostor za jedan jedini cijeli broj veoma je mala, ali se treba naviknuti na takvu mogućnost, jer ćemo uskoro dinamički alocirati znatno glomaznije objekte (npr. nizove) za koje lako može nestati memorije. Stoga bi se svaka dinamička alokacija trebala izvoditi unutar “**try**“ bloka, na način koji je principijelno prikazan u sljedećem isječku:

```
try {
    int *pokazivac = new int(5);
    ...
}
catch(...) {
    cout << "Problem: Nema dovoljno memorije!\n";
}
```

Tip izuzetka koji se pri tome baca je tip “`bad_alloc`”. Ovo je izvedeni tip podataka (poput tipa “`string`” i drugih izvedenih tipova podataka) definiran u biblioteci “`new`”. Tako, ukoliko želimo da specifiziramo da hvatamo baš izuzetke ovog tipa, možemo koristiti sljedeću konstrukciju (pod uvjetom da smo uključili zaglavlj biblioteke “`new`” u program):

```
try {
    int *pokazivac = new int(5);
    ...
}
catch(bad_alloc) {
    cout << "Problem: Nema dovoljno memorije!\n";
}
```

Primijetimo da nismo imenovali parametar tipa “`bad_alloc`”, s obzirom da nam on nije ni potreban. Inače, specifikacija tipa “`bad_alloc`” unutar “`catch`” bloka je korisna ukoliko želimo da razlikujemo izuzetke koje baca operator “`new`” od drugih izuzetaka. Ukoliko smo sigurni da jedini mogući izuzetak unutar “`try`” bloka može poteći samo od operatora “`new`”, možemo koristiti varijantu “`catch`” bloka sa tri tačke umjesto formalnog parametra, što ćemo ubuduće pretežno koristiti.

Dinamičke promjenljive se mogu ne samo *stvarati*, već i *uništavati* na zahtjev. Onog trenutka kada zaključimo da nam dinamička promjenljiva na koju pokazuje pokazivač više nije potrebna, možemo je uništiti primjenom operatora “`delete`”, iza kojeg se prosto navodi pokazivač koji pokazuje na dinamičku promjenljivu koju želimo da uništimo. Na primjer, naredbom

```
delete pokazivac;
```

uništićemo dinamičku promjenljivu na koju pokazivač “`pokazivac`” pokazuje. Pri tome je važno da razjasnimo šta se podrazumijeva pod tim *uništavanjem*. Pokazivač “`pokazivac`” će i dalje pokazivati na istu adresu na koju je pokazivao i prije, samo što će se izbrisati evidencija o tome da je ta lokacija zauzeta. Drugim riječima, ta lokacija može nakon uništavanja dinamičke promjenljive biti iskorištena od strane nekog drugog (npr. operativnog sistema), pa čak i od strane samog programa za neku drugu svrhu (na primjer, sljedeća primjena operatora “`new`” može ponovo iskoristiti upravo taj prostor za stvaranje neke druge dinamičke promjenljive). Stoga, više nije sigurno pristupati sadržaju na koju “`pokazivac`” pokazuje, odnosno on je ponovo postao *divlji pokazivač*. Divlji pokazivači koji pokazuju na objekte koji su iz bilo kojeg razloga *prestali da postoje* (jedan od mogućih razloga je njihovo eksplicitno uništavanje pozivom operatora “`delete`”) nazivaju se *viseći pokazivači* (engl. *dangling pointers*).

Treba napomenuti da viseći pokazivači mogu nastati i nevezano od dinamičke alokacije memorije. Sljedeći primjer pokazuje kako nesvesno možemo stvoriti viseći pokazivač:

```
int *pokazivac;
{
    int lokalna = 5;
    pokazivac = &lokalna;
}
*pokazivac = 6;
```

U ovom primjeru, pokazivač “`pokazivac`” je unutar bloka postavljen da pokazuje na lokalnu promjenljivu “`lokalna`” koja je deklarirana unutar bloka. Međutim, po završetku bloka, lokalna promjenljiva prestaje postojati, tako da je “`pokazivac`” postao viseći pokazivač, jer pokazuje na promjenljivu koja je prestala da postoji (situacija je analogna kao pri brisanju dinamičke

promjenljive pomoću operatora “**delete**”: promjenljiva “pokazivac“ i dalje pokazuje na isto mjesto u memoriji, ali program više ne smatra da to mjesto pripada nekoj promjenljivoj). Treba se čuvati visećih pokazivača, jer su oni (kao i uostalom svi divlji pokazivači) veoma čest uzrok fatalnih grešaka u programima, koje se teško otkrivaju, s obzirom da im se posljedice obično uoče tek naknadno. Zapravo, ranije smo vidjeli da postoje i viseće reference, koje su jednako opasne kao i viseći pokazivači, samo što je viseći pokazivač, na žalost, mnogo lakše “napraviti” od viseće reference!

Sve dinamičke promjenljive se automatski uništavaju po završetku programa. Međutim, bitno je napomenuti da ni jedna dinamička promjenljiva *neće biti uništena sama od sebe* prije završetka programa, osim ukoliko je eksplicitno ne uništimo primjenom operatora “**delete**”. Po tome se one bitno razlikuju od *automatskih promjenljivih*, koje se automatski uništavaju na kraju bloka u kojem su definirane. Ukoliko smetnemo sa uma ovu činjenicu, možemo zapasti u probleme. Pretpostavimo, na primjer, da smo izvršili sljedeću sekvencu naredbi:

```
int *pokazivac = new int;
*pokazivac = 5;
pokazivac = new int;
*pokazivac = 3;
```

U ovom primjeru, prvo se stvara jedna dinamička promjenljiva (recimo, na adresi 3764), nakon čega se njen sadržaj postavlja na vrijednost “5”. Iza toga slijedi nova dinamička alokacija kojom se stvara nova dinamička promjenljiva (recimo, na adresi 3765), a njen sadržaj se postavlja na vrijednost “3”. Pri tome, pokazivač “pokazivac“ pokazuje na novostvorenu dinamičku promjenljivu (na adresi 3765). Međutim, dinamička promjenljiva na adresi 3764 *nije uništena*, odnosno dio memorije u kojem se ona nalazi još uvijek se smatra zauzetim. Kako pokazivač “pokazivac“ više ne pokazuje na nju, njenom sadržaju više ne možemo pristupiti preko ovog pokazivača. Zapravo, na ovu dinamičku promjenljivu više ne pokazuje *niko*, tako da je ova dinamička promjenljiva postala *izgubljena* (niti joj možemo pristupiti, niti je možemo uništiti). Ova situacija prikazana je na sljedećoj slici:

Adrese	...	3758	3759	3760	3761	3762	3763	3764	3765	...
pokazivac				3765				5	3	

Dio memorije koji zauzima izgubljena dinamička promjenljiva ostaje rezerviran sve do kraja programa, i trajno je izgubljen za program. Ovakva pojava naziva se *curenje memorije* (engl. *memory leak*) i predstavlja dosta čestu grešku u programima. Ova greška je na neki način suprotna greškama uslijed upotrebe divljih i visećih pokazivača (kod divljih i visećih pokazivača imamo pokazivače koji ne pokazuju ni na kakav smislen prostor, dok kod curenja memorije imamo rezerviran prostor na kojeg niko ne pokazuje). Mada je curenje memorije manje fatalno u odnosu greške koje nastaju uslijed divljih ili visećih pokazivača, ono se također teško uočava i može da dovede do ozbiljnih problema. Razmotrimo na primjer sljedeću sekvencu naredbi:

```
int *p;
for(int i = 1; i <= 30000; i++) {
    p = new int;
    *p = i;
}
```

U ovom primjeru, unutar “**for**“ petlje je stvoreno 30000 dinamičkih promjenljivih, jer je svaka

primjena operatora “**new**“ stvorila novu dinamičku promjenljivu, od kojih niti jedna nije uništena (s obzirom da nije korišten operator “**delete**”). Međutim, od tih 30000 promjenljivih, 29999 je izgubljeno, jer na kraju pokazivač “**p**“ pokazuje samo na posljednju stvorenu promjenljivu! Uz pretpostavku da jedna cjelobrojna promjenljiva zauzima 4 bajta, ovim smo bespotrebno izgubili 119996 bajta memorije, koje ne možemo osloboditi, sve dok se ne oslobode automatski po završetku programa!

Jedna od tipičnih situacija koje mogu dovesti do curenja memorije je stvaranje dinamičke promjenljive unutar neke funkcije preko pokazivača koji je lokalna automatska (nestatička) promjenljiva unutar te funkcije. Ukoliko se takva dinamička promjenljiva ne uništi prije završetka funkcije, pokazivač koji na nju pokazuje biće uništen (poput svake druge automatske promjenljive), tako da će ona postati izgubljena. Na primjer, sljedeća funkcija demonstrira takvu situaciju:

```
void Curenje(int n) {  
    int *p = new int;  
    *p = n;  
}
```

Prilikom svakog poziva ove funkcije stvara se nova dinamička promjenljiva, koja postaje posve nedostupna nakon završetka funkcije, s obzirom da se pokazivač koji na nju pokazuje uništava. Ostatak programa nema mogućnost niti da pristupi takvoj promjenljivoj, niti da je uništi. Stoga svaki poziv funkcije “Curenje“ stvara novu izgubljenu promjenljivu, odnosno prilikom svakog poziva ove funkcije količina izgubljene memorije se povećava. Stoga bi svaka funkcija koja stvori neku dinamičku promjenljivu trebala i da je uništi. Izuzetak od ovog pravila može imati smisla jedino ukoliko se novostvorena dinamička promjenljiva stvara putem *globalnog pokazivača* (kojem se može pristupiti i izvan funkcije), ili ukoliko funkcija *vraća kao rezultat* pokazivač na novostvorenu promjenljivu. Na primjer, sljedeća funkcija može imati smisla:

```
int *Stvori(int n) {  
    int *p = new int(n);  
    return p;  
}
```

Ova funkcija stvara novu dinamičku promjenljivu, inicijalizira je na vrijednost zadalu parametrom, i *vraća kao rezultat* pokazivač na novostvorenu promjenljivu (zanemarimo ovdje činjenicu da je ova funkcija posve beskorisna, s obzirom da ne radi ništa više u odnosu na ono što već radi sam operator “**new**”). Na taj način omogućeno je da rezultat funkcije bude dodijeljen nekom pokazivaču, preko kojeg će se kasnije moći pristupiti novostvorenoj dinamičkoj promjenljivoj, i eventualno izvršiti njeno uništavanje. Na primjer, sljedeći slijed naredbi je posve smislen:

```
int *pokazivac;  
pokazivac = Stvori(10);  
cout << *Pokazivac;  
delete Pokazivac;
```

Napomenimo da se funkcija “**Stvori**“ mogla napisati i kompaktnije, bez deklariranja lokalnog pokazivača:

```
int *Stvori(int n) {  
    return new int(n);  
}
```

Naime, “**new**“ je *operator* koji vraća pokazivač kao rezultat, koji kao takav smije biti vraćen kao rezultat iz funkcije. Također, interesantno je napomenuti da je na prvi pogled veoma neobična konstrukcija

```
*stvori(5) = 8;
```

sintaksno posve ispravna (s obzirom da funkcija “*stvori*“ vraća kao rezultat *pokazivač*, a dereferencirani pokazivač je *l-vrijednost*), mada je logički smisao ovakve konstrukcije prilično upitan (prvo se stvara dinamička promjenljiva sa vrijednošću 5, nakon toga se njena vrijednost mijenja na 8, i na kraju se gubi svaka veza sa dinamičkom promjenljivom, jer pokazivač na nju nije nigdje sačuvan). Ipak, mogu se pojaviti situacije u kojima slične konstrukcije mogu biti od koristi, stoga je korisno znati da su one moguće.

Kreiranje individualnih dinamičkih promjenljivih nije od osobite koristi ukoliko se kreiraju promjenljive prostih tipova (kao što su npr. cjelobrojne promjenljive), tako da će kreiranje individualnih dinamičkih promjenljivih postati interesantno tek kada razmotrimo složenije tipove podataka, kakvi su *strukture* i *klase*. Međutim, znatno je interesantnija činjenica da se pomoću dinamičke alokacije memorije mogu kreirati *dinamički nizovi*, i to čija veličina *nije unaprijed poznata!* Za tu svrhu koristi se talođer operator “**new**“ iza kojeg ponovo slijedi *ime tipa* (koje ovaj put predstavlja *tip elemenata niza*), nakon čega u *uglastim zagradama* slijedi broj elemenata niza koji kreiramo. Međutim, za razliku od deklaracije običnih (statičkih) nizova, traženi broj elemenata niza *ne mora biti konstanta*, već može biti proizvoljan izraz. Operator “**new**“ će tada potražiti slobodno mjesto u memoriji u koje bi se mogao smjestiti niz tražene veličine navedenog tipa, i vratiti kao rezultat pokazivač na pronađeno mjesto u memoriji, ukoliko takvo postoji (u suprotnom će biti bačen izuzetak tipa “*bad_alloc*”). Na primjer, ukoliko je promjenljiva “*pokazivac*“ deklarirana kao pokazivač na cijeli broj (kao i u dosadašnjim primjerima), tada će naredba

```
pokazivac = new int[5];
```

potražiti prostor u memoriji koji je dovoljan da prihvati *pet cjelobrojnih vrijednosti*, i u slučaju da pronađe takav prostor, *dodijeliće njegovu adresu* pokazivaču “*pokazivac*“ (bitno je naglasiti da u uglastoj zagradi nije morala biti konstanta “5”, već proizvoljan cjelobrojni izraz, koji može sadržavati i promjenljive). Na primjer, neka je pronađen prostor na adresi 4433, a neka se sama pokazivačka promjenljiva “*pokazivac*“ nalazi na adresi 4430. Tada memorijska slika nakon uspješnog izvršavanja prethodne naredbe izgleda kao na sljedećoj slici (radi jednostavnosti je prepostavljeno da jedna cjelobrojna promjenljiva zauzima jednu memorijsku lokaciju, što u stvarnosti nije ispunjeno):

<i>Adrese</i>	...	4430	4431	4432	4433	4434	4435	4436	4437	...
		4433								

pokazivac

Ovako kreiranom dinamičkom nizu možemo pristupiti samo preko pokazivača. Međutim, kako se na pokazivače mogu primjenjivati operatori indeksiranja (podsjetimo se da se izraz “*p[n]*“ u slučaju kada je “*p*“ pokazivač interpretira kao “** (p+n)*“ uz primjenu pokazivačke

aritmetike), dinamički niz možemo koristiti *na posve isti način kao i obični niz*, pri čemu *umjesto imena niza koristimo pokazivač*. Na primjer, da bismo postavili sve elemente novokreiranog dinamičkog niza na nulu, možemo koristiti “**for**“ petlju kao da se radi o običnom nizu:

```
for(int i = 0; i < 5; i++) pokazivac[i] = 0;
```

Stoga, sa aspekta programera gotovo da nema nikakve razlike između korištenja običnih i dinamičkih nizova. Neko bi se mogao zapitati da li je pokazivačka aritmetika nad pokazivačem “pokazivac” legalna, s obzirom da on ne pokazuje na element nekog postojećeg niza. Odgovor je potvrđan. Naime, po standardu jezika C++, prostor alociran pomoću operatora “**new**” zaista se tretira kao niz, i pokazivačka aritmetika će garantirano biti valjana sve dok pokazivač ne “odluta” izvan prostora alociranog pomoću operatora “**new**” (eventualno se dozvoljava pozicioniranje pokazivača tik iza alociranog prostora, kao i u slučaju klasičnih nizova).

Strogo rečeno, dinamički nizovi *nemaju imena* i njima se može pristupati samo preko pokazivača koji pokazuju na njihove elemente (eventualno bi im se ime moglo dodijeliti vezanjem *reference* na njih, ali to se rijetko radi). Tako, ukoliko izvršimo deklaraciju poput

```
int *dynamicki_niz = new int[100];
```

mi zapravo stvaramo *dva objekta*: dinamički niz od 100 cijelih brojeva koji *nema ime*, i pokazivač “*dynamicki_niz*” koji je inicijaliziran tako da pokazuje na *prvi element ovako stvorenog dinamičkog niza*. Međutim, kako promjenljivu “*dynamicki_niz*” možemo koristiti na gotovo isti način kao da se radi o običnom nizu (pri čemu, zahvaljujući pokazivačkoj aritmetici, preko nje zaista pristupamo dinamičkom nizu), dopustićemo sebi izvjesnu slobodu izražavanja i ponekad ćemo, radi kratkoće izražavanja, govoriti da promjenljiva “*dynamicki_niz*” predstavlja dinamički niz (iako je prava istina da je ona zapravo pokazivač koji pokazuje na prvi element dinamičkog niza).

Razmotrimo ovu korespondenciju između dinamičkih nizova i pokazivača na njihov prvi element nešto detaljnije. Ranije smo vidjeli da se pokazivači i nizovi mogu koristiti gotovo na identičan način. Najuočljivija razlika sa aspekta programera je činjenica da ime niza (običnog) uvijek pokazuje na *prvi element niza*, tako da nad imenom niza ne možemo izvršiti niti jednu operaciju koja bi dovela do promjene adrese na koju on pokazuje (poput dodjele, ili operatora poput “**=**” ili “**++**”). U tom smislu, nizovi se ponašaju veoma slično kao *konstantni pokazivači*. Zaista, sa aspekta programera gotovo da nema nikakve razlike između deklaracije statičkog niza poput

```
int niz[100];
```

i sljedeće deklaracije, u kojoj se koristi konstantni pokazivač, inicijaliziran da pokazuje na dinamički dodijeljenu memoriju:

```
int *const niz = new int[100];
```

U oba slučaja, programer može koristiti promjenljivu “*niz*“ na isti način. Ipak, prethodne dvije deklaracije dovode do potpuno različitih memorijskih slika. U prvom slučaju, rezervira se prostor za 100 elemenata niza u memoriji, a promjenljiva “*niz*“ upotrijebljena bez indeksa po potrebi se automatski konvertira u pokazivač na početak tako rezerviranog bloka memorije. Ovo

je ilustrirano na sljedećoj slici:

Adrese	...	1230	1231	1232	1233	1234	1235	...	1329	...
niz										

U skladu sa ovim primjerom, kada računar treba da izvrši pristup elementu “niz[2]”, on će uzeti adresu na kojoj se nalazi niz (1230 u navedenom primjeru), zatim je sabrati sa 2 (radi jednostavnosti, ovdje je pretpostavljeno da svaki element niza zauzima jednu lokaciju, što gotovo sigurno nije slučaj) i na taj način dobiti adresu na kojoj se nalazi traženi element (1232 u ovom primjeru). S druge strane, u drugom slučaju u memoriji na početku imamo rezerviran prostor samo za pokazivač “niz”, nakon čega se pozivom operatora “**new**” pronalazi prostor za smještanje 100 elemenata niza, i adresa nađenog prostora dodjeljuje pokazivaču “niz”. Ovo je ilustrirano na sljedećoj slici:

Adrese	...	1230	1231	1232	1233	1234	1235	...	1331	...
niz		1232								

U ovom slučaju će računar kada treba da izvrši pristup elementu “niz[2]” prvo uzeti adresu na kojoj se nalazi pokazivač “niz” (1230 u navedenom primjeru), zatim pročitati sadržaj te adrese da utvrdi na koju adresu pokazivač pokazuje (1232 u navedenom primjeru), i tek tada sabrati sadržaj koji je pročitan sa 2, čime se dobija adresa traženog elementa (1234). Vidimo da se u slučaku kada je korišten pokazivač javlja jedan stepen indirekcije više. Stoga tvrdnja da su nizovi zapravo konstantni pokazivači koja se susreće u mnogim knjigama nije potpuno tačna: *nizovi nisu konstantni pokazivači* (iznimku predstavljaju formalni parametri nizovnog tipa koji, kao što smo već ranije objasnili, jesu pokazivači). Ipak, bez obzira na veliku razliku u internoj realizaciji, ova dva slučaja se zaista ponašaju veoma slično sa aspekta programera. Jedina razlika u ponašanju nizova i konstantnih pokazivača vidljiva programeru nastaje pri upotrebni operatora “**sizeof**” i “**&**”. Na primjer, operator “**sizeof**” primijenjen na *niz* vratiće kao rezultat veličinu *niza*, dok će isti operator primijenjen na *pokazivač* vratiti kao rezultat veličinu *pokazivača* (najčešće 4 bajta). Dalje, operator “**&**” primijenjen na *niz* daje kao rezultat *pokazivač na niz*, dok primijenjen na *pokazivač* daje kao rezultat *pokazivač na pokazivač* (dakle, dvostruki pokazivač). Dakle, u ova dva slučaja dobijaju se *pokazivači različitih tipova* (koji se stoga i ponašaju različito).

Neko bi mogao pomisliti da od dinamičke alokacije nizova nema osobite koristi, s obzirom da se sličan efekat (odnosno kreiranje kolekcije objekata istog tipa čiji broj nije unaprijed poznat) može postići upotrebom vektora, odnosno promjenljivih tipa “**vector**”. Međutim, ovakvo razmišljanje nije posve ispravno, s obzirom da je tip “**vector**” izvedeni tip podataka, koji je definiran u istoimenoj biblioteci, i koji je implementiran upravo pomoću dinamičke alokacije memorije i operatora “**new**”! Tačno je da je potreba za *eksplicitnom* dinamičkom alokacijom nizova smanjena nakon uvođenja tipa “**vector**” u standard jezika C++. Međutim, da ne postoji dinamička alokacija memorije i operator “**new**”, ne bi bilo moguće ni kreiranje tipa “**vector**”.

Stoga, ukoliko želimo u potpunosti da ovladamo tipom “vector” i da samostalno kreiramo *vlastite tipove podataka* koji su njemu srođni, moramo shvatiti mehanizam koji stoji u pozadini funkciranja ovog tipa, a to je upravo dinamička alokacija nizova!

Za razliku od običnih nizova koji se mogu inicijalizirati *pri deklaraciji*, i običnih dinamičkih promjenljivih koje se mogu inicijalizirati *pri stvaranju*, dinamički nizovi se ne mogu inicijalizirati u trenutku stvaranja (tj. njihov sadržaj je nakon stvaranja nepredvidljiv). Naravno, inicijalizaciju je moguće uvijek naknadno izvršiti *ručno* (npr. petljom iz prethodnog primjera). Također, nije problem napisati funkciju koja će stvoriti niz, inicijalizirati ga, i vratiti kao rezultat pokazivač na novostvoreni i inicijalizirani niz. Tada tako napisanu funkciju možemo koristiti za stvaranje nizova koji će odmah po kreiranju biti i inicijalizirani. Na primjer, razmotrimo sljedeću funkciju:

```
int *StvoriNizPopunjenoNulama(int n) {
    int *p = new int[n];
    for(int i = 0; i < n; i++) p[i] = 0;
    return p;
}
```

Pomoću ovakve funkcije možemo jednostavno kreirati dinamičke nizove koji će automatski biti inicijalizirani nulama. Na primjer:

```
int *dinamicki_niz = StvoriNizPopunjenoNulama(100);
```

Ovako napisanoj funkciji mogla bi se uputiti zamjerkao da uvijek kreira nizove istog tipa (u ovom slučaju, nizove cijelih brojeva). I ovaj nedostatak se može otkloniti upotrebom generičkih funkcija. Na primjer, moguće je napisati sljedeću generičku funkciju:

```
template <typename Tip>
Tip *StvoriNizPopunjenoNulama(int n) {
    Tip *p = new Tip[n];
    for(int i = 0; i < n; i++) p[i] = 0;
    return p;
}
```

Uz pomoć ovakve funkcije možemo stvarati inicijalizirane dinamičke nizove proizvoljnog tipa kojem se može dodijeliti nula. Na primjer,

```
double *dinamicki_niz = StvoriNizPopunjenoNulama<double>(100);
```

Primijetimo da smo prilikom poziva funkcije eksplisitno morali navesti tip elemenata niza u šiljastim zagradama “<>”. To je potrebno stoga što iz samog poziva funkcije nije moguće zaključiti šta tip “Tip” predstavlja, s obzirom da se ne pojavljuje u popisu formalnih parametara funkcije.

Mada je prethodni primjer veoma univerzalan, on se može učiniti još univerzalnijim. Naime, prethodni primjer podrazumijeva da se elementima novostvorenog niza može dodijeliti *nula*. To je tačno ukoliko su elementi niza *brojevi*. Međutim, šta ukoliko želimo da stvorimo npr. niz čiji su elementi *stringovi* (odnosno objekti tipa “string”) kojima se *ne može* dodijeliti nula? Da bi se povećala univerzalnost generičkih funkcija, uvedena je konvencija da se ime tipa može *pozvati kao funkcija bez parametara*, pri čemu je rezultat takvog poziva *podrazumijevana vrijednost* za

taj tip (ovo vrijedi samo za tipove koji posjeduju podrazumijevane vrijednosti, a većina tipova je takva). Na primjer, podrazumijevana vrijednost za sve brojčane tipove je 0, a za tip string podrazumijevana vrijednost je prazan string. Tako je vrijednost izraza “`int()`” jednaka nuli, kao i izraza “`double()`” (mada tipovi ovih izraza nisu isti: prvi je tipa “`int`” a drugi tipa “`double`”), dok je vrijednost izraza “`string()`” prazan string. Stoga ne treba da čudi da će naredba

```
cout << int();
```

ispisati nulu. Zahvaljujući ovoj, na prvi pogled čudnoj konstrukciji, moguće je napisati veoma univerzalnu generičku funkciju poput sljedeće:

```
template <typename Tip>
Tip *StvoriInicijaliziraniNiz(int n) {
    Tip *p = new Tip[n];
    for(int i = 0; i < n; i++) p[i] = Tip();
    return p;
}
```

Sa ovako napisanom funkcijom, moguće je pisati konstrukcije poput

```
double *niz_brojeva = StvoriInicijaliziraniNiz<double>(100);
string *niz_stringova = StvoriInicijaliziraniNiz<string>(150);
```

koje će stvoriti dva dinamička niza “`niz_brojeva`” i “`niz_stringova`” od 100 i 150 elemenata respektivno, čiji će elementi biti respektivno inicijalizirani nulama, odnosno praznim stringovima.

Kada nam neki dinamički niz više nije potreban, možemo ga također uništiti (tj. oslobođiti prostor koji je zauzimao) pomoću operatora “`delete`”, samo uz neznatno drugačiju sintaksu u kojoj se koristi par uglastih zagrada. Tako, ukoliko pokazivač “`pokazivac`” pokazuje na dinamički niz, uništavanje dinamičkog niza realizira se pomoću naredbe

```
delete [] pokazivac;
```

Neophodno je napomenuti da su uglaste zagrade *bitne*. Naime, postupci dinamičke alokacije običnih dinamičkih promjenljivih i dinamičkih nizova interno se obavljaju na potpuno drugačije načine, tako da ni postupak njihovog brisanja nije isti. Stoga se često govori da se kreiranje i brisanje običnih dinamičkih promjenljivih realizira uz pomoć operatora “`new`” i “`delete`”, dok se kreiranje i brisanje dinamičkih nizova realizira uz pomoć operatora “`new[]`” i “`delete[]`” (zaista, C++ interno posmatra ove varijante kao posve različite operatore). Mada postoji situacije u kojima bi se dinamički nizovi mogli obrisati primjenom običnog operatora “`delete`” (bez uglastih zagrada), takvo brisanje je uvijek veoma rizično, pogotovo ukoliko se radi o nizovima čiji su elementi složeni tipovi podataka poput struktura i klasa (na primjer, brisanje niza pomoću običnog operatora “`delete`” sigurno neće biti obavljeno kako treba ukoliko elementi niza posjeduju tzv. *destruktore*, o kojima ćemo govoriti kasnije). Stoga, ne treba mnogo filozofirati, nego se treba držati pravila: dinamički nizovi se uvijek moraju brisati pomoću konstrukcije “`delete[]`”. Ovdje treba biti posebno oprezan zbog činjenice da nas kompjajler *neće upozoriti* ne upotrijebimo li uglaste zagrade, s obzirom na činjenicu da kompjajler *ne može znati* na šta pokazuje pokazivač koji se navodi kao argument operatoru “`delete`”.

Već je rečeno da bi dinamička alokacija memorije trebala uvijek da se vrši unutar “**try**“ bloka, s obzirom da se može desiti da alokacija ne uspije. Stoga, sljedeći primjer, koji alocira dinamički niz čiju veličinu zadaje korisnik, a zatim unosi elemente niza sa tastature i ispisuje ih u obrnutom poretku, ilustrira kako se ispravno treba raditi sa dinamičkim nizovima:

```
try {
    int n;
    cout << "Koliko želite brojeva? ";
    cin >> n;
    int *niz = new int[n];
    cout << "Unesite brojeve:\n";
    for(int i = 0; i < 10; i++) cin >> niz[i];
    cout << "Niz brojeva ispisani naopako glasi:\n";
    for(int i = 9; i >= 0; i--) cout << niz[i] << endl;
    delete[] niz;
}
catch(...) {
    cout << "Nema dovoljno memorije!\n";
}
```

Obavljanje dinamičke alokacije memorije unutar “**try**” bloka je posebno važno kada se vrši dinamička alokacija *nizova*. Naime, alokacija sigurno neće uspjeti ukoliko se zatraži alokacija niza koji zauzima više prostora nego što iznosi količina slobodne memorije (npr. prethodni primjer će sigurno baciti izuzetak u slučaju da zatražite alociranje niza od recimo 100000000 elemenata)

Na ovom mjestu je neophodno napomenuti da se rezervacija memorije za čuvanje elemenata vektora također interno realizira pomoću dinamičke alokacije memorije i operadora “**new**”. To zapravo znači da prilikom deklaracije vektora također može doći do bacanja izuzetka (tipa “**bad_alloc**”) ukoliko količina raspoložive memorije nije dovoljna da se kreira vektor odgovarajućeg kapaciteta. Zbog toga bi se i deklaracije vektora načelno trebale nalaziti unutar “**try**” bloka. Stoga, ukoliko bismo u prethodnom primjeru željeli izbjegći eksplicitnu dinamičku alokaciju memorije, i umjesto nje koristiti tip “**vector**”, modificirani primjer trebao bi izgledati ovako:

```
try {
    int n;
    cout << "Koliko želite brojeva? ";
    cin >> n;
    vector<int> niz(n);
    cout << "Unesite brojeve:\n";
    for(int i = 0; i < 10; i++) cin >> niz[i];
    cout << "Niz brojeva ispisani naopako glasi:\n";
    for(int i = 9; i >= 0; i--) cout << niz[i] << endl;
}
catch(...) {
    cout << "Nema dovoljno memorije!\n";
}
```

Izuzetak tipa “**bad_alloc**” također može biti bačen kao posljedica operacija koje povećavaju

veličinu vektora (poput “`push_back`” ili “`resize`”) ukoliko se ne može udovoljiti zahtjevu za povećanje veličine (zbog nedostatka memorijskog prostora).

Možemo primijetiti jednu suštinsku razliku između primjera koji koristi operator “`new`” i primjera u kojem se koristi tip “`vector`”. Naime, u primjeru zasnovanom na tipu “`vector`” ne koristi se operator “`delete`”. Očigledno, lokalne promjenljive tipa “`vector`” se ponašaju kao i sve druge automatske promjenljive – njihov kompletan sadržaj se uništava nailaskom na kraj bloka u kojem su definirane (uključujući i oslobađanje memorije koja je bila alocirana za potrebe smještanja njihovih elemenata). U kasnijim poglavljima ćemo detaljno razmotriti na koji je način ovo postignuto.

Slično običnim dinamičkim promjenljivim, i dinamički nizovi se uništavaju tek na završetku programa, ili eksplisitnom upotrebom operatora “`delete[]`”. Stoga, pri njihovoј upotrebi također treba voditi računa da ne dođe do curenja memorije, koje može biti znatno ozbiljnije nego u slučaju običnih dinamičkih promjenljivih (s obzirom da dinamički nizovi mogu zauzimati mnogo više memorije, pogotovo ukoliko im je broj elemenata veliki). Naročito treba paziti da dinamički niz koji se alocira unutar neke funkcije preko pokazivača koji je lokalna promjenljiva obavezno treba i uništiti prije završetka funkcije, inače će taj dio memorije ostati trajno zauzet do završetka programa, i nikao ga neće moći osloboditi (izuzetak nastaje jedino u slučaju ako funkcija vraća kao rezultat pokazivač na alocirani niz – u tom slučaju onaj ko poziva funkciju ima mogućnost da oslobodi zauzetu memoriju kada ona više nije potrebna). Višestrukim pozivom takve funkcije (npr. unutar neke petlje) možemo veoma brzo nesvesno zauzeti svu raspoloživu memoriju! Dakle, svaka funkcija bi prije svog završetka morala osloboditi svu memoriju koju je dinamički zauzela (osim ukoliko vraća pokazivač na zauzeti dio memorije), i to bez obzira kako se funkcija završava: nailaskom na kraj funkcije, naredbom “`return`”, ili bacanjem izuzetka! Naročito se često zaboravlja da funkcija, prije nego što baci izuzetak, također treba da za sobom “počisti” sve što je “zabrljala”, što uključuje i oslobađanje dinamički alocirane memorije! Još je veći problem ukoliko funkcija koja dinamički alocira memoriju pozove neku drugu funkciju koja može da baci izuzetak. Posmatrajmo, na primjer, sljedeći isječak:

```
void F(int n) {
    int *p = new int[n];
    ...
    G(n);
    ...
    delete[] p;
}
```

U ovom primjeru, funkcija “`F`” zaista briše kreirani dinamički niz po svom završetku, ali problem nastaje ukoliko funkcija “`G`” koju ova funkcija poziva baci izuzetak! Kako se taj izuzetak ne hvata u funkciji “`F`”, ona će također biti prekinuta, a zauzeta memorija neće biti oslobođena. Naravno, prekid funkcije “`F`” dovodi do automatskog uništavanja automatske lokalne pokazivačke promjenljive “`p`”, ali dinamički niz na čiji početak “`p`” pokazuje nikada se ne uništava automatski, već samo eksplisitnim pozivom operatora “`delete[]`”. Stoga, ukoliko se operator “`delete[]`” ne izvrši eksplisitno, zauzeta memorija neće biti oslobođena! Ovaj problem se može riješiti na sljedeći način:

```
void F(int n) {
```

```

int *p = new int[n];
...
try {
    G(n);
}
catch(...) {
    delete[] p;
    throw;
}
...
delete[] p;
}

```

U ovom slučaju u funkciji “F“ izuzetak koji eventualno baca funkcija “G“ hvatamo samo da bismo mogli izvršiti brisanje zauzete memorije, nakon čega uhvaćeni izuzetak prosljeđujemo dalje, navođenjem naredbe “**throw**“ bez parametara.

Iz ovog primjera vidimo da je bitno razlikovati sam dinamički niz od pokazivača koji se koristi za pristup njegovim elementima. Ovdje je potrebno ponovo napomenuti da se opisani problemi *ne bi pojavili* ukoliko bismo umjesto dinamičke alokacije memorije koristili automatsku promjenljivu “p” tipa “vector” – ona bi automatski bila uništena po završetku funkcije “F”, bez obzira da li je do njenog završetka došlo na prirodan način, ili bacanjem izuzetka iz funkcije “G”. U suštini, kad god možemo koristiti tip “vector”, njegovo korištenje je jednostavnije i sigurnije od korištenja dinamičke alokacije memorije. Stoga, tip “vector” treba koristiti kad god je to moguće – njegova upotreba *sigurno* neće nikada dovesti do curenja memorije. Jedini problem je u tome što to *nije uvijek moguće*. Na primjer, nije moguće *napraviti* tip koji se ponaša slično kao tip “vector” (ili sam tip “vector”) bez upotrebe dinamičke alokacije memorije i dobrog razumijevanja kao dinamička alokacija memorije funkcionira.

Iz svega što je do sada rečeno može se zaključiti da dinamička alokacija memorije sama po sebi nije komplikirana, ali da je potrebno preduzeti dosta mjera predostrožnosti da ne dođe do curenja memorije. Dodatne probleme izaziva činjenica da operator “**new**“ može da baci izuzetak. Razmotrimo, na primjer, sljedeći isječak:

```

try { +++
    int *a = new int[n1], *b = new int[n2], *c = new int[n3];
    // Radi nešto sa a, b i c
    delete[] a; delete[] b; delete[] c;
}
catch(...) {
    cout << "Problemi sa memorijom!\n";
}

```

U ovom isječku vrši se alokacija tri dinamička niza, a u slučaju da alokacija ne uspije, prijavljuje se greška. Međutim, problemi mogu nastati u slučaju da, na primjer, alokacije prva dva niza uspiju, a alokacija trećeg niza ne uspije. Tada će treći poziv operatora “**new**“ baciti izuzetak, i izvršavanje će se nastaviti unutar “**catch**“ bloka kao što je i očekivano, ali memorija koja je zauzeta sa prve dvije uspješne alokacije ostaje zauzeta! Ovo može biti veliki problem. Jedan način da se riješi ovaj problem, mada prilično rogočatan, je da se koriste *ugniježdene* “**try**“ – “**catch**“ strukture, kao na primjer u sljedećem isječku:

```

try {
    int *a = new int[n1];

```

```

try {
    int *b = new int[n2];
    try {
        int *c = new int[n3];
        // Radi nešto sa a, b i c
        delete[] a; delete[] b; delete[] c;
    }
    catch(...) {
        delete[] b;
        throw;
    }
}
catch(...) {
    delete[] a;
    throw;
}
catch(...) {
    cout << "Problemi sa memorijom!\n";
}

```

Najbolje je da sami analizirate tok navedenog isječka uz raličite prepostavke, koje mogu biti: prva alokacija nije uspjela; druga alokacija nije uspjela; treća alokacija nije uspjela; sve alokacije su uspjele. Na taj način ćete najbolje shvatiti kako ovo rješenje radi.

Prikazano rješenje je zaista rogobatno, mada je prilično jasno i logično. Ipak, postoji i mnogo jednostavnije rješenje (ne računajući posve banalno rješenje koje se zasniva da umjesto dinamičke alokacije memorije koristimo tip “vector”, kod kojeg ne moramo eksplisitno voditi računa o brisanju zauzete memorije). Ovo rješenje zasniva se na činjenici da standard jezika C++ garantira da operatori “**delete**” odnosno “**delete**[]“ ne rade ništa ukoliko im se kao argument prosljedi *nul-pokazivač*. To omogućava sljedeće veoma elegantno rješenje navedenog problema:

```

int *a(0), *b(0), *c(0);
try {
    a = new int[n1]; b = new int[n2]; c = new int[n3];
    // Radi nešto sa a, b i c
}
catch(...) {
    cout << "Problemi sa memorijom!\n";
}
delete[] a; delete[] b; delete[] c;

```

U ovom primjeru svi pokazivači su prvo inicijalizirani na nulu, a zatim je u “**try**” bloku pokušana dinamička alokacija memorije. Na kraju se na sve pokazivače primjenjuje “**delete**[]“ operator, bez obzira da li je alokacija uspjela ili nije. Tako će oni nizovi koji su alocirani svakako biti uništeni, a ukoliko alokacija nekog od nizova nije uspjela, odgovarajući pokazivač će i dalje ostati *nul-pokazivač* (s obzirom da kasnija dodjela nije izvršena jer je operator “**new**“ bacio izuzetak), tako da operator “**delete**[]“ neće sa njim uraditi ništa, odnosno neće biti nikakvih neželjenih efekata.

Bitno je napomenuti da je ponašanje operatora “**delete**” odnosno “**delete**[]“ *nedefinirano* (i može rezultirati krahom programa) ukoliko se primijene na divlji pokazivač. Naročito česta greška je primijeniti operator “**delete**” odnosno “**delete**[]“ na pokazivač koji pokazuje na

prostor koji je već obrisan (tj. na viseći pokazivač). Ovo se, na primjer može desiti ukoliko dva puta uzastopno primijenimo ove operatore na isti pokazivač (kojem u međuvremenu između dvije primjene operatora “`delete`” ili “`delete[]`” nije dodijeljena neka druga vrijednost). Da bi se izbjegli ovi problemi, veoma dobra ideja je eksplicitno dodijeliti nul-pokazivač svakom pokazivaču nakon izvršenog uništavanja bloka memorije na koju on pokazuje. Na primjer, ukoliko “`p`” pokazuje na prvi element nekog dinamičkog niza, uništavanje tog niza najbolje je izvesti sljedećom konstrukcijom:

```
delete[] p;
p = 0;
```

Eksplicitnom dodjelom “`p = 0`” zapravo postižemo dva efekta. Prvo, takvom dodjelom eksplicitno naglašavamo da pokazivač “`p`” više ne pokazuje ni na šta. Drugo, ukoliko slučajno ponovo primijenimo operator “`delete[]`” na pokazivač “`p`”, neće se desiti ništa, jer je sada “`p`” nul-pokazivač.

Neko bi se mogao zapitati zbog čega operatori “`delete`” i “`delete[]`” ne dodjeljuju automatski nul-pokazivač svom argumentu, ukoliko je to već tako korisni i poželjno. Problem je u tome što argument ovih operatora ne mora biti isključivo *pokazivačka promjenljiva*, već može biti bilo koji *izraz pokazivačkog tipa*, koji ne mora nužno biti l-vrijednost, tako da dodjela u općem slučaju nije uvijek ni moguća. Drugim riječima, principijelno su legalni i izrazi poput “`delete (p + 1)`” i drugi kod kojih je kao operand operatora “`delete`” odnosno “`delete[]`” upotrijebljen izraz koji nije l-vrijednost. Ipak, napomenimo da ovakve konstrukcije smijemo koristiti jedino ukoliko smo apsolutno sigurni da znamo šta njima želimo postići. Početnici sigurno ne bi trebali da koriste takve konstrukcije!

U ovom poglavlju smo razmatrali samo dinamičku alokaciju *individualnih promjenljivih* i *jednodimenzionalnih nizova*. U praksi se, međutim, često javlja potreba za dinamičkom alokacijom *višedimenzionalnih nizova*. Mada dinamička alokacija višedimenzionalnih objekata nije direktno podržana u jeziku C++, ona se može vješto *simulirati*. Kako su višedimenzionalni nizovi zapravo *nizovi nizova*, to je za indirektni pristup njihovim elementima (pa samim tim i za njihovu dinamičku alokaciju) potrebno koristiti *složenje pokazivačke tipove*, kao što su *pokazivači na nizove*, *nizovi pokazivača* i *pokazivači na pokazivače*, zavisno od primjene. Ovakvim pokazivačkim tipovima, kao i dinamičkoj alokaciji višedimenzionalnih objekata, posvećeno je sljedeće poglavlje.

27. Složeni pokazivački tipovi

Već smo napomenuli da ne postoje samo pokazivači na jednostavne objekte, kao što su cijeli i realni brojevi ili znakovi, nego da je moguće napraviti pokazivače na *bilo koje objekte*. Tako npr. postoje *pokazivači na nizove*, *pokazivači na pokazivače* (odnosno *dvojni pokazivače*), *pokazivači na funkcije*, itd. Također, elementi nizova mogu sasvim legalno biti pokazivači, tako da možemo govoriti o *nizovima pokazivača*. Ovakve složene tipove najlakše je napraviti pomoću “**typedef**“ deklaracija. Na primjer, neka su date sljedeće deklaracije:

```
typedef int *PokNaCijeli;
typedef int NizCijelih[10];
PokNaCijeli p1[30];
NizCijelih *p2;
PokNaCijeli *p3;
```

U ovom primjeru, “p1“ je *niz od 30 pokazivača na cijele brojeve*, “p2“ je pokazivač na tip “NizCijelih”, odnosno *pokazivač na niz od 10 cijelih brojeva* (odnosno na *čitav niz kao cjelinu* a ne *pokazivač na prvi njegov element* – uskoro ćemo vidjeti u čemu je razlika), dok je “p3“ pokazivač na tip “PokNaCijeli”, odnosno *pokazivač na pokazivač na cijele brojeve*. Ovakve konstrukcije moguće je deklarirati i neposredno, bez “**typedef**“ deklaracija (mada “**typedef**“ deklaracije čine cijelu stvar mnogo jasnijom). Na primjer, iste deklaracije mogle su se napisati i ovako:

```
int *p1[30];
int (*p2)[10];
int **p3;
```

Obratimo pažnju između razlike u deklaraciji “p1“ koja predstavlja *niz pokazivača*, i deklaracije “p2“ koja predstavlja *pokazivač na niz*. Na izvjestan način, pojam “niz” prilikom deklaracija ima prioritet u odnosu na pojam “pokazivač”, pa su u drugom primjeru zgrade bile neophodne da “izvrnu” ovaj prioritet. Vrijedi još napomenuti da se ovakve konstrukcije mogu usložnjavati unedogled (npr. principijelno je moguće napraviti *pokazivač na niz od 10 pokazivača na pokazivače na niz od 50 realnih brojeva*), mada se potreba za ovako složenim konstrukcijama javlja zaista iznimno rijetko, i samo u veoma specifičnim primjenama.

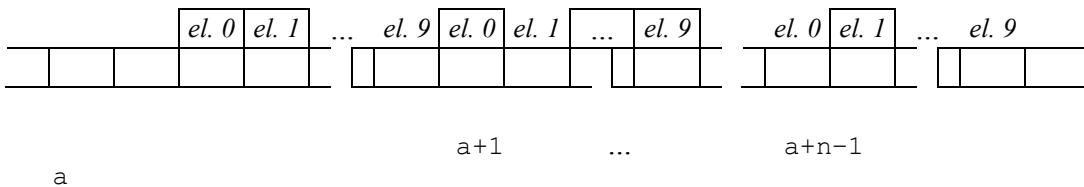
Mada konstrukcije poput gore prikazanih izgledaju veoma egzotično, potreba za njima se javlja u izvjesnim situacijama, na primjer prilikom dinamičke alokacije višedimenzionalnih nizova. Razmotrimo, na primjer, kako bi se mogla realizirati dinamička alokacija *dvodimenzionalnih nizova*. Sjetimo se da su dvodimenzionalni nizovi zapravo *nizovi čiji su elementi nizovi*, a da se dinamička alokacija nizova čiji su elementi tipa “Tip” ostvaruje posredstvom pokazivača na tip “Tip”. Slijedi da su nam za dinamičku alokaciju nizova čiji su elementi nizovi potrebni *pokazivači na nizove*. I zaista, koristeći opisano rezonovanje, sasvim je lako ostvariti dinamičku alokaciju dvodimenzionalnog niza kod kojeg je *druga dimenzija unaprijed poznata*, npr. matrice čiji je broj kolona unaprijed poznat (razlog za neophodnost apriornog poznавanja druge dimenzije biće uskoro razjašnjen). Prepostavimo npr. da druga dimenzija niza kojeg hoćemo da alociramo iznosi 10, a da je prva dimenzija zadana u promjenljivoj “n“ (čija vrijednost nije unaprijed poznata). Tada ovu alokaciju možemo ostvariti pomoću sljedeće konstrukcije:

```
typedef int Niz[10];
```

```
Niz *a = new Niz[n];
```

Prvom naredbom definirali smo novi tip “`Niz`“ koji predstavlja niz od najviše 10 cijelih brojeva. U drugoj naredbi smo pomoću operatora “`new`“ alocirali niz od “`n`“ elemenata tipa “`Niz`“ (što nije ništa drugo nego dvodimenzionalni niz sa najviše “`n`“ redova i 10 kolona), i usmjerili pokazivač “`a`“ da pokazuje na prvi element takvog niza. Ova dodjela je legalna, jer se pokazivaču na neki tip uvijek može dodijeliti adresa dinamičkog niza tog tipa. Primijetimo da je u ovom slučaju pokazivač “`a`“ *pokazivač na (čitav) niz*. Ovo treba razlikovati od običnih pokazivača, za koje znamo da se po potrebi mogu smatrati kao *pokazivači na elemente niza*. U brojnoj literaturi se ova dva pojma često brkaju, tako da se, kada se kaže *pokazivač na niz*, obično misli na *pokazivač na prvi element niza*, a ne na *pokazivač na niz kao cjelinu* (inače, ova dva tipa pokazivača razlikuju se u tome kako na njih djeluje pokazivačka aritmetika). Pokazivači na (čitave) nizove koriste se prilično rijetko, i glavna im je primjena upravo za dinamičku alokaciju dvodimenzionalnih nizova čija je druga dimenzija poznata.

U gore navedenom primjeru možemo reći da pokazivač “`a`“ pokazuje na prvi element *dinamičkog niza* od “`n`“ elemenata čiji su elementi *obični (statički) nizovi* od najviše 10 cjelobrojnih elemenata. Bez obzira na činjenicu da je “`a`“ pokazivač, sa njim možemo baratati na iste način kao da se radi o običnom dvodimenzionalnom nizu. Tako je indeksiranje poput “`a[i][j]`“ posve legalno, i interpretira se kao “`(*(a + i))[j]`“. Naime, “`a`“ je pokazivač, pa se izraz “`a[i]`“ interpretira kao “`* (a + i)`“. Međutim, kako je “`a`“ pokazivač na tip “`Niz`”, to nakon njegovog dereferenciranja dobijamo element tipa “`Niz`“ na koji se može primijeniti indeksiranje. Interesantno je kako na pokazivače na (čitave) nizove djeluje pokazivačka aritmetika. Ukoliko “`a`“ pokazuje na prvi element niza, razumije se da “`a + 1`“ pokazuje na *sljedeći element*. Međutim, kako su element ovog niza sami za sebe nizovi (tipa “`Niz`”), adresa na koju “`a`“ pokazuje uvećava se za *čitavu dužinu nizovnog tipa* “`Niz`“. U ovome je osnovna razlika između pokazivača na (čitave) nizove i pokazivača na elemente niza. Sljedeća slika može pomoći da se shvati kako izgleda stanje memorije nakon ovakve alokacije, i šta na šta pokazuje:



Jasno je zbog čega je ovakva dinamička alokacija dvodimenzionalnih nizova moguća samo u slučaju kada je druga dimenzija poznata. Naime, dimenzija nizovnog tipa koji se kreira pomoću “`typedef`“ deklaracije (u našem primjeru tipa “`Niz`”) mora biti apriori poznata. Ipak, interesantno je napomenuti da za dinamičku alokaciju dvodimenzionalnog niza sa poznatom drugom dimenzijom nije neophodno uvoditi novi nizovni tip pomoću “`typedef`“ naredbe. Naime, ista alokacija se mogla obaviti i samo pomoću jedne naredbe na sljedeći način:

```
int (*a)[10] = new int[n][10];
```

Deklaracijom “`int (*a)[10]`“ neposredno deklariramo “`a`“ kao *pokazivač na niz od 10 cijelih brojeva*, dok konstrukciju “`new int[n][10]`“ možemo čitati kao “potraži u memoriji prostor za `n` elemenata koji su nizovi od najviše 10 cijelih brojeva, i vrati kao rezultat adresu pronađenog

prostora (u formi odgovarajućeg pokazivačkog tipa)”. Ovdje broj “10” u drugoj uglastoj zagradi iza operatora “`new`” ne predstavlja parametar operatora, već *sastavni dio tipa*, tako da on mora biti prava konstanta, a ne promjenljiva ili nekonstantni izraz. Drugim riječima, operator “`new[]`” ima samo jedan parametar, mada on dopušta i drugi par uglastih zagrada, samo što se u drugom paru uglastih zagrada *obavezno mora nalaziti prava konstanta* (s obzirom da one čine sastavni dio tipa). Zbog toga nije moguće neposredno primjenom operatora “`new[]`” alocirati dvodimenzionalne dinamičke nizove čija druga dimenzija *nije unaprijed poznata*.

Gore pokazana primjena pokazivača na nizove kao cjeline uglavnom je i jedina primjena takvih pokazivača. Zapravo, indirektno postoji i još jedna primjena – imena dvodimenzionalnih nizova upotrijebljena sama za sebe automatski se konvertiraju u *pokazivače na (čitave) nizove* s obzirom da su dvodimenzionalni nizovi u suštini nizovi nizova. Također, formalni parametri funkcija koji su deklarirani kao dvodimenzionalni nizovi zapravo su pokazivači na (čitave) nizove (u skladu sa općim tretmanom formalnih parametara nizovnog tipa). Ovo ujedno dodatno pojašnjava zbog čega druga dimenzija u takvim formalnim parametrima mora biti apriori poznata. Interesantno je primijetiti da se pravilo o automatskoj konverziji nizova u pokazivače ne primjenjuje rekurzivno, tako da se imena dvodimenzionalnih nizova upotrijebljena sama za sebe konvertiraju u *pokazivače na nizove*, a ne u *pokazivače na pokazivače*, kako bi neko mogao pomisliti (u pokazivače na pokazivače se konvertiraju imena *nizova pokazivača* upotrijebljena sama za sebe). Također treba primijetiti da pokazivači na nizove i pokazivači na pokazivače nisu jedno te isto (razlika je opet u djelovanju pokazivačke aritmetike).

Razmotrimo sada kako bismo mogli dinamičku alokaciju dvodimenzionalnih nizova čija druga dimenzija nije poznata unaprijed (strog rečeno, takva dinamička alokacija *nije moguća*, ali se može uspješno *simulirati*). Za tu svrhu su nam potrebni *nizovi pokazivača*. Sami po sebi, nizovi pokazivača nisu ništa osobito: to su nizovi *čiji su elementi pokazivači*. Na primjer, sljedeća deklaracija

```
int *niz_pok[5];
```

deklarira niz “`niz_pok`” od 5 elemenata koji su pokazivači na cijele brojeve. Stoga, ukoliko su npr. “`br_1`”, “`br_2`” i “`br_3`” cjelobrojne promjenljive, sve dolje navedene konstrukcije imaju smisla:

```
niz_pok[0] = &br_1;           // "niz_pok[0]" pokazuje na "br_1"
niz_pok[1] = &br_2;           // "niz_pok[1]" pokazuje na "br_2"
niz_pok[2] = new int(3);      // Alocira dinamičku promjenljivu
niz_pok[3] = new int[10];     // Alocira dinamički niz
niz_pok[4] = new int[br_1];   // Također...
*niz_pok[0] = 1;             // Djeluje poput "br_1 = 1";
br_3 = *niz_pok[1];          // Djeluje poput "br_3 = br_2"
*niz_pok[br_1] = 5;          // Indirektno mijenja promjenljivu "br_2"
cout << niz_pok[2];         // Ovo ispisuje adresu...
cout << *niz_pok[2];
delete niz_pok[2];          // Briše dinamičku promjenljivu
delete[] niz_pok[4];        // Briše dinamički niz
niz_pok[3][5] = 100;         // Vrlo interesantno...
```

Posljednja naredba je posebno interesantna, jer pokazuje da se nizovima pokazivača može pristupati kao da se radi o *dvodimenzionalnim nizovima* (pri čemu je takav pristup smislen samo

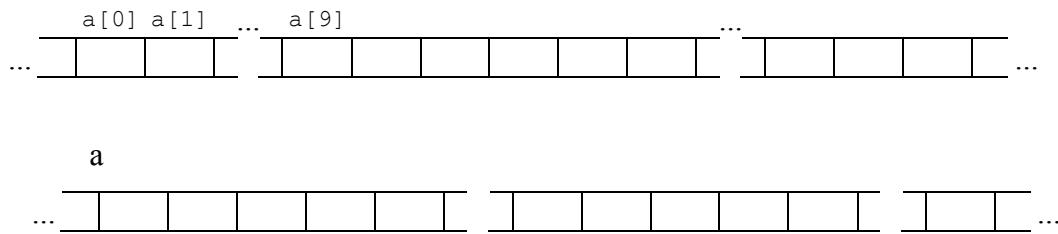
ukoliko odgovarajući element niza pokazivača pokazuje na element nekog drugog niza, koji može biti i dinamički kreiran). Zaista, svaki element niza pokazivača je pokazivač, a na pokazivače se može primijeniti indeksiranje. Stoga se izraz “`niz_pok[i][j]`” u suštini interpretira kao “`* (niz_pok[i] + j)`”. Interesantno je napomenuti da se nizovi pokazivača također mogu inicijalizirati prilikom deklaracije, pod uvjetom da *svi izrazi u inicijalizacionoj listi predstavljaju pokazivače odgovarajućeg tipa* (ili nulu, koja predstavlja nul-pokazivač). Tako smo prvih pet dodjela iz prethodnog primjera mogli izvršiti i prilikom inicijalizacije na sljedeći način:

```
int *niz_pok[5] = {&br_1, &br_2, new int, new int[10], new int[br_1]};
```

Navedeni primjer također ukazuje kako je moguće realizirati dinamički dvodimenzionalni niz čija je prva dimenzija poznata, ali čija *druga dimenzija nije poznata* (npr. matrice čiji je broj redova poznat, ali broj kolona nije). Kao što je već rečeno, jezik C++ ne podržava neposredno mogućnost kreiranja takvih nizova, ali se oni *veoma lako mogu efektivno simulirati*. Naime, dovoljno je formirati niz pokazivača, a zatim svakom od pokazivača dodijeliti *adresu dinamički alociranih nizova koji predstavljaju redove matrice*. Na primjer, neka je potrebno realizirati dinamičku matricu sa 10 redova i “`m`” kolona, gdje je “`m`” promjenljiva. To možemo uraditi na sljedeći način:

```
int *a[10];
for(int i = 0; i < 10; i++) a[i] = new int[m];
```

U ovom slučaju elementi niza pokazivača “`a`” pokazuju na prve elemente svakog od redova matrice. Strogo rečeno, “`a`” uopće nije dvodimenzionalni niz, nego niz pokazivača koji pokazuju na početke neovisno alociranih dinamičkih nizova, od kojih svaki predstavlja po jedan red matrice, i od kojih svaki može biti lociran u potpuno različitim dijelovima memorije! Međutim, sve dok “elementima” ovakvog “dvodimenzionalnog niza” možemo pristupati pomoću izraza “`a[i][j]`” (a možemo, zahvaljujući pokazivačkoj aritmetici) ne trebamo se mnogo brinuti o stvarnoj organizaciji u memoriji. Zaista, “`a`” možemo koristiti kao dvodimenzionalni niz sa 10 redova i “`m`” kolona, iako se radi samo o veoma vještoj simulaciji. Sa ovako simuliranim dvodimenzionalnim nizovima možemo raditi na gotovo isti način kao i sa pravim dvodimenzionalnim nizovima. Jedine probleme mogli bi eventualno izazvati operatori “`sizeof`” i “`&`”, kao i “`prljavi`” trikovi sa pokazivačkom aritmetikom koji bi se oslanjali na pretpostavku da su redovi dvodimenzionalnih nizova smješteni u memoriji striktno jedan za drugim (što ovdje nije slučaj). Pravo stanje u memoriji kod ovako simuliranih dvodimenzionalnih nizova moglo bi se opisati recimo sljedećom slikom:



Pažljivijem čitatelju ili čitateljki sigurno neće promaći sličnost između opisane tehnike za dinamičku alokaciju dvodimenzionalnih nizova i kreiranja nizova čiji su elementi vektori. Kasnije ćemo vidjeti da u oba slučaja u pozadini zapravo stoji isti mehanizam.

Kada smo govorili o nizovima čiji su elementi vektori, govorili smo o mogućnosti kreiranja struktura podataka nazvanih *grbavi nizovi* (engl. *ragged arrays*), koji predstavljaju strukture koje po načinu upotrebe liče na dvodimenzionalne nizove, ali kod kojih različiti “redovi” imaju različit broj elemenata. Primijetimo da nam upravo opisana simulacija dvodimenzionalnih nizova preko nizova pokazivača također omogućava veoma jednostavno kreiranje grbavih nizova. Na primjer, ukoliko izvršimo sekvencu naredbi

```
int *a[10];
for(int i = 0; i < 10; i++) a[i] = new int[i + 1];
```

tada će se niz pokazivača “a” sa aspekta upotrebe ponašati kao dvodimenzionalni niz u kojem prvi red ima jedan element, drugi element dva reda, treći element tri reda, itd. Ovakvim grbavim nizovima može se ostvariti ušteda memorije ukoliko je potrebno zapamtiti elemente neke trougaone ili simetrične matrice. Naročito su korisni grbavi nizovi znakova, o kojima ćemo govoriti nešto kasnije.

Važno je primijetiti da je u svim prethodnim primjerima zasnovanim na nizovima pokazivača, “dvodimenzionalni niz” zapravo sastavljen od gomile posve neovisnih dinamičkih jednodimenzionalnih nizova, koje međusobno objedinjuje niz pokazivača koji pokazuju na početke svakog od njih. Stoga, ukoliko želimo uništiti ovakve “dvodimenzionalne nizove” (tj. oslobođiti memorijski prostor koji oni zauzimaju), moramo posebno uništiti svaki od jednodimenzionalnih nizova koji ih tvore. To možemo učiniti npr. na sljedeći način:

```
for(int i = 0; i < 10; i++) delete[] a[i];
```

Na kraju, razmotrimo kako simulirati dvodimenzionalni niz u kojem niti jedna dimenzija nije unaprijed poznata. Rješenje se samo nameće: potrebno je i *niz pokazivača* (koji pokazuju na početke svakog od redova) *realizirati kao dinamički niz*. Pretpostavimo, na primjer, da želimo simulirati dvodimenzionalni niz sa “n” redova i “m” kolona, pri čemu niti “m” niti “n” nemaju unaprijed poznate vrijednosti. Najlakše to možemo uraditi ukoliko uvedemo posebni pokazivački tip naredbom “**typedef**”, koji će nam olakšati dinamičko kreiranje niza pokazivača:

```
typedef int *PokNaCijeli;
PokInt *a = new PokNaCijeli[n];
for(int i = 0; i < n; i++) a[i] = new int[m];
```

Nakon ovoga, “a” možemo koristiti kako da se radi o dvodimenzionalnom nizu i pisati “a[i][j]”, mada je “a” zapravo *pokazivač na pokazivač (dvojni pokazivač)*! Zaista, kako je “a” pokazivač, na njega se može primijeniti indeksiranje, tako da se izraz “a[i]” interpretira kao “*(a + i)”. Međutim, nakon dereferenciranja dobijamo ponovo pokazivač (jer je “a” pokazivač *na pokazivač*), na koji se ponovo može primijeniti indeksiranje, tako da se na kraju izraz “a[i][j]” zapravo interpretira kao “*(*(a + i) + j)”. Bez obzira na interpretaciju, za nas je važna činjenica da se “a” gotovo svuda može koristiti na način kako smo navikli raditi sa običnim dvodimenzionalnim nizovima. Također, naredbu “**typedef**“ bismo mogli izbjegići ukoliko “a” odmah deklariramo kao dvojni pokazivač (obratite pažnju na zvjezdicu koja govori da alociramo niz pokazivača na cijele brojeve a ne niz cijelih brojeva):

```
int **a = new int*[n];
```

```
for(int i = 0; i < n; i++) a[i] = new int[m];
```

Naravno, u oba slučaja ovakav “dvodimenzionalni niz” morali bismo brisati postupno (prvo sve redove, a zatim niz pokazivača koji ukazuju na početke redova):

```
for(int i = 0; i < n; i++) delete[] a[i];
delete[] a;
```

Interesantno je uporediti obične (statičke) dvodimenzionalne nizove, dinamičke dvodimenzionalne nizove sa poznatom drugom dimenzijom, simulirane dinamičke dvodimenzionalne nizove sa poznatom prvom dimenzijom, i simulirane dinamičke dvodimenzionalne nizove sa obje dimenzije nepoznate. U sva četiri slučaja za pristup individualnim elementima niza možemo koristiti sintaksu “`a[i][j]`”, s tim što se ona u drugom slučaju logički interpretira kao “`(* (a + i)) [j]`”, u trećem slučaju kao “`* (a[i] + j)`”, a u četvrtom slučaju kao “`* (* (a + i) + j)`”. Međutim, uzmemli u obzir da se ime niza upotrijebljeno bez indeksiranja automatski konvertira u pokazivač na prvi element tog niza, kao i činjenicu da su istinski elementi dvodimenzionalnih nizova zapravo jednodimenzionalni nizovi, doći ćemo do veoma interesantnog zaključka da su u sva četiri slučaja sve četiri sintakse (“`a[i][j]`”, “`(* (a + i)) [j]`”, “`* (a[i] + j)`” i “`* (* (a + i) + j)`”) u potpunosti ekvivalentne, i svaka od njih se može koristiti u sva četiri slučaja! Naravno da se u praksi gotovo uvijek koristi sintaksa “`a[i][j]`”, ali je za razumijevanje suštine korisno znati najprirodniju interpretaciju za svaki od navedenih slučajeva.

Pokazivači na pokazivače početnicima djeluju komplikirano zbog dvostrukе indirekcije, ali oni nisu ništa komplikirani od klasičnih pokazivača, ako se ispravno shvati njihova suština. U jeziku C++ pokazivači na pokazivače pretežno se koriste za potrebe dinamičke alokacije dvodimenzionalnih nizova, dok su se u jeziku C dvojni pokazivači intenzivno koristili u situacijama u kojima je u jeziku C++ prirodnije upotrijebiti *reference na pokazivač* (tj. reference vezane za neki pokazivač). Na primjer, ukoliko je “`PokNaCijeli`” tip pokazivača na cijeli broj, a “`pok`” pokazivačka promjenljiva tipa “`PokNaCijeli`”, tada deklaracijom

```
PokNaCijeli &ref = pok;
```

deklariramo referencu “`ref`” vezanu na pokazivačku promjenljivu “`pok`” (tako da je “`ref`” zapravo referenca na pokazivač). Sada se “`ref`” ponaša kao *alternativno ime* za pokazivačku promjenljivu “`pok`”. Referencu na pokazivač možemo deklarirati i bez uvođenja pomoćnog tipa “`PokNaCijeli`” konstrukcijom poput

```
int *&ref = pok;
```

Obratite pažnju na redoslijed znakova “*” i “&”. Ukoliko bismo zamijenili redoslijed ovih znakova, umjesto reference na pokazivač pokušali bismo deklarirati *pokazivač na referencu*, što nije dozvoljeno u jeziku C++ (postojanje pokazivača na referencu omogućilo bi pristup internoj strukturi reference, a dobro nam je poznato da tvorci jezika C++ nisu željeli da ni na kakav način podrže takvu mogućnost). Inače, vjerovatno ste još pri prikazu deklaracije konstantnih pokazivača primijetili da su deklaracije pokazivačkih tipova mnogo jasnije ako se čitaju *zdesna nalijevo* (tako da uz takvo čitanje, iz prethodne deklaracije jasno vidimo da “`ref`” predstavlja *referencu na pokazivač na cijele brojeve*).

Reference na pokazivače najčešće se koriste ukoliko je potrebno neki pokazivač prenijeti po

referenci kao parametar u funkciju, što je potrebno npr. ukoliko funkcija treba da izmjeni sadržaj samog pokazivača (a ne objekta na koji pokazivač pokazuje). Na primjer, sljedeća funkcija vrši razmjenu dva pokazivača koji su joj proslijedeni kao stvarni parametri (zanemarimo ovom prilikom činjenicu da će generička funkcija “Razmijeniti” koju smo napisali u poglavljju o generičkim funkcijama sasvim lijepo razmijeniti i dva pokazivača):

```
void RazmijeniPokazivace(double *&p, double *&q) {
    double *pomocna = p;
    p = q; q = pomocna;
}
```

Kako u jeziku C nisu postojale reference, sličan efekat se mogao postići jedino upotrebom *dvojnih pokazivača*, kao u sljedećoj funkciji

```
void RazmijeniPokazivace(double **p, double **q) {
    double *pomocna = *p;
    *p = *q; *q = pomocna;
}
```

Naravno, prilikom poziva ovakve funkcije “RazmijeniPokazivace”, kao stvarne argumente morali bismo navesti *adrese* pokazivača koje želimo razmijeniti (a ne same pokazivače). Drugim riječima, za razmjenu dva pokazivača “*p1*” i “*p2*” (na tip “**double**”) morali bismo izvršiti sljedeći poziv:

```
RazmijeniPokazivace(&p1, &p2);
```

Očigledno, upotreba referenci na pokazivače (kao uostalom i upotreba bilo kakvih referenci) oslobađa korisnika funkcije potrebe da eksplicitno razmišlja o adresama.

Svi do sada prikazani postupci dinamičke alokacije matrica nisu vodili računa o tome da li su alokacije zaista uspjele ili nisu. U realnim situacijama moramo i o tome voditi računa, tako da bismo dinamičku alokaciju matrice realnih brojeva sa “*n*“ redova i “*m*“ kolona zapravo trebali realizirati ovako (razmislite sami zbog čega se prvo svi pokazivači inicijaliziraju na nule):

```
try {
    double **a = new int*[n];
    for(int i = 0; i < n; i++) a[i] = 0;
    try {
        for(int i = 0; i < n; i++) a[i] = new int[m];
    }
    catch(...) {
        for(int i = 0; i < n; i++) delete[] a[i];
        delete[] a;
        throw;
    }
}
```

```

}
catch(...) {
    cout << "Problemi sa memorijom!\n";
}

```

Vidimo da dinamička alokacija matrica proizvoljnih dimenzija može biti mukotrpna ukoliko vodimo računa o mogućim memoriskim problemima. Naročito je mukotrpno identičan postupak ponavljati za svaku od matrica koju želimo da kreiramo. Zbog toga se kao prirodno rješenje nameće *pisanje funkcija koje obavlaju dinamičko stvaranje odnosno uništavanje matrica proizvoljnih dimenzija*, koje na sebe preuzimaju opisani postupak. Sljedeći primjer pokazuje kako bi takve funkcije mogle izgledati (funkcije su napisane kao generičke funkcije, tako da omogućavaju stvaranje odnosno uništavanje matrica čiji su elementi proizvoljnog tipa):

```

template <typename Tip>
void UnistiMatricu(Tip **mat, int broj_redova) {
    if(mat == 0) return;
    for(int i = 0; i < broj_redova; i++) delete[] mat[i];
    delete[] mat;
}

template <typename Tip>
Tip **StvoriMatricu(int broj_redova, int broj_kolona) {
    double **mat = new Tip*[broj_redova];
    for(int i = 0; i < broj_redova; i++) mat[i] = 0;
    try {
        for(int i = 0; i < broj_redova; i++) mat[i] = new Tip[broj_kolona];
    }
    catch(...) {
        UnistiMatricu(mat, broj_redova);
        throw;
    }
    return mat;
}

```

Obratimo pažnju na nekoliko detalja u ovim funkcijama. Prvo, funkcija “UnistiMatricu” je napisana ispred funkcije “StvoriMatricu”, s obzirom da se ona poziva iz funkcije “StvoriMatricu”. Također, funkcija “UnistiMatricu” napisana je tako da ne radi ništa u slučaju da joj se kao parametar prenese nul-pokazivač, čime je ostvarena konzistencija sa načinom na koji se ponašaju operatori “**delete**” i “**delete**[]”. Konačno, funkcija “UnistiMatricu” vodi računa da iza sebe “počisti” sve dinamički alocirane nizove prije nego što eventualno baci izuzetak u slučaju da dinamička alokacija nije uspjela do kraja (ovo “čišćenje” je realizirano pozivom funkcije “UnistiMatricu”).

Sljedeći primjer ilustrira kako možemo koristiti napisane funkcije za dinamičko kreiranje i uništavanje matrica:

```

int n, m;
cin >> n, m;
try {
    double **a = 0, **b = 0;
    a = StvoriMatricu<double>(n, m);
    b = StvoriMatricu<double>(n, m);
}

```

```

    // Radi nešto sa matricama a i b
}
catch(...) {
    cout << "Problemi sa memorijom!\n";
}
UnistiMatricu(a, n);
UnistiMatricu(b, n);

```

Primijetimo da smo dvojne pokazivače “*a*” i “*b*” koje koristimo za pristup dinamički alociranim matricama prvo inicijalizirali na nulu, prije nego što zaista pokušamo dinamičku alokaciju pozivom funkcije “*StvoriMatricu*”. Na taj način garantiramo da će u slučaju da alokacija ne uspije, odgovarajući pokazivač biti nul-pokazivač, tako da kasniji poziv funkcije “*UnistiMatricu*” neće dovesti do problema u slučaju da kreiranje matrice uopće nije izvršeno (ovdje imamo situaciju identičnu situaciji o kojoj smo već govorili prilikom razmatranja dinamičke alokacije običnih nizova). Generalno, kad god dinamički alociramo *više od jednog objekta*, trebamo izbjegavati obavljanje dinamičke alokacije odmah prilikom inicijalizacije pokazivača, jer u suprotnom prilikom brisanja objekata možemo imati problema u slučaju da nije uspjela alokacija svih objekata (naime, ne možemo znati koji su objekti alocirani, a koji nisu). Primijetimo još da u funkciju “*UnistiMatricu*“ moramo kao parametar prenosi i broj redova matrice, s obzirom da nam je on potreban u “**for**“ petlji, a njega nije moguće saznati iz samog pokazivača.

Prilikom poziva generičke funkcije “*StvoriMatricu*” morali smo unutar šiljastih zagrada “*<>*” eksplisitno specificirati tip elemenata matrice, s obzirom da se tip elemenata ne može odrediti dedukcijom tipa iz parametara funkcije. Ovaj problem možemo izbjegići ukoliko modifciramo funkciju “*StvoriMatricu*” tako da kao jedan od parametara prihvata dvojni pokazivač za pristup elementima matrice. Tako modifirana funkcija treba da u odgovarajući parametar *smjesti* adresu dinamički alocirane matrice (umjesto da tu adresu *vratи kao rezultat*). Naravno, u tom slučaju ćemo za dinamičko alociranje matrice umjesto poziva poput

```
a = StvoriMatricu<double>(n, m);
```

koristiti poziv poput

```
StvoriMatricu(a, n, m);
```

Naravno, prvi parametar se mora prenosi po referenci da bi uopće mogao biti promijenjen, što znači da odgovarajući formalni parametar mora biti *referenca na dvojni pokazivač* (ne plašite se što ovdje imamo *trostruku indirekciju*). Ovakvu modifikaciju ostavljamo čitatelju odnosno čitateljki kao vježbu. Treba li uopće napominjati da se u jeziku C (koji ne posjeduje reference) sličan efekat može ostvariti jedino upotrebom *trostrukih pokazivača* (odnosno *pokazivača na pokazivače na pokazivače*)?

Dinamički kreirane matrice (npr. matrice kreirane pozivom funkcije “*StvoriMatricu*”) u većini konteksta možemo koristiti kao i obične dvodimenzionalne nizove. Moguće ih je i

prenositi u funkcije, samo što odgovarajući formalni parametar koji služi za pristup dinamičkoj matrici mora biti definiran kao dvojni pokazivač (kao u funkciji “`UnistiMatricu`”). Ipak, razlike između dinamički kreiranih matrica kod kojih nijedna dimenzija nije poznata unaprijed i običnih dvodimenzionalnih nizova izraženije su u odnosu na razlike između običnih i dinamičkih jednodimenzionalnih nizova. Ovo odražava činjenicu da u jeziku C++ zapravo ne postoje pravi dvodimenzionalni dinamički nizovi kod kojih druga dimenzija nije poznata unaprijed. Oni se mogu veoma vjerno simulirati, ali ne u potpunosti savršeno. Tako, na primjer, nije moguće napraviti jedinstvenu funkciju koja bi prihvatala kako statičke tako i dinamičke matrice, jer se ime dvodimenzionalnog niza upotrijebljeno samo za sebe ne konvertira u dvojni pokazivač, nego u pokazivač na niz, o čemu smo već govorili (naravno, u slučaju potrebe, moguće je napraviti dvije funkcije sa *identičnim tijelima a različitim zaglavljima*, od kojih jedna prima statičke a druga dinamičke matrice).

Pedantne čitatelje i čitateljke treba upozoriti na još jedan detalj. Poznato je da je veoma dobro sve formalne parametre nizovnog ili pokazivačkog tipa koji služe za pristup elementima niza koji se *neće mijenjati* unutar funkcije označiti kvalifikatorom “`const`”. Stoga bi pedantan programer ili programerka zaglavljje funkcije koja ispisuje elemente dinamičke matrice vjerovatno pokušali napisati ovako:

```
void IspisiMatricu(const double **mat, int m, int n)
```

Međutim, ukoliko bismo pokušali pozvati ovaku funkciju prenoseći joj neku dinamičku matricu kao stvarni parametar, primjetili bismo da će kompjuter prijaviti grešku na mjestu poziva. Razlog zbog kojeg nastaje ovaj problem mnogima nije poznat, i stoga ga obično rješavaju tako što prosto uklone kvalifikator “`const`”. Ipak, takvo rješenje je “linija manjeg otpora”. Bolje je shvatiti zbog čega problem nastaje, i tretirati ga na ispravan način. Problem je u tome što je, u gornjoj deklaraciji, “`mat`” tipa *pokazivač na (nekonstantni) pokazivač na konstantni realni broj*. Iz izvjesnih razloga, koji će uskoro biti objašnjeni, jezik C++ ne dozvoljava da se ovakvom pokazivaču dodijeli *obični dvojni pokazivač* (odnosno *pokazivač na (nekonstantni) pokazivač na (nekonstantni) realni broj*). Problem se rješava tako što se formalni parametar deklarira kao *pokazivač na konstantni pokazivač na konstantni realni broj*, kao u sljedećem zaglavljtu:

```
void IspisiMatricu(const double *const *mat, int m, int n)
```

Onome koji samo želi da koristi kvalifikator “`const`” gdje god je to moguće, ovo objašnjenje je sasvim dovoljno. Međutim, za one čitatelje i čitateljke koji neće moći mirno spavati dok ne saznaju zbog čega pomenuta dodjela nije dozvoljena, slijedi i objašnjenje. Razlog nije posve jednostavan, tako da oni koji ne žele da se udubljuju u detalje, mogu slobodno preskočiti objašnjenje koje slijedi (iako je njegovo razumijevanje dobra vježba). Razlog je zapravo sljedeći: pretpostavimo da je takva dodjela moguća, i razmotrimo sljedeću sekvencu naredbi:

```
const int x = 1;
```

```

int *p1;
int **p2 = &p1;
const int ***p3 = p2;
*p3 = &x;
*p1 = 2;

```

Ako dobro analizirate gornju sekvencu naredbi, shvatićete da bi ona zapravo trebala promijeniti vrijednost konstante “x” sa 1 na 2! Ipak, ovo nije moguće, s obzirom da dodjela izvršena u četvrtoj naredbi u prethodnoj sekvenci *nije legalna*.

Već smo rekli da nizovi pokazivača glavnu primjenu imaju prilikom simuliranja dinamičke alokacije dvodimenzionalnih nizova. Međutim, nizovi pokazivača mogu imati i druge primjene. Naročito su korisni *nizovi pokazivača na znakove*. Zamislimo, na primjer, da je neophodno da u nekom nizu čuvamo imena svakog od 12 mjeseci u godini. Ukoliko za tu svrhu koristimo običan niz nul-terminaliranih stringova odnosno dvodimenzionalni niz znakova, koristili bismo sljedeću deklaraciju:

```

char imena_mjeseci[12][10] = {"Januar", "Februar", "Mart", "April",
    "Maj", "Juni", "Juli", "August", "Septembar", "Oktobar", "Novembar",
    "Decembar"};

```

U ovom slučaju smo za svaki naziv morali zauzeti po 10 znakova, koliko iznosi najduži naziv (“Septembar”), uključujući i prostor za “NUL” graničnik. Kao što smo već vidjeli, mnogo je bolje kreirati *niz dinamičkih stringova*, kao u sljedećoj deklaraciji:

```

string imena_mjeseci[12] = {"Januar", "Februar", "Mart", "April",
    "Maj", "Juni", "Juli", "August", "Septembar", "Oktobar", "Novembar",
    "Decembar"};

```

Na ovaj način, svaki element niza zauzima samo onoliki prostor koliko je zaista dug odgovarajući string. Međutim, još racionalnije rješenje je deklaracija *niza pokazivača* na konstantne znakove, kao u sljedećoj deklaraciji:

```

const char *imena_mjeseci[12] = {"Januar", "Februar", "Mart", "April",
    "Maj", "Juni", "Juli", "August", "Septembar", "Oktobar", "Novembar",
    "Decembar"};

```

Ovakva inicijalizacija je posve legalna, s obzirom da se pokazivači na konstantne znakove mogu inicijalizirati stringovnim konstantama. Primijetimo da postoji velika razlika između

prethodne tri deklaracije. U prvom slučaju se za niz “imena_mjeseci” zauzima prostor od $12 \cdot 10 = 120$ znakova nakon čega se u taj prostor kopiraju navedene stringovne konstante. U drugom slučaju, navedene stringovne konstante se ponovo kopiraju u elemente niza, ali se pri tome za svaki element niza zauzima samo onoliko prostora koliko je neophodno za smještanje odgovarajućeg stringa (uključujući i prostor za “NUL” graničnik), što ukupno iznosi prostor za 83 znaka. Očigledno je time ostvarena značajna ušteda. Međutim, u trećem slučaju se za niz “imena_mjeseci” zauzima samo prostor za 10 pokazivača (tipično 4 bajta po pokazivaču na današnjim računarima) koji se inicijaliziraju da pokazuju na navedene stringovne konstante (bez njihovog kopiranja), koje su svakako pohranjene negdje u memoriji pa se na taj način ne troši dodatni memorijski prostor. Uštede koje se na taj način postižu svakako su primjetne, i bile bi još veće da su stringovi bili duži. I ovom prilikom treba napomenuti da ne bi trebalo niz pokazivača na (nekonstantne) znakove inicijalizirati stringovnim konstantama, bez obzira što će gotovo svi kompjajleri za C++ dozvoliti nešto takvo (mada ne bi trebali).

Ako uzmemo u obzir da funkcije mogu kao rezultat vraćati *pokazivače*, možemo napraviti vrlo interesantnu i efikasnu funkciju koja kao rezultat vraća *ime mjeseca* čiji je redni broj (od 1 do 12):

```
char *ImeMjeseca(int mjesec) {
    const char *imena_mjeseci[] = {"Januar", "Februar", "Mart", "April",
        "Maj", "Juni", "Juli", "August", "Septembar", "Oktobar",
        "Novembar", "Decembar"};
    return imena_mjeseci[mjesec + 1];
}
```

Strogo rečeno ova funkcija ne vraća *ime mjeseca* (takvu funkciju bismo mogli napraviti jedino vraćanjem objekta tipa “string” kao rezultat) već *pokazivač na prvi znak imena mjeseca*, ali zbog tretmana pokazivača na znakove izgleda kao da je zaista vraćeno ime. Naime, sasvim je legalno pisati

```
cout << ImeMjeseca(5);
strcpy(neki_mjesec, ImeMjeseca(3));
```

i slične konstrukcije. Zapravo, zbog automatske konverzije nizova u pokazivače, funkcija “ImeMjeseca” se mogla napisati i pomoću običnog dvodimenzionalnih nizova znakova, ali bismo na taj način imali nepotreban utrošak memorije.

Kao primjer ovakve upotrebe nizova pokazivača na znakove, napravićemo ponovo program koji određuje dan u sedmici za zadani datum unutar 2000. godine, koji smo već pisali u poglavlju o nizovima, ali u znatno skraćenoj varijanti, u kojoj je umjesto glomazne “switch” naredbe upotrijebljen niz pokazivača na imena pojedinih dana u sedmici:

```

#include <iostream>
using namespace std;
int main() {
    const int mjeseci[12] = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    char *imena_dana[7] = {"Subota", "Nedjelja", "Ponedjeljak", "Utorak",
                           "Srijeda", "Četvrtak", "Petak"};
    int dan, mjesec;
    bool pogresan_datum;           // "true" ako je unesen neispravan datum
    do {
        cout << "Unesi datum u obliku DAN MJESEC: ";
        cin >> dan >> mjesec;
        pogresan_datum = !cin || mjesec < 1 || mjesec > 12
            || dan < 1 || dan > mjeseci[mjesec];
        if(pogresan_datum) cout << "Unijeli ste besmislen datum!\n";
        if(!cin) cin.clear();
        cin.ignore(10000, '\n');
    } while(pogresan_datum);
    int broj_proteklih_dana = dan - 1;
    for(int i = 0; i < mjesec - 1; i++)
        broj_proteklih_dana += mjeseci[i];
    cout << imena_dana[broj_proteklih_dana % 7];
    return 0;
}

```

Nizovi pokazivača su zaista korisne strukture podataka, i njihove primjene u jeziku C++ su višestruke, tako da ćemo se sa njima intenzivno susretati u poglavljima koja slijede. Da bismo bolje upoznali njihovu pravu prirodu, razmotrimo još jedan ilustrativan primjer.

Pretpostavimo, na primjer, da je neophodno sa tastature unijeti nekoliko rečenica, pri čemu broj rečenica nije unaprijed poznat. Dalje, neka je poznato da dužine rečenica mogu znatno varirati, ali da neće biti duže od 1000 znakova. Jasno je da nam je potrebna neka dvodimenzionalna znakovna struktura podataka, ali kakva? Kako broj rečenica nije poznat, prva dimenzija nije poznata unaprijed. Što se tiče druge dimenzije, mogli bismo je fiksirati na 1000 znakova, ali je veoma je neracionalno za svaku rečenicu rezervirati prostor od 1000 znakova, s obzirom da će većina rečenica biti znatno kraća. Mnogo je bolje za svaku rečenicu zauzeti *onoliko prostora koliko ona zaista zauzima*. Jedan način da to učinimo, i to zaista dobar, je korištenje niza (recimo, dinamičkog) čiji su elementi tipa “string”. Međutim, ovdje ćemo demonstrirati rješenje koje se zasniva na korištenju (dinamičkog) *niza pokazivača*.

Rješenje se zasniva na činjenici da nizovi pokazivača omogućavaju simulaciju “grbavih nizova”, odnosno dvodimenzionalnih nizova čiji redovi nemaju isti broj znakova. Tako možemo formirati grbavi niz rečenica, odnosno dvodimenzionalni niz znakova u kojem svaki red može imati ima različit broj znakova. Sljedeći primjer ilustrira kako to najbolje možemo izvesti:

```

int broj_recenica;
cout << "Koliko želite unijeti rečenica: ";
cin >> broj_recenica;
cin.ignore(1000, '\n');
char **recenice = new char*[broj_recenica];
for(int i = 0; i < broj_recenica; i++) {

```

```

char radni_prostor[1000];
cin.getline(radni_prostor, sizeof radni_prostor);
recenice[i] = new char[strlen(radni_prostor) + 1];
strcpy(recenice[i], radni_prostor);
}
for(int i = broj_recenica - 1; i >= 0; i--)
cout << recenice[i] << endl;

```

U ovom primjeru svaku rečenicu prvo unosimo u pomoćni (statički) niz “radni_prostor”. Nakon toga, pomoću funkcije “strlen” saznajemo koliko unesena rečenica zaista zauzima prostora, alociramo prostor za nju (dužina se uvećava za 1 radi potrebe za smještanjem “NUL” graničnika) i kopiramo rečenicu u alocirani prostor. Postupak se ponavlja za svaku unesenu rečenicu. Primijetimo da je “radni_prostor” deklariran lokalno unutar petlje, tako da se on automatski uništava čim se petlja završi. Kada smo unijeli rečenice, sa njima možemo raditi šta god želimo (u navedenom primjeru, ispisujemo ih u obrnutom poretku u odnosu na poredak u kojem smo ih unijeli). Na kraju, ne smijemo zaboraviti uništiti alocirani prostor onog trenutka kada nam on više nije potreban (što u prethodnom primjeru radi jednostavnosti nije izvedeno), inače će doći do curenja memorije.

Opisano rješenje radi veoma dobro. Ipak, ono je znatno komplikovanije od rješenja koje se zasniva na nizu čiji su elementi tipa “string”. Prvo, u prikazanom primjeru moramo se eksplisitno brinuti o alokaciji memorije za svaku rečenicu, dok na kraju moramo eksplisitno uništiti alocirani prostor. U slučaju korištenja tipa “string” sve se odvija automatski (mada je, interno posmatrano, niz čiji su elementi tipa “string” u memoriji organiziran na vrlo sličan način kao i opisani grbavi niz znakova, odnosno niz pokazivača na početke svakog od stringova). Dalje, da bi gore prikazani primjer bio pouzdan, bilo bi potrebno dodati dosta složene “try” – “catch” konstrukcije, koje će u slučaju da tokom rada ponestane memorije, obezbijediti brisanje do tada alociranog prostora (pri upotrebi tipa “string”, ovo se također odvija automatski). Ipak, uskoro ćemo vidjeti da sa aspekta efikasnosti, rješenje zasnovano na upotrebni nizova pokazivača može imati izrazite prednosti u odnosu na rješenje zasnovano na nizu elemenata tipa “string”, pogotovo u slučajevima kada sa elementima niza treba vršiti neke operacije koje zahtijevaju *intenzivno premještanje elemenata niza* (što se dešava, recimo, prilikom sortiranja). To je dovoljan razlog da trebamo razmotriti i takva rješenja.

Iz izloženog primjera vidimo da grbavi nizovi mogu biti izuzetno efikasni sa aspekta utroška memorije. Međutim, pri radu sa njima neophodan je izvjestan oprez upravo zbog činjenice da njegovi redovi mogu imati različite dužine. Prepostavimo, recimo, da je potrebno *sortirati* uneseni niz rečenica iz prethodnog primjera. Ne ulazeći u samu tehniku kako se obavlja sortiranje (o tome ćemo govoriti kasnije), posve je jasno da je pri ma kakvom postupku sortiranja ma kakvog niza neophodno s vremena na vrijeme vršiti razmjenu elemenata niza ukoliko se ustanovi da su oni u neispravnom poretku. Za slučaj da sortiramo niz rečenica “recenice” koji je zadan kao običan dvodimenzionalni niz znakova, razmjenu rečenica koje se nalaze na *i*-toj i *j*-toj poziciji vjerovatno bismo izveli ovako:

```

char pomocna[1000];
strcpy(pomocna, recenice[i]);
strcpy(recenice[i], recenice[j]);
strcpy(recenice[j], pomocna);

```

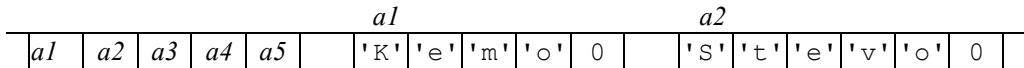
Ova strategija bi vrijedila i za slučaj dinamičkog niza rečenica, u slučaju da smo za svaku rečenicu alocirali podjednak prostor (mada ćemo uskoro vidjeti da za slučaj dinamičkih nizova postoji mnogo efikasniji pristup). S druge strane, ukoliko je niz “recenice” izведен kao niz elemenata tipa “string”, razmjenu bismo vjerovatno izvršili ovako (na isti način kao kad razmjenjujemo recimo elemente niza brojeva):

```
string pomocna = recenice[i];
recenice[i] = recenice[j]; recenice[j] = pomocna;
```

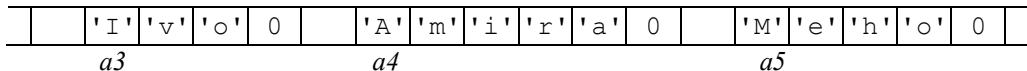
Šta bismo, međutim, trebali uraditi ukoliko koristimo niz “recenice” koji je organiziran kao *grbavi niz znakova*, odnosno niz pokazivača na početke svake od rečenica? Jasno je da varijanta u kojoj bismo kopirali čitave rečenice pomoću funkcije “`strcpy`” nije prihvatljiva, s obzirom da smo za svaku rečenicu alocirali tačno onoliko prostora koliko ona zaista zauzima. Stoga bi kopiranje duže rečenice na mjesto koje je zauzimala kraća rečenica neizostavno dovelo do pisanja u nedozvoljeni dio memorije (npr. preko neke druge rečenice), što bi moglo vrlo vjerovatno dovesti i do kraha programa. Da bismo stekli uvid u to šta zaista trebamo učiniti, razmislimo malo o tome šta u ovom slučaju tačno predstavlja niz “recenice”. On je *niz pokazivača* koji pokazuju na početke svake od rečenica. Međutim, da bismo efektivno sortirali takav niz, uopće nije potrebno da ispremještamo pozicije rečenica u memoriji: *dovoljno je samo da ispremještamo pokazivače koji na njih pokazuju!* Dakle, umjesto da razmijenimo čitave rečenice, dovoljno je samo razmijeniti pokazivače koji na njih pokazuju, kao u sljedećem isječku:

```
char *pomocni = recenice[i];
recenice[i] = recenice[j]; recenice[j] = pomocni;
```

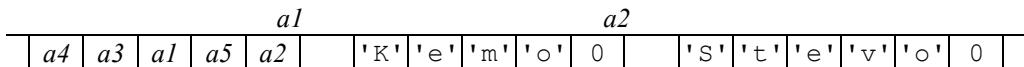
Da bismo lakše shvatili šta se zaista dešava, pretpostavimo da niz “recenice” sadrži 5 imena “Kemo”, “Stevo”, “Ivo”, “Amira” i “Meho” (on zapravo sadrži *pokazivače* na mesta u memoriji gdje se čuvaju zapisi tih imena). Stoga situaciju u memoriji možemo ilustrirati sljedećom slikom (pri čemu oznake $a1$, $a2$, $a3$, $a4$ i $a5$ označavaju neke adrese čija je tačna vrijednost nebitna):



recenice



Nakon izvršenog sortiranja, u kojem se vrši razmjena pokazivača (sama tehnika kako obaviti sortiranje u ovom trenutku nije bitna), situacija u memoriji izgledala bi ovako:



rečenice

	'I'	'v'	'o'	0		'A'	'm'	'i'	'r'	'a'	0		'M'	'e'	'h'	'o'	0
	<i>a3</i>					<i>a4</i>							<i>a5</i>				

Vidimo da ukoliko pokazivače čitamo redom, oni pokazuju respektivno na imena “Amira”, “Ivo”, “Kemo”, “Meho” i “Stevo”, a to je upravo sortirani redoslijed! Ovim ne samo da smo riješili problem, nego smo i samo sortiranje učinili *znatno efikasnijim*. Naime, sigurno je mnogo lakše razmijeniti dva pokazivača u memoriji, nego premještati čitave rečenice upotrebom funkcije “`strcpy`”. Interesantno je da postupak razmjene sa aspekta sintakse obavljamo na isti način kao da se radi o nizu čiji su elementi tipa “`string`”, mada treba voditi računa da se u ovom slučaju uopće ne razmjenjuju *stringovi* već samo *pokazivači*. Međutim, bitno je napomenuti da za poređenje rečenica (sa ciljem utvrđivanja da li su u ispravnom poretku) i dalje moramo koristiti funkciju “`strcmp`”, jer bi obična upotreba relacionih operatora “`<`” ili “`>`” uporedjivala *adrese* na kojima se rečenice nalaze (u skladu sa pokazivačkom aritmetikom), a to nije ono što nam treba (relacione operatore smijemo koristiti samo ukoliko su elementi niza koje upoređujemo tipa “`string`”).

Izloženi primjer ujedno ilustrira najvažniju primjenu nizova pokazivača u jeziku C++. Kada god je potrebno kreirati nizove čiji su elementi *masivni objekti* (poput *stringova*) sa kojima je nezgrapno vršiti različite manipulacije poput premještanja elemenata niza, mnogo je isplatnije kreirati *niz pokazivača na objekte*, a same objekte dinamički alocirati (sa ovakvim primjerima ćemo se sve češće susretati u poglavljima koja slijede, jer ćemo se susretati sa sve više i više tipova masivnih objekata). Tada, umjesto manipulacija sa samim objektima vršimo manipulacije sa pokazivačima koji na njih pokazuju, što je neuporedivo efikasnije. Postoje čak i objekti koji se uopće ne mogu premještati ili kopirati, tako da se manipulacije sa takvim objektima mogu vršiti jedino indirektno, putem pokazivača koji na njih pokazuju!

Od “egzotičnih” vrsta pokazivača potrebno je još objasniti *pokazivače na funkcije*. Oni se deklariraju slično kao i prototipovi funkcija, samo uz jednu dodatnu zvjezdicu i par zagrada. Na primjer, deklaracija

```
int (*pok_na_fn) (int, int);
```

deklarira promjenljivu “`pok_na_fn`” koja predstavlja *pokazivač na funkciju koja prima dva cjelobrojna parametra, a vraća cjelobrojni rezultat*. Dodatni par zagrada je neophodan, jer bi deklaracija

```
int *pok_na_fn(int, int);
```

predstavljala prototip *funkcije* “`pok_na_fn`” koja prima dva cjelobrojna parametra, a vraća pokazivač na cijeli broj kao rezultat. Kao i u slučaju nizova pokazivača i pokazivača na nizove, i ovdje na neki način “funkcija” ima prioritet u odnosu na “pokazivač”.

Pokazivači na funkcije u sebi sadrže *adresu funkcije*, što je zapravo *adresa mašinskog programa* koji izvršava datu funkciju. Stoga se ovi pokazivači ne mogu dereferencirati, niti se na njih može primjenjivati pokazivačka aritmetika (ipak, oni se mogu porediti međusobno). Međutim, pokazivačima na funkcije se može dodijeliti ime neke funkcije, pod uvjetom da je njen tip saglasan sa tipom funkcije na koju funkcija pokazuje. Pored toga, na pokazivače na funkciju može se primijeniti *poziv funkcije*, kao da se radi o

funkciji (prilikom takvog poziva, poziva se funkcija na koju pokazivač pokazuje). Na primjer, neka je deklariran pokazivač na funkciju “`pok_na_fn`“ na već opisani način, i neka su date dvije funkcije “`Min`“ i “`Max`“ (od kojih prva vraća manji a druga veći od dva cjelobrojna argumenta):

```
int Min(int p, int q) {
    return (p < q) ? p : q;
}

int Max(int a, int b) {
    return (p < q) ? p : q;
}
```

Tada su dozvoljene konstrukcije poput sljedeće (uz pretpostavku da su “`a`“ i “`b`“ propisno deklarirane cjelobrojne promjenljive):

```
pok_na_fn = Min;
cout << "Minimum je: " << pok_na_fn(a, b);
pok_na_fn = Max;
cout << "Maksimum je: " << pok_na_fn(a, b);
```

Dakle, pokazivači na funkcije omogućavaju *indirektan poziv funkcija* i tokom izvršavanja programa mogu pokazivati na različite funkcije, kao što pokazivači na obične promjenljive omogućavaju indiretan pristup sadržaju promjenljivih i tokom izvršavanja programa mogu pokazivati na različite promjenljive.

Da bismo uvidjeli primjenu pokazivača na funkcije, razmotrimo jedan konkretan i ilustrativan primjer iz numeričke analize. Neka je potrebno odrediti nulu neke funkcije $f(x)$ unutar intervala (a, b) . Ukoliko je funkcija $f(x)$ *neprekidna* na intervalu (a, b) i ukoliko su vrijednosti $f(a)$ i $f(b)$ suprotnog znaka (tj. ukoliko je $f(a) \cdot f(b) < 0$), tada funkcija *sigurno* ima nulu unutar intervala (a, b) , koja se lako može odrediti postupkom *polovljenja intervala*. Prvo biramo tačku $c = (a+b)/2$ tačno na sredini intervala (a, b) i ispitujemo u kojem od podintervala (a, c) odnosno (c, b) funkcija mijenja znak. Nakon toga, interval koji ne sadrži nulu odbacimo, čime smo širinu intervala sveli na polovinu. Postupak možemo ponavljati sve dok širina intervala ne postane manja od unaprijed zadane tačnosti ϵ . U skladu sa opisanim postupkom lako je napisati funkciju “`NulaFunkcije`“ kojoj se kao parametri zadaju granice intervala i tačnost. Jedna izvedba ove funkcije je sljedeća (u kojoj je tačnost definirana kao parametar sa podrazumijevanom vrijednošću 10^{-7}):

```
double NulaFunkcije(double a, double b, double eps = 1e-7) {
    if(F(a) * F(b) > 0) throw "F(a) i F(b) su istog znaka!";
    if(a > b) throw "Nekorektan interval!";
    while(b - a > eps) {
        double c = (a+b)/2;
        if(F(a) * F(c) < 0) b = c;
        else a = c;
    }
    return (a + b) / 2;
}
```

Ova funkcija poziva funkciju “`F`“ čija se nula traži. Na primjer, ukoliko želimo da pronademo nulu funkcije $f(x) = x^3 - x - 1$, morali bismo prethodno negdje u programu definirati funkciju “`F`“ kao

```
double F(double x) {
    return x * x * x - x - 1;
}
```

Nakon toga, funkciju "NulaFunkcije" bismo mogli pozvati na uobičajeni način:

```
cout << "Nula F(x) na (1,2) je " << NulaFunkcije(1, 2) << endl;
```

Osnovni problem sa ovako napisanom funkcijom "NulaFunkcije" je u tome što ona uvijek poziva jednu te istu funkciju – funkciju "F". Šta ukoliko nam je potrebno da u istom programu nađemo nulu funkcije $f(x)$ na intervalu $(1,2)$, zatim nulu funkcije $g(x)=\cos x-x$ na intervalu $(0,1)$ kao i nulu funkcije $\sin x$ na intervalu $(3,4)$? Sa ovako napisanom funkcijom "NulaFunkcije" morali bismo napisati tri verzije iste funkcije, od kojih bi jedna pozivala funkciju "F", druga funkciju "G", a treća funkciju "sin". Prirodno je postaviti pitanje može li se nekako funkciji "NulaFunkcije" prilikom poziva saopćiti *koju funkciju treba da poziva*. Odgovor na ovo pitanje nude upravo pokazivači na funkcije. Naime, umjesto pozivanja fiksne funkcije "F", poziv treba vršiti *indirektno* putem *pokazivača na funkciju*. Za tu svrhu, funkciji "NulaFunkcije" treba dodati novi formalni parametar " f " koji će biti tipa *pokazivač na funkciju koja prima jedan realan argument, a vraća realan broj kao rezultat*. Nova verzija funkcije "NulaFunkcije" mogla bi sada izgledati ovako:

```
double NulaFunkcije(double (*f)(double), double a, double b,
                     double eps = 1e-7) {
    if(f(a) * f(b) > 0) throw "f(a) i f(b) su istog znaka!";
    if(a > b) throw "Nekorektan interval!";
    while(b - a > eps) {
        double c = (a+b)/2;
        if(f(a) * f(c) < 0) b = c;
        else a = c;
    }
    return (a + b) / 2;
}
```

Pretpostavimo da je u programu pored funkcije "F" na propisan način deklarirana i funkcija "G" (u skladu sa svojom definicijom). Tada bismo funkciju NulaFunkcije mogli iskoristiti na sljedeći način:

```
cout << "Nula F(x) na (1,2) je " << NulaFunkcije(F, 1, 2) << endl;
cout << "Nula G(x) na (1,2) je " << NulaFunkcije(G, 1, 2) << endl;
cout << "Nula sin(x) na (3,4) je " << NulaFunkcije(sin, 1, 2) << endl;
```

Primijetimo da se prilikom poziva funkcije kao prvi parametar navodi *ime funkcije* koju funkcija NulaFunkcije treba da poziva. Pri tome je svejedno da li se radi o korisnički napisanoj funkciji poput "F" ili "G", ili ugrađenoj funkciji "sin", sve dok njen tip odgovara tipu na koji pokazivač pokazuje. Potrebno je još napomenuti da između funkcija i pokazivača na funkcije postoji slična veza kao između nizova i pokazivača na elemente nizova. Naime, ime funkcije kada se upotrijebi *samo za sebe* (bez zagrade koje označavaju poziv funkcije) automatski se *konvertira u pokazivač na funkciju*. Stoga se ne treba čuditi što će naredba poput

```
cout << sin;
```

ispisati nekakav heksadekadni broj (adresu mašinskog programa koji izvršava funkciju "sin". Također, ovim postaje jasnije šta se zapravo dešava u slučaju da zaboravimo navesti par zagrada "()" prilikom pozivanja neke funkcije koja nema parametara.

Bitno je napomenuti da i pokazivači na funkcije sadrže slučajne vrijednosti dok im se eksplicitno ne dodijeli vrijednost. To može potencijalno biti veoma opasno. Na primjer, sljedeća sekvenca naredbi

```
void (*pok_na_fn) (int);
pok_na_fn(5);
```

skoro sigurno će “srušiti” program, jer pokazivač “pok_na_fn” sadrži na početku nepredvidljivu vrijednost, pa će i indirektni poziv posredstvom takvog pokazivača pozvati mašinski program sa nepredvidljive adrese, i uz nepredvidljive posljedice.

Standardna biblioteka “algorithm”, o kojoj smo već govorili, sadrži mnoštvo funkcija koje kao neki od svojih parametara prihvataju pokazivače na funkcije, čime je ostvarena velika fleksibilnost, jer na taj način možemo preciznije odrediti njihovo ponašanje. Neke od ovih funkcija smo već upoznali, ali sa drugačijim parametrima (očito se radi o preklapanju funkcija). Na ovom mjestu ćemo spomenuti neke od jednostavnijih funkcija iz biblioteke “algorithm” koje prihvataju pokazivače na funkcije kao parametre (više takvih funkcija ćemo upoznati kada budemo govorili o nekim specifičnijim primjenama). Pri opisu značenja parametara korištene su iste konvencije kao i ranije:

<code>for_each(p1, p2, f)</code>	Poziva uzastopno funkciju f prenoseći joj redom kao argument sve elemente između pokazivača $p1$ i $p2$.
<code>transform(p1, p2, p3, f)</code>	Poziva uzastopno funkciju f prenoseći joj redom kao argument sve elemente između pokazivača $p1$ i $p2$. Vrijednosti vraćene iz funkcije smještaju se redom u blok memorije koji počinje od pokazivača $p3$.
<code>transform(p1, p2, p3, p4, f)</code>	Poziva uzastopno funkciju f prenoseći joj redom kao prvi argument sve elemente između pokazivača $p1$ i $p2$, a kao drugi argument odgovarajuće elemente iz bloka koji počinje od pokazivača $p3$. Vrijednosti vraćene iz funkcije smještaju se redom u blok memorije koji počinje od pokazivača $p4$.
<code>count_if(p1, p2, f)</code>	Vraća kao rezultat broj elemenata između pokazivača $p1$ i $p2$ za koje funkcija f vraća kao rezultat “ true ” kad joj se proslijede kao argument.
<code>find_if(p1, p2, f)</code>	Vraća kao rezultat pokazivač na prvi element u bloku između pokazivača $p1$ i $p2$ za koje funkcija f vraća kao rezultat “ true ” kad joj se proslijedi kao argument. Ukoliko takav element ne postoji, vraća $p2$ kao rezultat.
<code>replace_if(p1, p2, f, v)</code>	Zamjenjuje sve elemente između pokazivača $p1$ i $p2$ za koje funkcija f vraća kao rezultat “ true ” kad joj se proslijede kao argument, sa elementima sa vrijednošću v .
<code>replace_copy_if(p1, p2, p3, f, v)</code>	Kopira blok elemenata između pokazivača $p1$ i $p2$ na lokaciju određenu pokazivačem $p3$ uz zamjenu svakog elementa za koji funkcija f vraća kao rezultat “ true ” kad joj se proslijedi kao argument, sa elementom koji ima vrijednost v .
<code>remove_if(p1, p2, f)</code>	“Uklanja” sve elemente iz bloka između pokazivača $p1$ i $p2$ za koje funkcija f vraća kao rezultat “ true ” kad joj se proslijede kao argument, tako što ih premješta na

`remove_copy_if(p1, p2, p3, f)`

`equal(p1, p2, p3, f)`

kraj bloka. Kao rezultat vraća pokazivač na dio bloka u koji su premješteni uklonjeni elementi (tako da, na kraju, u bloku između pokazivača $p1$ i vraćenog pokazivača neće biti niti jedan element za koji funkcija f vraća kao rezultat “`true`”). Ukoliko takvih elemenata nije bilo, funkcija vraća $p2$ kao rezultat.

Kopira blok elemenata između pokazivača $p1$ i $p2$ na lokaciju određenu pokazivačem $p3$, uz izbacivanje elemenata za koji funkcija f vraća kao rezultat “`true`” kad joj se proslijede kao argument. Kao rezultat, funkcija vraća pokazivač koji pokazuje tačno iza posljednjeg elementa odredišnog bloka.

Vraća kao rezultat logičku vrijednost “`true`” ukoliko funkcija f vraća kao rezultat “`true`” za sve parove argumenata od kojih se prvi argumenti uzimaju redom iz bloka elemenata između pokazivača $p1$ i $p2$, a drugi argumenti iz odgovarajućih elemenata iz bloka na koji pokazuje pokazivač $p3$. U suprotnom, funkcija vraća logičku vrijednost “`false`”.

Razmotrimo na nekoliko vrlo jednostavnih primjera kako se mogu koristiti neke od opisanih funkcija. Pretpostavimo da imamo tri niza “`niz1`”, “`niz2`” i “`niz3`” koji sadrže 10 cijelih brojeva, i da su u programu definirane sljedeće funkcije:

```
void Ispisi(int x) {
    cout << x << endl;
}

int Kvadrat(int x) {
    return x * x;
}

int Zbir(int x, int y) {
    return x + y;
}

bool DaLiJeParan(int x) {
    return x % 2 == 0;
}
```

Tada naredba

```
for_each(niz1, niz1 + 10, Ispisi);
```

ispisuje sve elemente niza “niz1” na ekran, svaki element u posebnom redu. Naredba

```
transform(niz1, niz1 + 10, niz2, Kvadrat);
```

prepisuje u niz “niz2” kvadrate odgovarajućih elemenata iz niza “niz1”. Naredba

```
transform(niz1, niz1 + 10, niz1, Kvadrat);
```

zamjenjuje sve elemente niza “niz1” njihovim kvadratima (odnosno prepisuje kvadrate elemenata niza preko njih samih). Naredba

```
transform(niz1, niz1 + 10, niz2, niz3, Zbir);
```

prepisuje u niz “niz3” zbroje odgovarajućih elemenata iz nizova “niz1” i “niz2”. Konačno, sekvenca naredbi

```
int *pok = find_if(niz1, niz1 + 10, DaLiJeParan);
if(pok == niz1 + 10) cout << "Niz ne sadrži parne elemente\n";
else cout << "Prvi parni element u nizu je " << *pok << endl;
```

ispisuje prvi parni element iz niza “niz1”, ili poruku da takvog elementa nema.

Ako razmotrimo navedene primjere, primijetićemo nešto interesantno. Ovi primjeri pozivaju funkcije “`for_each`”, “`transform`” i “`find_if`” (koje mi nismo napisali, i za koje ne moramo znati kako su napisane), a koje ponovo pozivaju funkcije “`Ispisi`”, “`Kvadrat`”, “`Zbir`” i “`DaLiJeParan`” koje smo mi napisali i koje pripadaju našem programu. *Mi ne vidimo eksplicitno odakle se ove funkcije pozivaju*. Na taj način, pokazivači na funkcije omogućavaju da neka funkcija koja je ugrađena u sam programske jezik ili operativni sistem ponovo poziva funkciju iz našeg programa (tako što ćemo joj npr. imati funkcije prosljediti kao parametar), iako mi ne vidimo odakle se taj poziv odvija. Ovakav mehanizam poziva naziva se *povratni poziv* (engl. *callback*), i predstavlja osnovu programiranja za sve operativne sisteme koji su upravljeni tokom događaja, kao što su operativni sistemi iz MS Windows serije operativnih sistema. Na primjer, funkcije koje iscrtavaju menije na ekranu mogu kao parametre očekivati pokazivače na funkcije koje će biti pozvane kada korisnik izabere neku od opcija iz menija (s obzirom da funkcije koje samo *iscrtavaju* menije ne mogu unaprijed znati koje akcije korisnik želi izvršiti izborom pojedinih opcija u meniju). Dakle, razumijevanje povratnih poziva je od *vitalne važnosti da bi se shvatilo programiranje aplikacija za MS Windows operativne sisteme*.

Na kraju ovog poglavlja, recimo još i nekoliko riječi o *nizovima pokazivača na funkcije*. Ovakve egzotične tvorevine također mogu imati veoma interesantne primjene. Prepostavimo

da smo pomoću “**typedef**“ naredbe definirali novi tip “**PokNaFn**“ koji predstavlja tip pokazivača na funkciju koja prihvata jedan cijeli broj kao argument a vraća kao rezultat realan broj:

```
typedef double (*PokNaFn) (int);
```

Nakon ovakve definicije sasvim je lako definirati niz pokazivača na funkcije:

```
PokNaFn f[10];
```

Ovim je “f“ deklariran kao niz od 10 pokazivača na funkcije (koje prihvataju jedan cijeli broj kao argument, a vraćaju realni broj kao rezultat). Ovakav niz se može deklarirati i neposredno, bez “**typedef**“ naredbe:

```
double (*f[10]) (int);
```

Čemu ovakvi nizovi mogu poslužiti? Zamislimo, na primjer, da imamo 10 funkcija nazvanih “f1”, “f2”, “f3”, itd. do “f10“, i da trebamo ispisati vrijednosti ovih funkcija za jednu te istu vrijednost argumenta (recimo “x”). Jedno očigledno i ne osobito elegantno rješenje je sljedeće:

```
cout << f1(x) << endl << f2(x) << endl << f3(x) << endl << f4(x)  
<< endl << f5(x) << endl << f6(x) << endl << f7(x) << endl << f8(x)  
<< endl << f9(x) << endl << f10(x) << endl;
```

Nizovi pokazivača na funkcije nude znatno elegantije rješenje, koje je lako uopćiti na proizvoljan broj funkcija:

```
double (*f[10])(int) = {f1, f2, f3, f4, f5, f6, f7, f8, f9, f10};  
for(int i = 0; i < 10; i++) cout << f[i](x) << endl;
```

Druga primjena nizova pokazivača na funkcije može biti kada unaprijed ne znamo koja tačno funkcija od nekoliko funkcija treba biti pozvana nad nekim argumentom, nego se to tek sazna tokom izvođenja programa. Tada je imena svih funkcija moguće smjestiti u niz pokazivača na funkcije, a zatim kad se utvrdi koja se od njih zaista treba pozvati, poziv može izvršiti preko odgovarajućeg indeksa. Iskusniji čitalac će vjerovatno uočiti i mogućnosti za brojne druge primjene pored ovdje navedenih primjena.

28. Sortiranje i pretraživanje

Sortiranje i pretraživanje vjerovatno spadaju u najčešće operacije koje se u praktičnim primjenama obavljaju nad skupinama podataka. Mada detaljna analiza algoritama za sortiranje i pretraživanje i njihove efikasnosti spada u domen teorije algoritama, sa elementarnim postupcima za obavljanje ovih operacija neophodno se upoznati već na osnovnim kursevima programiranja, jer je bez njih nemoguće kreirati iole upotrebljiviji program.

Sortiranje nizova predstavlja operaciju koja se oslanja na intenzivnom razmjenjivanju pozicija pojedinih elemenata niza sa ciljem njihovog dovođenja u traženi poredak. Da bismo se bolje koncentrirali na sam postupak sortiranja a ne na razmjenu elemenata, u svim primjerima koji slijede pretpostavčemo da postoji generička funkcija "Razmijeni" koja vrši razmjenu svoja dva parametra (inače, identična funkcija pod nazivom "swap" već postoji u standardnoj biblioteci "algorithm"). Ova funkcija mogla bi izgledati recimo ovako:

```
template <typename Tip>
inline void Razmijeni(Tip &a, Tip &b) {
    Tip pomocna = a;
    a = b; b = pomocna;
}
```

Razmotrimo sada kako bismo mogli obaviti sortiranje nekog niza od n elemenata u recimo *rastući poredak*. Rješenje koje se prvo nameće je da prvo pronademo *najmanji element* pa da ga dovedemo na prvo mjesto u nizu (pri tome, element koji se već nalazio na prvom mjestu treba prebaciti na mjesto na kojem se prethodno nalazio najmanji element, odnosno treba izvršiti *razmjenu* prvog i najmanjeg elementa). U drugom koraku, pronalazimo najmanji element počev od drugog elementa niza nadalje, a zatim ga dovedemo na drugo mjesto u nizu. Općenito, u nekom k -tom koraku pronalazimo najmanji element počev od k -tog elementa niza nadalje, a zatim ga dovedemo na k -to mjesto u nizu (na njegovo mjesto dovodimo element koji se do tada nalazio na k -tom mjestu). Očigledno, nakon $n-1$ koraka, niz je sortiran. Opisana ideja prikazana je na sljedećoj slici na nasumično odabranom nizu. U svakom koraku, nađeni najmanji element prikazan je podebljano, a dva elementa koja treba razmijeniti prikazana su podvučeno. U slučaju da je najmanji element već na pravom mjestu, samo je on podvučen:

3	5	1	8	6	9	5	7
1	<u>5</u>	3	8	6	9	5	7
1	3	5	8	6	9	5	7
1	3	5	<u>8</u>	6	9	5	7
1	3	5	<u>5</u>	6	9	8	7
1	3	5	5	<u>6</u>	9	8	7
1	3	5	5	6	<u>9</u>	8	7
1	3	5	5	6	7	8	9

Na osnovu ove ideje, možemo lako napraviti generičku funkciju "KlasicniSort" koja sortira niz "niz" od "n" elemenata u rastući poredak (pri tome se za elemente niza pretpostavlja da se mogu međusobno porebiti):

```
template <typename UporediviTip>
void KlasicniSort(UporediviTip niz[], int n) {
    for(int k = 0; k < n - 1; k++) {
        int gdje_je_min(k);
        for(int i = k + 1; i < n; i++)
            if(niz[i] < niz[gdje_je_min]) gdje_je_min = i;
```

```

    if(gdje_je_min != k) Razmijeni(niz[k], niz[gdje_je_min]);
}
}

```

Razumije se da se opisana ideja mogla realizirati na mnoštvo različitih načina, ali gore prikazana izvedba je vjerovatno jedna od najefikasnijih. U ovoj izvedbi, umjesto samog pamćenja najmanjeg elementa, mi pamtimo *gdje se nalazi najmanji element* (odnosno indeks do tada pronađenog najmanjeg elementa) u promjenljivoj “*gdje_je_min*“. Primijetimo da bismo ovu promjenljivu svakako morali uvoditi, s obzirom da nakon pronalaženja najmanjeg elementa moramo element koji se do tada nalazio na k -tom mjestu razmijeniti sa njim, a za to nam je neophodno poznavanje njegove pozicije. Primijetimo da uvjet “*gdje_je_min != k*“ obezbjeđuje da razmjenu ne vršimo ukoliko se element već nalazio na pravoj poziciji. Ukoliko bismo izbacili ovaj uvjet, element koji je na pravom mjestu bismo razmjenjivali sa *samim sobom*, što ne bi štetilo, ali bespotrebno usporava postupak.

Razmotrimo sada koliko je efikasna napisana funkcija. Spoljašnja petlja (tj. petlja sa brojačkom promjenljivom k) izvršava se $n-1$ puta (za k od 0 do $n-2$), dok se unutrašnja petlja izvršava $n-k-1$ put (zavisno od k), tako da ukupan broj prolazaka kroz unutrašnju petlju iznosi $(n-1)+(n-2)+\dots+3+2+1$ odnosno $n(n-1)/2$ puta (ovo je također i broj izvršenih upoređivanja). Za veliko n broj prolazaka kroz petlju kao i broj izvršenih upoređivanja iznosi približno $n^2/2$, tako da je vrijeme izvršavanja funkcije približno proporcionalno *kvadratu* broja elemenata niza. Stoga se kaže da ova funkcija ima *kvadratnu složenost*. S druge strane, broj izvršenih *razmjena* elemenata nikad neće biti veći od n , što je očigledno iz same funkcije.

Opisani postupak spada u *najlošije* (odnosno najsporije) postupke za sortiranje i obično se naziva *sortiranje izborom* (engl. *selection sort*). Stoga je on prihvatljiv samo za nizove sa relativno malim brojem elemenata (recimo 50–100). Recimo, za $n=10000$ sortiranje ovim postupkom traži blizu 50 miliona prolazaka kroz petlju, što definitivno mora trajati prilično dugo. Ipak, od svih postupaka za sortiranje, ovaj postupak traži najmanji broj razmjena elemenata. Stoga se smatra da je ovaj postupak veoma pogodan u slučaju kada niz ima mali broj elemenata, a koji su u isto vrijeme masivni (npr. dugački stringovi) tako da njihova razmjena može biti zahtjevna. Pored toga, primijetimo da trajanje sortiranja ovim postupkom ne ovisi od toga kakva je struktura elemenata niza. Bez obzira da li su elementi niza potpuno nasumično razbacani ili je niz već skoro sortiran (pa čak i posve sortiran), sortiranje izborom trajeće podjednako dugo.

Prilikom sortiranja nizova sa malim brojem elemenata, često se koristi jedna modifikacija sortiranja izborom, koja dovodi do zaista jednostavne izvedbe. Naime, u procesu traženja minimuma možemo čim primijetimo da je tekući element manji od k -tog odmah izvršiti njegovu razmjenu sa k -tim elementom. Na kraju će najmanji element sigurno doći na k -to mjesto. Ovaj postupak je ilustriran na sljedećoj slici, u kojoj su elementi koji se razmjenjuju također prikazani podvučeno:

3	5	1	8	6	9	5	7
1	5	3	8	6	9	5	7
1	3	5	8	6	9	5	7
1	3	5	6	8	9	5	7
1	3	5	5	8	9	6	7
1	3	5	5	6	9	8	7
1	3	5	5	6	8	9	7
1	3	5	5	6	7	9	8
1	3	5	5	6	7	8	9

Funkcija koja realizira prikazani postupak mogla bi izgledati recimo ovako:

```
template <typename UporediviTip>
```

```

void KlasicniSort2(UporediviTip niz[], int n) {
    for(int k = 0; k < n - 1; k++)
        for(int i = k + 1; i < n; i++)
            if(niz[i] < niz[k]) Razmijeni(niz[i], niz[k]);
}

```

Napisana funkcija je sigurno najkraća moguća izvedba neke funkcije koja obavlja postupak sortiranja niza. Međutim, ovo je vjerovatno i *najgora moguća izvedba* sa aspekta efikasnosti. Mada je broj prolazaka kroz petlju i broj izvršenih poređenja isti kao u funkciji "KlasicniSort", broj izvršenih *razmjena* može biti mnogo veći, i u najgorem slučaju može iznositi također oko $n^2/2$ (ovo se recimo dešava ukoliko je niz koji sortiramo već sortiran ali u obrnutom poretku, što nije teško provjeriti). Ukoliko je razmjena dva elementa zahtjevna operacija (tj. ukoliko su elementi niza masivni objekti), funkcija "KlasicniSort2" može biti mnogo sporija od (ionako spore) funkcije "KlasicniSort". Slična stvar vrijedi ukoliko umjesto nizova sortiramo neke druge strukture podataka (npr. datoteke na disku) kod kojih je razmjena dva elementa zahtjevna po pitanju vremena izvršavanja. Zbog toga funkciju "KlasicniSort2" smijemo koristiti samo kada sortiramo nizove sa malim brojem elemenata, koji su uz to jednostavni (odnosno čija razmjena ne predstavlja zahtjevnu operaciju). Jedini razlog zbog kojeg se funkcije poput "KlasicniSort2" uopće koriste je njihova izuzetna kratkoča i jednostavnost.

U većini kurseva programiranja razmatra se i jedna modifikacija opisanog postupka, poznata pod nazivom *mjehurasto sortiranje* (engl. *bubble sort*). Za razliku od prethodno opisanog postupka u kojem se element na *fiksnoj poziciji* poredi sa svim elementima koji slijede iza njega (uz eventualno vršenje razmjene ukoliko se ustanovi da su u neispravnom poretku), kod ovog postupka uvijek poredimos *dva susjedna elementa* (također uz vršenje razmjene u slučaju neispravnog porekta). Nakon što na taj način prodemo kroz čitav niz, najveći element će sigurno "isplivati" na kraj niza. Ovaj postupak ilustriran je na sljedećoj slici. Elementi koji se upoređuju označeni su crticama, dok su elementi u pogrešnom poretku (koji će biti razmijenjeni) označeni podebljano:

3	5	1	8	6	9	5	7
3	5	1	8	6	9	5	7
3	1	5	8	6	9	5	7
3	1	5	8	6	9	5	7
3	1	5	6	8	9	5	7
3	1	5	6	8	9	5	7
3	1	5	6	8	5	9	7
3	1	5	6	8	5	7	9

Naravno, ovim niz još nije sortiran, nego je samo najveći element doveden na svoje mjesto. Međutim, sada na ovako dobijeni niz možemo ponovo primijeniti isti postupak, čime ćemo element koji je odmah po veličini ispod najvećeg dovesti na pravo (pretposljednje) mjesto. Pri tome, postupak ne moramo provoditi do kraja niza već ga možemo prekinuti kada dođemo do pretposljednjeg elementa, s obzirom da je najveći element već doveden na svoje mjesto. Tok ovog postupka prikazan je na sljedećoj slici, pri čemu radi jednostavnosti nećemo prikazivati posljednji element koji je već doveden na svoje mjesto:

3	1	5	6	8	5	7
1	3	5	6	8	5	7
1	3	5	6	8	5	7
1	3	5	6	8	5	7
1	3	5	6	8	5	7
1	3	5	6	5	8	7
1	3	5	6	5	7	8

Očigledno, ako ovako nastavimo, nakon najviše $n-1$ prolaza niz će biti sortiran (jer u svakom prolazu

dovodimo barem jedan element na svoje mjesto). Pri tome, u svakom narednom prolazu ispitivanje možemo prekinuti jedan korak prije nego u prethodnom prolazu. Međutim, može se desiti da niz postane sortiran i prije nego što izvršimo svih $n-1$ prolaza. Na primjer, ukoliko na niz iz prethodnog primjera primijenimo još jedan prolaz, dobijamo sljedeću situaciju:

1	3	5	6	5	7
1	<u>3</u>	<u>5</u>	6	5	7
1	<u>3</u>	<u>5</u>	<u>6</u>	5	7
1	3	<u>5</u>	6	5	7
1	3	5	<u>5</u>	<u>6</u>	7
1	3	5	5	<u>6</u>	<u>7</u>

U ovom slučaju smo očigledno dobili sortiran niz već nakon trećeg prolaza. Ukoliko bismo pokušali da izvršimo još jedan prolaz, vidjeli bismo da se neće izvršiti niti jedna razmjena, što je sigurna indikacija da je niz sortiran. Ukoliko pratimo šta se u svakom prolazu dešava sa najvećim elementom, primjetićemo da najveći elementi "isplivavaju" na kraj niza poput mjehurova zraka u posudama sa tečnošću, odakle i potiče naziv ovog metoda za sortiranje. Na osnovu opisanog postupka, nije teško napisati funkciju "BubbleSort" koja implementira opisani postupak. Uočimo upotrebu logičke promjenljive "razmijenjen" koja služi za sprečavanje započinjanja novog prolaza ukoliko u prethodnom prolazu nije izvršena niti ni jedna razmjena (tj. ukoliko je sortiranje završeno):

```
template <typename UporediviTip>
void BubbleSort(UporediviTip niz[], int n) {
    bool razmijenjen(true);
    for(int k = 1; k < n && razmijenjen; k++) {
        razmijenjen = false;
        for(int i = 0; i < n - k; i++)
            if(niz[i] > niz[i + 1]) {
                Razmjeni(niz[i], niz[i + 1]);
                razmijenjen = true;
            }
    }
}
```

Primijetimo da za razliku od sortiranja izborom, funkcija "BubbleSort" završava postupak odmah nakon prvog prolaza ukoliko joj se ponudi već sortirani niz, što je svakako veoma povoljno. S druge strane, ukoliko se funkciji proslijedi niz koji je sortiran u obrnutom poretku od željenog, izvršiće se svi prolazi. Nije teško vidjeti da je u slučaju da se izvrše svi prolazi, broj prolazaka kroz petlju i broj izvršnih poređenja isti kao kod sortiranja izborom. Također, u najgorem slučaju broj izvršenih razmjena je isti kao u lošoj varijanti sortiranja izborom (odnosno funkciji "KlasicniSort2"). Drugim riječima, u najgorem slučaju, mjehurasto sortiranje je podjednako sporo kao i lošija varijanta sortiranja izborom. S druge strane, vidjeli smo da u nekim slučajevima mjehurasto sortiranje može da bude mnogo brže. Naime, za razliku od sortiranja izborom, trajanje mjehurastog sortiranja zavisi od rasporeda elemenata u nizu. Stoga je prirodno postaviti kakva je efikasnost mjehurastog sortiranja *u prosjeku*. Odgovor, nažalost, nije nimalo ohrabrujući. Slučajevi u kojima mjehurasto sortiranje radi dobro mnogo su rijeci od slučajeva u kojima ono radi loše. U prosjeku, trajanje sortiranja ovim postupkom tek je neznatno brže od lošije varijante sortiranja izborom, dok su veoma česti slučajevi da klasična (bolja) varijanta sortiranja izborom radi brže od mjehurastog sortiranja, zbog manjeg broja izvršenih razmjena. Dakle, bez obzira na činjenicu da mjehurasto sortiranje radi dobro za skoro sortirane nizove (tj. za nizove u kojima ne treba izvršiti mnogo razmjena da bi niz postao sortiran), mjehurasto sortiranje je i dalje veoma loš postupak, za koji se smatra da je čak lošiji od klasičnog sortiranja izborom (od mjehurastog sortiranja je vjerovatno gora jedino lošija varijanta sortiranja izborom).

Od jednostavnih postupaka za sortiranje potrebno je još razmotriti postupak poznat pod nazivom

sortiranje umetanjem (engl. *insertion sort*), koji je vjerovatno najbolji od jednostavnih postupaka za sortiranje. Ovaj postupak također ima osobinu da radi dobro za skoro sortirane nizove, pri čemu je tipično 2–4 puta brži od mjeđuhastog sortiranja, i ima jednostavniju implementaciju. Ideja ovog postupka je sljedeća. Pretpostavimo da je prvi $k-1$ elemenata niza sortirano, i da želimo dovesti k -ti element niza na pravu poziciju. Možemo uporediti k -ti i $k-1$ -vi element, i razmijeniti ih ukoliko su u pogrešnom redoslijedu. Nakon toga možemo porediti $k-1$ -vi i $k-2$ -gi element (uz eventualnu razmjenu u slučaju potrebe), zatim $k-2$ -gi i $k-3$ -ći element, itd. Onog trenutka kada ustanovimo da su dva susjedna elementa u ispravnoj poziciji, nema potrebe da nastavljamo dalje prema početku niza, s obzirom da su svi elementi do $k-1$ -vog bili sortirani, tako da je k -ti element sigurno došao na poziciju na kojoj treba da bude. Opisani postupak predstavlja jedan prolaz sortiranja umetanjem. Kompletno sortiranje postiže se ponavljanjem ovog postupka za vrijednosti k redom od 2 do n (odnosno od 1 do $n-1$ ukoliko indeksi niza počinju od nula). Sljedeća slika ilustrira tok sortiranja ovim postupkom na primjeru istog niza kao u ranijim primjerima. Element kojem tražimo pravo mjesto označen je podebljano, dok su elementi koje treba razmijeniti označeni podvučeno:

3	5	1	8	6	9	5	7
3	5	1	8	6	9	5	7
3	<u>5</u>	1	8	6	9	5	7
3	<u>1</u>	5	8	6	9	5	7
1	3	5	8	6	9	5	7
1	3	5	8	6	9	5	7
1	3	5	<u>8</u>	6	9	5	7
1	3	5	<u>6</u>	8	9	5	7
1	3	5	6	8	9	5	7
1	3	5	6	8	<u>9</u>	5	7
1	3	5	6	8	<u>5</u>	9	7
1	3	5	6	<u>5</u>	8	9	7
1	3	5	5	6	8	9	7
1	3	5	5	6	8	<u>9</u>	7
1	3	5	5	6	<u>8</u>	7	9
1	3	5	5	6	<u>7</u>	8	9

Na osnovu opisanog postupka slijedi sljedeća implementacija funkcije koja realizira sortiranje umetanjem, koja je očigledno jednostavna gotovo kao i funkcija “KlasicniSort2”;

```
template <typename UporediviTip>
void SortUmetanjem(UporediviTip niz[], int n) {
    for(int k = 1; k < n; k++) {
        for(int i = k; i > 0 && niz[i - 1] > niz[i]; i--)
            Razmjeni(niz[i], niz[i - 1]);
    }
}
```

Sortiranje umetanjem može se još ubrzati ukoliko primijetimo da u traženju mjesta za k -ti element ne moramo svaki put vršiti razmjenu elementa sa susjednim. Naime, dovoljno je *zapamtiti* vrijednost k -tog elementa u neku pomoćnu promjenljivu, a nakon toga prilikom traženja njegove prave pozicije prosto *prebacivati* elemente koji nisu na svom mjestu za jedno mjesto naviše (umjesto obavljanja *razmjene*). Na kraju, kada nademo poziciju na kojoj bi se razmatrani element trebao smjestiti, prosto na tu poziciju upišemo zapamćenu vrijednost. Tako smo broj izvršenih prebacivanja između promjenljivih smanjili skoro za dvije trećine, što nije zanemarljivo. Ova ideja dovodi do sljedeće izvedbe sortiranja umetanjem:

```
template <typename UporediviTip>
void SortUmetanjem2(UporediviTip niz[], int n) {
    for(int k = 1; k < n; k++) {
        UporediviTip v = niz[k];
        int i;
```

```

        for(i = k; i > 0 && niz[i - 1] > v; i--) niz[i] = niz[i - 1];
        niz[i] = v;
    }
}

```

Mada sortiranje umetanjem u najgorem slučaju radi podjednako loše kao i mjeđurasto sortiranje, postoji mnogo slučajeva u kojima sortiranje umetanjem radi veoma dobro. Recimo, nije teško vidjeti da sortiranje umetanjem, poput mjeđurastog sortiranja, također završava veoma brzo za slučaj kada se pokuša sortirati već sortiran niz. Dalje, ukoliko smo imali niz koji je već bio sortiran, i na njegov kraj dopisali nekoliko elemenata koji kvare sortirani poredak, sortiranje umetanjem će tako formirati niz sortirati vjerovatno brže nego i jedan drugi poznati metod, čak i ukoliko niz ima mnogo elemenata. Stoga je ovaj metod jako praktičan kada želimo već sortirani niz dopunjavati novim podacima, uz očuvanje prethodno ostvarene sortiranosti. Detaljna analiza pokazuje da je trajanje sortiranja umetanjem proporcionalna *broju inverzija* u nizu, pri čemu se pod inverzijom podrazumijeva svaki par elemenata (ne nužno susjednih) koji nisu u ispravnom poretku jedan u odnosu na drugi. Slijedi da je sortiranje umetanjem efikasno za sve nizove u kojima je broj inverzija mali. Ipak, za nasumično formirane nizove, sortiranje umetanjem također ima kvadratnu složenost. Prosječna složenost ovog metoda je također kvadratna (iako je u prosjeku najbrži od do sada razmotrenih metoda).

Ovim smo završili razmatranje jednostavnih metoda za sortiranje. Od svih razmotrenih metoda, u praksi se koriste samo klasična varijanta sortiranja izborom (funkcija "KlasicniSort"), koja je pogodna za sortiranje nizova sa malim brojem masivnih elemenata, i poboljšana izvedba sortiranja umetanjem (funkcija "SortUmetanjem2"), koja je pogodna za skoro sortirane nizove (tj. nizove sa malim brojem inverzija). Ostali metodi navedeni su više kao ilustracija tehnika programiranja nego kao praktično upotrebljivi metodi sortiranja.

Sve napisane izvedbe funkcija za sortiranje prihvatale su kao parametre niz koji se sortira i broj elemenata niza. Kao što smo već govorili u poglavlju o pokazivačima, moderni trendovi u programiranju ukazuju da je bolje pisati funkcije koje kao parametre prihvataju *pokazivače* koji omeđavaju dio niza na koji se tražena operacija nad nizom (npr. sortiranje) odnosi. Prednost takvog pristupa je u činjenici da se tako napisane funkcije lako generaliziraju na druge kontejnerske strukture podataka koje nisu nizovi (npr. na *liste*, sa kojima ćemo se upoznati kasnije). Pored toga, tako napisane funkcije često mogu biti i efikasnije od klasičnih izvedbi (ova dva razloga su dovoljna da gotovo sve funkcije iz biblioteke "algorithm", kao što smo već vidjeli, koriste ovakvu konvenciju o parametrima). Nije teško do sada napisane funkcije prepraviti da koriste takvu konvenciju o parametrima. Slijede verzije do sada napisanih funkcija koje kao parametre primaju pokazivač na početak dijela niza koji se sortira, i pokazivač iza kraja dijela niza koji se sortira (poput parametara koji koriste funkcije iz biblioteke "algorithm"). Analiza ovih funkcija je korisna za savladavanje pokazivačke aritmetike:

```

template <typename UporediviTip>
void KlasicniSort(UporediviTip *pocetak, UporediviTip *iza_kraja) {
    for(; pocetak < iza_kraja - 1; pocetak++) {
        UporediviTip *gdje_je_min = pocetak;
        for(UporediviTip *p = pocetak + 1; p < iza_kraja; p++)
            if(*p < *gdje_je_min) gdje_je_min = p;
        if(gdje_je_min != pocetak) Razmijeni(*pocetak, *gdje_je_min);
    }
}

template <typename UporediviTip>
void KlasicniSort2(UporediviTip *pocetak, UporediviTip *iza_kraja) {
    for(; pocetak < iza_kraja - 1; pocetak++)
        for(UporediviTip *p = pocetak + 1; p < iza_kraja; p++)
            if(*p < *pocetak) Razmijeni(*p, *pocetak);
}

template <typename UporediviTip>
void BubbleSort(UporediviTip *pocetak, UporediviTip *iza_kraja) {

```

```

bool razmijenjen(true);
for(int k = 1; k < iza_kraja - pocetak && razmijenjen; k++) {
    razmijenjen = false;
    for(UporediviTip *p = pocetak; p < iza_kraja - k; p++)
        if(*p > *(p + 1)) {
            Razmijeni(*p, *(p + 1));
            razmijenjen = true;
        }
}
}

template <typename UporediviTip>
void SortUmetanjem(UporediviTip *pocetak, UporediviTip *iza_kraja) {
    for(UporediviTip *p1 = pocetak + 1; p1 < iza_kraja; p1++)
        for(UporediviTip *p2 = p1; p2 > pocetak && *(p2 - 1) > *p2; p2--)
            Razmijeni(*p2, *(p2 - 1));
}

template <typename UporediviTip>
void SortUmetanjem2(UporediviTip *pocetak, UporediviTip *iza_kraja) {
    for(UporediviTip *p1 = pocetak + 1; p1 < iza_kraja; p1++) {
        UporediviTip v = *p1, *p2 = p1;
        for(; p2 > pocetak && *(p2 - 1) > v; p2--) *p2 = *(p2 - 1);
        *p2 = v;
    }
}
}

```

Primijetimo da su se konstrukcije poput “ $*(p + 1)$ ” i “ $*(p - 1)$ ”, kojima dohvatomo *sljedeći* odnosno *prethodni* element u odnosu na element na koji pokazuje pokazivač “ p ”, načelno mogle zamijeniti sa “ $p[1]$ ” odnosno “ $p[-1]$ ”. Međutim, s obzirom da “ p ” nije niz nego klizeći pokazivač, ovakva notacija bi vjerovatno više zbumjivala nego što bi koristila. Inače, između više međusobno ekvivalentnih konstrukcija, programer se uvijek treba da odluči za onu koja najjasnije iskazuje namjeru programera.

Svi do sada razmotreni postupci za sortiranje imali su, kako u najgorem, tako i u prosječnom slučaju, kvadratnu složenost. Stoga je prirodno postaviti pitanje da li je uopće moguće proizvoljan niz sortirati brže, odnosno da li je moguće u najgorem ili prosječnom slučaju imati složenost koja je garantirano bolja od kvadratne. Odgovor je potvrđan. Postoji cijeli niz postupaka nazvanih *brzi postupci za sortiranje* kod kojih je vrijeme potrebno za sortiranje niza od n elemenata proporcionalno sa $n \log n$ umjesto sa n^2 , što je veliki napredak (npr. za $n=10000$ imamo $n^2=100000000$ dok je $n \log n=40000$). Najpoznatiji brzi postupci za sortiranje su *Shellovo sortiranje* (engl. *Shell sort*), *sortiranje spajanjem* (engl. *merge sort*), *sortiranje slaganjem na hrpu* (engl. *heap sort*), *sortiranje razdvajanjem* (engl. *partition sort*), *sortiranje pomoću steka* (engl. *stack sort*), *Ford-Johnsonovo sortiranje*, *sortiranje višestrukim razvrstavanjem* (engl. *radix sort*), itd. Svaki od ovih postupaka ima mnoge podvarijante.

Većina brzih postupaka za sortiranje su dosta komplikovani i često zahtijevaju pomoćne nizove za smještanje raznih međurezultata. Također, veliki broj brzih postupaka za sortiranje su *rekurzivni*. Vjerovatno najbolji kompromis između brzine i složenosti ostvaren je pri sortiranju razdvajanjem. Ovaj postupak otkrio je C. Hoare 1960. godine i predstavlja najpoznatiji brzi postupak za sortiranje. Mnoge analize pokazuju da je, u prosjeku gledano, ovo najbrži poznati postupak za sortiranje (mada postoje situacije u kojima drugi postupci rade brže). Stoga se on često naziva prosto *brzo sortiranje* (engl. *quick sort*). Na ovom mjestu nećemo ulaziti u razmatranje raznih brzih postupaka za sortiranje, već ćemo se zaustaviti samo na opisu *quick sort* postupka, kao najpoznatijeg brzog postupka za sortiranje.

Postoji mnogo podvarijanti ovog postupka, a sve se zasnivaju na istoj ideji da se niz podijeli na dva približno jednaka dijela takva da je svaki element iz prvog dijela manji od svakog elementa iz drugog dijela. To se može uraditi tako što se prvo nasumice izabere jedan element niza koji ćemo nazvati *pivot*.

Nakon toga, koristimo dva pokazivača (zapravo indeksa), koja ćemo nazvati *lijevi pokazivač* i *desni pokazivač*. Lijevi pokazivač postavljamo na prvi element niza, nakon čega tražimo prvi element koji je veći ili jednak pivotu, krećući se *nadesno* (tj. ka većim indeksima). Desni pokazivač postavljamo na posljednji element niza, nakon čega tražimo prvi element koji je manji ili jednak pivotu, krećući se *nalijevo* (tj. ka manjim indeksima). Kada pronađemo takve elemente, razmijenimo ih (pri tome se može desiti da pivot također promjeni svoju poziciju). Nakon toga nastavljamo isti postupak sve dok lijevi pokazivač ne postane veći od desnog. Nije teško uvidjeti da je nakon toga niz razdvojen na dva dijela sa traženim svojstvima, pri čemu su granice dijelova određene upravo trenutnim pozicijama pokazivača. Ovaj postupak ilustriran je na primjeru sa sljedeće slike. Pivot je označen podebljano, dok su pozicije lijevog i desnog pokazivača označeni sa jednom odnosno dvije crte respektivno. Radi jednostavnosti, prikazane su samo pozicije pokazivača u trenucima neposredno prije nego što će se izvršiti razmjena.

2	9	5	4	1	7	6	1	7	3	2	5	1	4	8
2	4	5	4	1	7	6	1	7	3	2	5	1	9	8
2	4	5	4	1	1	6	1	7	3	2	5	7	9	8
2	4	5	4	1	1	5	1	7	3	2	6	7	9	8
2	4	5	4	1	1	5	1	<u>2</u>	<u>3</u>	<u>7</u>	6	7	9	8

1. dio

2. dio

Ovim smo očigledno niz podijelili na dva dijela sa traženim svojstvima. Sortiranje se sada nastavlja tako što se isti postupak *rekurzivno* primjenjuje na oba dijela. Pri tome se rekursija obavlja sve dok se ne dođe do dijelova koji sadrže samo jedan ili nijedan element, jer tada nemamo šta raditi. Ukoliko usvojimo da za pivot uvijek biramo element koji se nalazi *u sredini*, tada bi implementacija funkcije koja obavlja postupak brzog sortiranja mogla izgledati ovako:

```
template <typename UporediviTip>
void QuickSort(UporediviTip niz[], int odakle, int dokle) {
    if(odakle >= dokle) return;
    UporediviTip pivot = niz[(odakle + dokle) / 2];
    int lpok(odakle), dpok(dokle);
    while(lpok <= dpok) {
        while(niz[lpok] < pivot) lpok++;
        while(niz[dpok] > pivot) dpok--;
        if(lpok <= dpok) {
            Razmjeni(niz[lpok], niz[dpok]);
            lpok++; dpok--;
        }
    }
    QuickSort(niz, odakle, dpok); QuickSort(niz, lpok, dokle);
}
```

Primijetimo da se ova funkcija poziva drugačije nego dosadašnje funkcije za sortiranje. Naime, umjesto broja elemenata niza funkciji se kao parametri prosljeđuju indeks početnog i krajnjeg elementa niza koji se sortira. Ovakva konvencija je neophodna da bi se funkcija mogla rekursivno pozivati nad dijelovima niza. Zapravo, moguće je prepraviti funkciju "QuickSort" tako da prihvata samo niz i broj elemenata kao parametre, ukoliko iskoristimo činjenicu da su formalni parametri nizovnog tipa zapravo pokazivači, kao i činjenicu da se nad imenima nizova može koristiti pokazivačka aritmetika. U nastavku je prikazana jedna od mogućih modifikacija takve vrste. Analiza ove modifikacije ostavlja se čitateljima odnosno čitateljkama za vježbu.

```
template <typename UporediviTip>
void QuickSort(UporediviTip niz[], int n) {
    if(n <= 1) return;
```

```

UporediviTip pivot = niz[n / 2];
int lpok(0), dpok(n - 1);
while(lpok <= dpok) {
    while(niz[lpok] < pivot) lpok++;
    while(niz[dpok] > pivot) dpok--;
    if(lpok <= dpok) {
        Razmjeni(niz[lpok], niz[dpok]);
        lpok++; dpok--;
    }
}
QuickSort(niz, dpok + 1); QuickSort(niz + lpok, n - lpok);
}

```

Naravno, i ova funkcija se može napraviti tako da prima kao parametre pokazivače. Dobijena varijanta je možda i najprirodnija, jer su u njoj "lpok" i "dpok" zaista *pokazivači*, a ne indeksi, a rekurzivni pozivi se izvode na prirodan način. Slijedi i varijanta funkcije "QuickSort" prilagodena ovakvoj konvenciji prenosa parametara:

```

void QuickSort(UporediviTip *pocetak, UporediviTip *iza_kraja) {
    if(pocetak + 1 >= iza_kraja) return;
    UporediviTip pivot = *(pocetak + (iza_kraja - pocetak) / 2);
    UporediviTip *lpok = pocetak, *dpok = iza_kraja - 1;
    while(lpok <= dpok) {
        while(*lpok < pivot) lpok++;
        while(*dpok > pivot) dpok--;
        if(lpok <= dpok) {
            Razmjeni(*lpok, *dpok);
            lpok++; dpok--;
        }
    }
    QuickSort(pocetak, dpok + 1); QuickSort(lpok, iza_kraja);
}

```

U ovoj funkciji je interesantno razmotriti kako je izvedeno uzimanje srednjeg elementa za pivot. Izraz koji je tom prilikom upotrijebljen ima formu " $a + (b - a)/2$ " koji je, na prvi pogled, ekvivalentan izrazu " $(a + b)/2$ ". Međutim, to je tačno samo kada su a i b brojevi. U slučaju kada su a i b pokazivači, prvi izraz je dobro definiran, a drugi nije (s obzirom da se dva pokazivača ne mogu sabirati).

Opisani postupak u većini slučajeva radi zaista veoma brzo. Međutim, može se pokazati da je ovaj postupak ponekad dosta osjetljiv na izbor elementa koji se uzima za pivot. Na primjer, često se susreću implementacije u kojima se za pivot uzima *prvi* element u nizu, ali takav izbor je dobar samo za dobro izmešane nizove. Lako je pokazati da ovakav izbor pivota u slučaju da je niz potpuno sortiran ili skoro sortiran vodi također ka vremenu izvršavanja koje je proporcionalno kvadratu broja elemenata niza (što je još gore, u takvim slučajevima je broj rekurzivnih poziva proporcionalan broju elemenata niza, tako da u slučaju velikih nizova može doći do zagušenja mašinskog steka). Ovo se ne dešava kada se za pivot uzima element iz sredine niza, kao što je urađeno u prikazanoj izvedbi. Međutim, i za takvu izvedbu moguće je konstruisati nizove za koje će prikazani algoritam raditi sporo.

Da bi se izbjegao ovaj problem, uvedene su različite modifikacije algoritma. Jedna od poznatijih modifikacija poznata je pod nazivom *medijan-3*, u kojoj se za pivot bira *srednji po veličini između prvog, posljednjeg i srednjeg elementa niza*. Ipak, možda se najbolji rezultati postižu tako što se pri svakom rekurzivnom pozivu pivot izabere kao *posve slučajni element niza*. Na taj način je vjerovatnoća da će se u svakom rekurzivnom pozivu izabrati baš nepovoljan pivot svedena na posve zanemarljiv minimum, tako da je praksa pokazala da tako formirana verzija brzog sortiranja radi brzo u praktično svim slučajevima. Ostaje pitanje kako izvršiti slučajan izbor elementa. Za tu svrhu jezik C++ posjeduje izvjesne funkcije

koje kao rezultat vraćaju *slučajan broj*. Preciznije, računar ne može bukvalno dati slučajan broj kao rezultat, ali je rezultat za čovjeka nepredvidljiv u toj mjeri da se može smatrati slučajnim. Jedna takva funkcija je funkcija "rand" iz biblioteke "cstdlib" (zaglavje ove biblioteke u jeziku C i starijim verzijama kompjajlera za C++ zove se "stdlib.h"). Ova funkcija daje kao rezultat slučajan (bolje rečeno nepredvidljiv) broj u opsegu od 0 do neke velike vrijednosti koja je definirana konstantom "RAND_MAX". Svaki put kada se funkcija "rand" pozove, rezultat može biti drugačiji, tako da je u tom smislu vrijednost koju ova funkcija daje slučajna, tim prije što je nemoguće (ili barem veoma teško) predvidjeti koju će vrijednost funkcija "rand" vratiti. Za primjenu bi bila pogodnija funkcija koja će vratiti slučajan broj *iz zadanog opsega*. Srećom, takvu funkciju je lako napraviti pomoću funkcije "rand". Slijedi jedna verzija funkcije "Slučajni" koja kao rezultat vraća slučajni cijeli broj u opsegu od *a* do *b* uključivo, pri čemu se *a* i *b* zadaju kao parametri:

```
int Slučajni(int a, int b) {
    return a + rand() % (b - a + 1);
}
```

Princip rada ove funkcije je očigledan ukoliko primijetimo da je ostatak dijeljenja ma kojeg broja sa brojem *n* uvijek u opsegu od 0 do *n*-1. Međutim, zbog načina na koji interno radi većina algoritama za generiranje slučajnih brojeva, može se desiti da sekvenca brojeva koju vraća funkcija "Slučajni" pokazuje priličnu pravilnost u slučaju kada izraz *b-a+1* predstavlja stepen broja 2 (ovo je posljedica nekih osobina binarnog zapisa brojeva). Statističke analize pokazuju da sljedeća, komplikovanija verzija funkcije "Slučajni", tipično daje nepravilniju sekvensu brojeva, što je poželjno ukoliko želimo generiranu sekvensu brojeva zaista smatrati slučajnom:

```
int Slučajni(int a, int b) {
    return a + int((b - a + 1) * double(rand()) / (RAND_MAX + 1));
}
```

Kada smo se upoznali sa funkcijama za generiranje slučajnih brojeva, možemo pokazati i kako se može napraviti brzo sortiranje sa slučajnim izborom pivota (engl. *randomized quicksort*). Za tu svrhu, dovoljno je samo promijeniti naredbu koja definira vrijednost pivota. Preciznije, naredbe za izbor pivota koje u tri prethodno napisane verzije funkcije "Quicksort" izgledaju respektivno ovako:

```
UporediviTip pivot = niz[(odakle + dokle) / 2];
UporediviTip pivot = niz[n / 2];
UporediviTip pivot = *(pocetak + (iza_kraja - pocetak) / 2);
```

treba respektivno zamijeniti sa jednom od sljedećih naredbi:

```
UporediviTip pivot = niz[Slučajni(odakle, dokle)];
UporediviTip pivot = niz[Slučajni(0, n - 1)];
UporediviTip pivot = *(pocetak + Slučajni(0, iza_kraja - pocetak - 1));
```

Princip rada izvršenih izmjena je očigledan. Inače, algoritmi čiji se rad zasniva na upotrebi slučajnih brojeva nazivaju se *randomizirani algoritmi*. Nije rijedak slučaj da randomizirani algoritmi u prosjeku rade mnogo brže nego odgovarajući algoritmi koji ne koriste slučajne brojeve.

Postupak "quick sort" je, općenito gledano, jedan od najbržih poznatih postupaka za sortiranje. Ipak, treba napomenuti da je ovaj postupak namijenjen za sortiranje velikih nizova, tako da za nizove sa malim brojem elemenata sortiranje umetanjem može biti efikasnije. Ista primjedba vrijedi i za skoro sortirane nizove. Uz izvjesne modifikacije, moguće je postići još neka ubrzanja postupka "quick sort", ali ne bitna. Pored toga, takve modifikacije znatno usložnjavaju samu funkciju. Na primjer, često korištena

modifikacija je da se rekurzivni postupak ne provodi do kraja, tj. do nizova koji imaju samo jedan element, nego se čim broj elemenata niza koji se rekurzivno sortira opadne ispod određene granice (npr. 30) postupak dalje nastavlja primjenom sortiranja umetanjem (koje je efikasnije za male nizove).

Treba reći da su poznate brojne varijante i modifikacije ovog metoda, tako da se u literaturi susreće veoma mnogo različitih implementacija algoritma pod istim imenom "quick sort". Nažalost, u literaturi se nerijetko mogu susresti i *neispravne implementacije*, koje u izvjesnim rijetkim slučajevima ne rade korektno. Naime, mada je osnovna ideja ovog postupka prilično jednostavna, prilikom same izvedbe razdvajanja niza lako je napraviti izvjesne propuste koji vode ka programima koji naizgled rade, i koji zaista rade ispravno u većini slučajeva, iako ne rade korektno u nekim specijalnim slučajevima. Postupak "quick sort" je veoma osjetljiv u smislu da je lako napraviti grešku koja vodi ka programima koji rade skoro uvijek, ali ne i uvijek (za razliku od većine drugih algoritama kod kojih greške obično dovode do programa koji *uopće ne rade*, što olakšava uočavanje grešaka). Stoga je najbolje sa algoritmom "quick sort" mnogo ne eksperimentisati, nego koristiti onu varijantu za koju je provjereno da uvijek radi ispravno, i to bez improvizacija i nepotrebnih modifikacija.

Na kraju priče o sortiranju, treba ukazati na još jednu činjenicu. Metod "quick sort" je očigledno *rekurzivan*, što vrijedi i za većinu drugih brzih postupaka za sortiranje. Stoga je prirodno očekivati da bi nerekurzivna verzija ovog metoda trebala da bude još brža, s obzirom da smo ranije vidjeli da su nerekurzivna rješenja uvijek efikasnija od rekurzivnih. Mada je ovo tačno i za ovaj metod, "quick sort" je tipičan primjer algoritma za koji se previše *ne isplati* tražiti nerekurzivno rješenje. Naime, nerekurzivni "quick sort" je sasvim neznatno brži od rekurzivnog, a oko 3 puta je složeniji, i zahtijeva pomoćne nizove (koji u rekurzivnoj verziji nisu potrebni). Možda nekome izgleda čudno to što ovaj metod radi brzo usprkos činjenici da funkcija "Quicksort" rekurzivno poziva samu sebe na dva mesta, što je u slučaju problema Hanojskih kula i računanja Fibonačijevih brojeva vodilo ka eksponencijalnom trajanju izvršavanja. Odgovor na ovu zagonetku leži u tome da se u svakom rekurzivnom pozivu funkcije "Quicksort" veličina dijela niza koji treba sortirati u prosjeku smanjuje na oko polovinu, tako da veličina niza opada eksponencijalnom brzinom, za razliku od Hanojskih kula i Fibonačijevih brojeva u kojima se veličina problema koji se rješava u rekurzivnim pozivima smanjivala za konstantu (1 ili 2). Stoga se kod funkcije "Quicksort" eksponencijalno razgranavanje koje uzrokuje dvostruka rekurzija i eksponencijalno smanjenje veličine problema u svakom rekurzivnom pozivu međusobno kompenziraju, tako da na kraju ipak dobijamo rješenje koje radi efikasno. Generalno, rekurzivni algoritmi kod kojih se veličina problema koji se rješava smanjuje eksponencijalnom brzinom u svakom rekurzivnom pozivu mogu biti veoma efikasni. Ovaj primjer još jedanput ukazuje da rekurzija "nije igračka za svakoga", i da se analiza efikasnosti rekurzivnih algoritama često ne može odrediti neposrednim posmatranjem, već samo nakon detaljne analize koja ponekad zahtijeva i primjenu specijalnog matematskog aparata koji se obično izučava u okviru teorije algoritama.

Funkcije za generiranje slučajnih brojeva mogu lijepo poslužiti i za testiranje raznih postupaka koji se obavljuju nad nizovima, kao što je upravo sortiranje. Naime, ukoliko želimo da ispitamo efikasnost nekog postupka za sortiranje na nizu od recimo 10000 elemenata, najprirodnije je da elemente niza prije nego što pozovemo funkciju za sortiranje napunimo *slučajnim vrijednostima*, što je pomoću opisanih funkcija veoma lako učiniti. Međutim, ukoliko probate da napunite neki niz slučajnim vrijednostima a zatim da ispištete te vrijednosti na ekran, primjetićete da su te vrijednosti, mada potpuno nasumične, *iste svaki put kada ponovo pokrenete program!* Razlog za tu činjenicu leži u tome da funkcija "rand" (pa prema tome i funkcija "slučajni", koja je izvedena iz nje) zapravo pri svakom pozivu vraća kao rezultat sljedeći član jedne unaprijed fiksirane i tačno određene sekvence brojeva, koja je tako formirana da *izgleda* kao da njeni elementi nemaju nikakve međusobne veze. Ipak, ta sekvenca je *ista* svaki put kada pokrenete program. Ponekad ovakvo ponašanje nije poželjno, tako da C++ posjeduje i funkciju "srand" (također u biblioteci "cstdlib") kojoj se kao parametar proslijeđuje cijeli broj, tzv. "sjeme" (engl. *seed*) pomoću kojeg se vrši izbor sekvence koju generira funkcija rand (funkcija "srand" ne vraća nikakvu vrijednost). Ukoliko želimo da se svaki put kada se program pokrene generira nova sekvenca, potrebno je na početku programa

funkciji "srand" kao argument proslijediti neku vrijednost koja se je drugačija pri svakom pokretanju programa. Jedna mogućnost je iskoristiti rezultat funkcije "time" iz biblioteke "ctime" (zaglavlje ove biblioteke zove se "time.h" u jeziku C i starijim verzijama kompjerala za C++) koja kada joj se proslijedi 0 kao argument kao rezultat vraća broj sekundi koji je protekao od ponoći 1. januara 1970. godine do trenutka poziva funkcije (jasno je da je taj rezultat različit pri svakom pokretanju programa). Dakle, naredba oblika

```
srand(time(0));
```

ubačena na početak programa uzrokovaje generiranje nove sekvence slučajnih brojeva pri svakom pokretanju programa.

U praktičnom programiranju u jeziku C++, funkcije za sortiranje nije potrebno posebno programirati, s obzirom da standardna biblioteka "algorithm" posjeduje funkciju "sort", koja izvodi sortiranje niza. Parametri ove funkcije su pokazivači koji omeđuju dio niza koji treba sortirati (slično kao u srodnim verzijama funkcija za sortiranje koje smo samostalno pisali). Pored funkcije "sort", ova biblioteka sadrži i neke njoj srodne funkcije, od kojih su najvažnije prikazane u sljedećoj tabeli:

<code>sort(p1, p2)</code>	Sortira elemente između pokazivača $p1$ i $p2$ u rastući poredk (tj. u poredk definiran operatorom " $<$ ").
<code>stable_sort(p1, p2)</code>	Radi isto kao i funkcija "sort", ali zadržava one elemente koji su sa aspekta kriterija sortiranja ekvivalentni u istom poretku u kakvom su bili prije sortiranja. O ovome ćemo detaljnije govoriti kasnije.
<code>partial_sort(p1, p2, p3)</code>	Obavlja djelimično sortiranje elemenata između pokazivača $p1$ i $p3$ tako da na kraju elementi između pokazivača $p1$ i $p2$ budu u rastućem poretku, dok preostali elementi ne moraju biti ni u kakvom određenom poretku.
<code>nth_element(p1, p2, p3)</code>	Obavlja djelimično sortiranje elemenata između pokazivača $p1$ i $p3$ tako da na kraju element na koji pokazuje pokazivač $p2$ bude onakav kakav bi bio u potpuno sortiranom nizu (u rastućem poretku). Ostali elementi ne moraju pri tom biti sortirani, ali se garantira da će svi elementi ispred odnosno iza elementa na koji pokazuje pokazivač $p2$ biti manji odnosno veći od njega.
<code>partial_sort_copy(p1, p2, p3, p4)</code>	Kopira elemente između pokazivača $p1$ i $p2$ u rastućem (sortiranom) poretku u dio memorije koji se nalazi između pokazivača $p3$ i $p4$. Kopiranje prestaje ili kada se odredišni blok popuni, ili kada se izvorni blok iscrpi. U svakom slučaju, funkcija kao rezultat vraća pokazivač na mjesto iza posljednjeg kopiranog elementa. Elementi između $p1$ i $p2$ ostaju netaknuti.

Funkcije poput "partial_sort" i "nth_element" su korisne ukoliko nam ne treba sortiranje cijelog niza, već samo recimo prvih 10 elemenata sortiranog niza (npr. ukoliko pravimo rang liste u kojima nas zanima samo prvih 10 najboljih rezultata a ostali nas ne zanimaju), ili ukoliko nas zanima samo element koji će se u sortiranom nizu naći na određenoj poziciji. Naravno, u oba slučaja možemo

koristiti i funkciju “`sort`”, ali funkcije “`partial_sort`” i “`nth_element`” tipično rade mnogo brže. Inače, standard jezika C++ ne propisuje koji algoritam za sortiranje koriste opisane funkcije. S druge strane, pošto je cilj ovih funkcija da budu što brže, u svakom slučaju se koristi neki dobar algoritam. Većina implementacija funkciju “`sort`” izvodi kombinacijom “quick sort” algoritma (varijanta medijan-3) i sortiranja umetanjem (pri čemu se sortiranje umetanjem koristi za slučaj manjih ili skoro sortiranih nizova), tako da će funkcija “`sort`” gotovo sigurno raditi brže od većine funkcija za sortiranje koje bismo eventualno sami napisali.

Funkcije poput “`sort`” podrazumijevano obavljaju sortiranje u *rastući poredak*. Ponekad je potrebno sortirati niz u *opadajući* ili neki drugi poredak. Sve napisane funkcije za sortiranje je lako prepraviti tako da obavljaju sortiranje u opadajući poredak (naime dovoljno je međusobno zamijeniti uloge relacionih operatora “`<`” i “`>`”). Da bi se omogućilo korištenje funkcija poput “`sort`” za sortiranje u opadajućem ili nekom drugom poretku, ovim funkcije primaju kao opcionalni posljednji parametar pokazivač na funkciju kriterija koja treba da definira kriterij poređenja. Ova funkcija prima dva parametra, koji predstavljaju dva elementa koji se porede (njihov tip treba odgovarati tipu elemenata niza koji se sortira), i koja treba da vrati logičku vrijednost “`true`” ili “`false`”, ovisno od toga da li su elementi koji se porede u ispravnom poretku ili ne. Na primjer, ukoliko želimo sortirati neki niz realnih brojeva u opadajući poredak primjenom funkcije “`sort`”, prvo ćemo definirati funkciju kriterija poput sljedeće:

```
bool veci(double a, double b) {
    return a > b;
}
```

Zatim ćemo, ukoliko je “niz” niz od 1000 elemenata koji želimo sortirati, upotrijebiti sljedeći poziv:

```
sort(niz, niz + 1000, veci);
```

Nije teško shvatiti kako je ova mogućnost implementirana. Naime, u implementaciji funkcije “`sort`”, sva poređenja tipa “`if(x < y)`” zamijenjena su sa “`if(dobar_poredak(x, y))`” gdje je “`dobar_poredak`” ime parametra koji predstavlja pokazivač na funkciju kriterija.

Zahvaljujući mogućnosti zadavanja funkcije kriterija, moguće je sortirati i nizove za čiji element nije definiran standardni poredak (npr. nizove kompleksnih brojeva), ukoliko samostalno definiramo kriterij porekta. Na primjer, niz kompleksnih brojeva možemo sortirati u rastući poredak po absolutnoj vrijednosti, ukoliko zadamo funkciju kriterija poput sljedeće:

```
bool manji_po_apsolutnoj(complex<double> a, complex<double> b) {
    return abs(a) < abs(b);
}
```

Funkcija kriterija je uvijek potrebna kad želimo sortirati nizove za čije elemente nije definiran podrazumijevani poredak. Na primjer, u kasnijim poglavljima ćemo vidjeti da je moguće formirati tipove podataka koji predstavljaju skupinu informacija o jednom studentu. Jasno je da između takvih podataka ne postoji jedinstveno definiran poredak. U tom slučaju funkcija kriterija je neophodna da precizira smisao sortiranja. Na primjer, možemo zahtijevati da niz podataka o studentima sortiramo u opadajući redoslijed po prosječnoj ocjeni studija, ili u rastući redoslijed po broju indeksa, ili u rastući abecedni poredak po imenu i prezimenu, itd. O ovome ćemo detaljnije govoriti kada se upoznamo sa strukturnim tipovima podataka.

objasniti `stable_sort`

Postoje situacije kada treba sortirati nizove elemenata za koje je formalno definiran poredak pomoću

operatora “<”, ali u kojima primjena tog operatora ne daje očekivani rezultat. U takvim slučajevima je također neophodno definirati funkciju kriterija. Na primjer, tipična situacija nastaje ukoliko želimo sortirati niz pokazivača na znakove. Na primjer, zamislimo da imamo niz poput sljedećeg:

```
char *imena[5] = { "Kemo", "Stevo", "Ivo", "Amira", "Meho"};
```

Ukoliko bismo pokušali da sortiramo ovaj niz (u abecednom poretku) prostim pozivom

```
sort(imena, imena + 5);
```

rezultati ne bi bili u skladu sa očekivanjima. Naime, elementi niza “imena” nisu sama imena, nego *pokazivači* na mesta u memoriji gdje su ova imena zapisana. Prilikom poziva funkcije “sort”, ona bi za poređenje elemenata niza koristila relaciju “<”. Međutim, kako su elementi ovog niza pokazivači, relacija “<” će zapravo porediti pokazivače (odnosno *adrese*). Slijedi da će prikazani poziv funkcije “sort” izvršiti sortiranje u rastući poredak *po adresama na kojima su navedena imena smještena u memoriji*, a ne po samim imenima, što sigurno nije ono što smo željeli. Mi zapravo ne želimo porediti same pokazivače, već *nizove znakova na koje oni pokazuju*, što se može obaviti pomoću funkcije “strcmp”. Slijedi da je potrebno definirati funkciju kriterija poput sljedeće:

```
bool ispred_po_abecedi(char *a, char *b) {
    return strcmp(a, b) < 0;
}
```

Sada bi poziv poput

```
sort(imena, imena + 5, ispred_po_abecedi);
```

obavio željeno sortiranje imena u abecedni poredak. Pri tome je bitno shvatiti da se tom prilikom neće izmijeniti poredak u kojem su imena zaista smještena u memoriji. Izmjeniće se samo poredak pokazivača koji na njih pokazuju, o čemu smo ukratko govorili u prethodnom poglavlju.

Treba napomenuti da funkcija kriterija ne bi bila potrebna da smo umjesto nizova pokazivača na znakove koristili niz dinamičkih stringova (tj. niz elemenata tipa “string”), s obzirom da je za objekte tipa “string” relacioni operator “<” definiran upravo tako da vrši poređenje po abecednom poretku. Ipak, treba naglasiti da je sortiranje nizova pokazivača na znakove znatno efikasnije nego sortiranje nizova dinamičkih stringova. Naime, sa aspekta efikasnosti, znatno je jednostavnije prosto razmijeniti dva pokazivača, nego razmijeniti čitava dva stringa u memoriji.

Funkcije za sortiranje iz biblioteke “algorithm” podrazumijevaju da je za elemente niza koji se sortiraju definirano pridruživanje pomoću operatora “=”, odnosno da se razmjena dva elementa može izvršiti pomoću funkcije “swap” (koja je zapravo identična funkciji “Razmijeni” koju smo koristili). Ova osobina je zaista ispunjena za većinu tipova koji se u jeziku C++ susreću. Međutim, ovo ograničenje onemogućava korištenje funkcije “sort” za sortiranje nizova čiji su elementi recimo nul-terminalirani nizovi znakova (tj. za sortiranje dvodimenzionalnih nizova znakova), s obzirom da su njihovi elementi ponovo *nizovi*, a za nizove nije definirano pridruživanje pomoću operatora “=”. Stoga, sortiranje takvih nizova možemo obaviti jedino ručnim pisanjem funkcije za sortiranje, u kojoj ćemo razmjenu obavljati pomoću funkcije “strcpy”. Ovo nije preveliko ograničenje, s obzirom da upotrebu dvodimenzionalnih nizova znakova svakako treba izbjegavati, zbog neracionalnog trošenja memorije.

Izlaganje o sortiranju i funkcijama za sortiranje ćemo završiti napomenom da biblioteka “cstdlib” također posjeduje funkciju za sortiranje nizova pod nazivom “qsort”, koja je naslijedena iz jezika C. Međutim, u jeziku C++ ne savjetuje se upotreba ove funkcije iz više razloga. Na prvom mjestu, njena

upotreba je komplikovanija. Ona prima četiri parametra, od kojih je četvrti parametar pokazivač na funkciju kriterija koji se obavezno mora zadati (tj. ne može se izostaviti). Međutim, funkcija kriterija se zadaje na mnogo komplikovaniji način, preko generičkih pokazivača, što komplificira upotrebu funkcije "qsort". Drugi nedostatak funkcije "qsort" je što se ona može primijeniti samo na POD tipove podataka, za koje se podrazumijeva da se mogu bezbjedno kopirati bajt po bajt (svi tipovi naslijedeni iz jezika C su takvi, ali C++ posjeduje tipove za koje to ne vrijedi). Na primjer, funkcija "qsort" može izazvati pravi haos ukoliko se pokuša primijeniti na sortiranje niza čiji su elementi tipa "string". Ovo su dovoljni razlozi za izbjegavanje upotrebe funkcije "qsort" u C++ programima (ove napomene su namijenjene pretežno onima koji su upoznati sa funkcijom "qsort", na primjer programerima koji su prije programirali u jeziku C).

još malo o qsort

Recimo sada nekoliko riječi o pretraživanju. Generalno je problem pretraživanja znatno jednostavniji od problema sortiranja, mada i pretraživanje raznih složenih struktura podataka (kao što su npr. grafovi, stabla, liste itd.) može biti dosta komplikovano. Pošto se opisanim strukturama podataka na ovom mjestu nećemo baviti, ovdje ćemo se ograničiti samo na pretraživanje *nizova*. Cilj pretraživanja niza je pronaći *na kojoj poziciji* se nalazi element koji *posjeduje neko svojstvo*. Na primjer, često je potrebno pronaći gdje se nalazi element koji je jednak nekoj fiksnoj vrijednosti koja se obično naziva *ključ pretrage*. Sljedeća generička funkcija vraća kao rezultat indeks prvog elementa u nizu "niz" od "n" elemenata koji je jednak vrijednosti "kljuc". U slučaju da takav element ne postoji, funkcija vraća kao rezultat -1, što može služiti kao indikacija da pretraga nije bila uspješna (s obzirom da -1 ne može nikada biti indeks nekog elementa u nizu):

```
template <typename Tip>
int Pretraga(Tip niz[], int n, Tip kljuc) {
    for(int i = 0; i < n; i++)
        if(niz[i] == kljuc) return i;
    return -1;
}
```

Ovakav način pretraživanja naziva se *linearno ili sekvencialno pretraživanje*, s obzirom da se elementi niza pretražuju "linijski", element po element. Napisanu funkciju "Pretraga" nije teško prepraviti tako da kao parametre prihvata pokazivače koji omeđuju dio niza na koji se pretraga odnosi (poput funkcije "find" iz biblioteke "algorithm"), što se ostavlja čitatelju ili čitateljici za vježbu. Također, nije teško napraviti funkciju koja će tražiti element sa nekim drugim svojstvom, a ne samo element koji je jednak ključu. Moguće je napraviti i univerzalnu funkciju za pretragu, u kojoj se traženo svojstvo zadaje preko pokazivača na funkciju koja ispituje ispunjenost traženog svojstva (poput funkcije "find_if" iz biblioteke "algorithm"). Inače, funkcije "find" i "find_if" iz biblioteke "algorithm" obavljaju upravo sekvencialno pretraživanje (samo što one umjesto indeksa traženog elementa vraćaju kao rezultat pokazivač na njega, što je univerzalnije).

Očigledno će pretraživanje niza od n elemenata ovim metodom zahtijevati najviše n prolazaka kroz petlju, i to ukoliko se element koji tražimo nalazi na posljednjem mjestu, ili ga uopće nema u nizu. To i nije tako mnogo, mada se može postaviti pitanje da li je moguće pretragu obaviti *brže*. Lako je pokazati da za niz posve proizvoljnog sadržaja ne postoji nikakva mogućnost ubrzanja pretrage u odnosu na linearno pretraživanje. Međutim, u slučaju da je niz koji se pretražuje prethodno *sortiran* (npr. u rastući poretku), pretraga se može obaviti *znatno brže*, postupkom poznatim pod imenom *binarno pretraživanje*. Naime, vrijednost koju tražimo prvo treba tražiti *u sredini* niza. U slučaju da je element u sredini niza veći od traženog elementa, možemo odmah znati da se traženi element može nalaziti samo u *prvoj polovini niza*, s obzirom da je niz sortiran u rastući poretku. Slično, u slučaju da je element u sredini niza manji od

traženog elementa, on se može nalaziti samo u *drugoj polovini niza*. Tako, u sljedećem koraku možemo veličinu niza reducirati na polovinu, i ponavljati isti postupak sve dok pretraga ne bude uspješna, ili dok se veličina niza ne svede na jedan element (primijetimo da je metod polovljenja intervala za nalaženje nula funkcije o kojem smo ranije govorili, zapravo kontinualna varijanta binarne pretrage). Ova ideja implementirana je u sljedećoj funkciji:

```
template <typename UporediviTip>
int BinarnaPretraga(UporediviTip niz[], int n, UporediviTip kljuc) {
    int lpok(0), dpok(n - 1);
    while(lpok <= dpok) {
        int sredina = (lpok + dpok) / 2;
        if(niz[sredina] == kljuc) return sredina;
        else if(niz[sredina] < kljuc) lpok = sredina + 1;
        else dpok = sredina - 1;
    }
    return -1;
}
```

Ovdje su, slično kao i kod “quick sort” algoritma, iskorištena dva indeksa “*lpok*“ i “*dpok*“ koji na početku pokazuju na prvi i posljednji element niza. Nakon svakog poređenja, ovi indeksi se pomjeraju na početak i kraj onog dijela niza u kojem treba nastaviti pretragu. Kako se u svakom koraku veličina niza prepovoljava (tj. opada eksponencijalnom brzinom), vrijeme izvršavanja ovog algoritma u najgorem slučaju proporcionalna je sa $\log_2 n$ umjesto sa n . Razlika može biti drastična. Npr. za $n=1000000$ linearna pretraga može zahtijevati i do 1000000 koraka, dok binarna pretraga pronalazi traženi element u najviše 20 koraka, što je ubrzanje od 50000 puta!

Očigledna mana binarnog pretraživanja je što niz koji se pretražuje mora biti sortiran. Ukoliko trebamo pretraživati nesortiran niz, prirodno je postaviti pitanje da li se više isplati koristiti linearnu pretragu, ili prvo sortirati niz pa koristiti binarnu pretragu. Odgovor zavisi od toga koliko elemenata niza treba pretraživati (tj. koliko puta ćemo pozivati funkciju za pretraživanje). Ukoliko je potrebno tražiti svega nekoliko elemenata, bolje je koristiti linearno pretraživanje, jer i najbrže sortiranje troši znatno više vremena od linearne pretrage. S druge strane, ukoliko je potrebno da tražimo veliki broj različitih elemenata, često se isplati prvo sortirati niz (s obzirom da se to radi samo jedanput), a nakon toga za svaki element koji se traži koristiti binarnu pretragu. Treba još naglasiti i to da ukoliko postoji više elemenata niza koji su jednaki ključu, klasična binarna pretraga nalazi *neki od njih*, a ne nužno *prvi* kao što je slučaj kod linearne pretrage. Doduše, u slučaju potrebe, lako je naći poziciju prvog elementa koji je jednak ključu, prostom pretragom unazad od pozicije nekog od nađenih elementa jednakih ključu.

Biblioteka “*algorithm*“ također sadrži i funkcije koje obavljaju binarnu pretragu (tako da su znatno brže od srodnih funkcija koje obavljaju klasičnu sekvencijsku pretragu). Najvažnije funkcije iz ove grupe funkcija prikazane su u sljedećoj tabeli. Sve one podrazumijevaju da je razmatrani niz elemenata prethodno sortiran u rastući poredak. U suprotnom, rezultati neće biti u skladu sa očekivanjima. Ukoliko je niz sortiran u neki drugi poredak (a ne rastući), ove funkcije se također mogu koristiti, ali se tada kao dodatni parametar (na posljednjem mjestu) mora navesti funkcija kriterija koja definira upotrijebljeni poredak (na isti način kao i kod funkcija za sortiranje):

`binary_search(p1, p2, v)`

Vraća kao rezultat “**true**“ ili “**false**“ u ovisnosti da li se element *v* nalazi ili ne nalazi u bloku elemenata između pokazivača *p1* i *p2*.

`lower_bound(p1, p2, v)`

Vraća kao rezultat pokazivač na prvi element u bloku elemenata između pokazivača *p1* i *p2* čija je vrijednost veća ili jednaka *v*.

`upper_bound(p1, p2, v)`

Vraća kao rezultat pokazivač koji pokazuje iza posljednjeg elementa u bloku elemenata između pokazivača $p1$ i $p2$ čija je vrijednost manja ili jednaka v .

još specijalnih funkcija za pretragu

29. Strukturalni tipovi podataka

Nizovi se u programiranju koriste za smještanje skupine povezanih podataka od kojih su svi *istog tipa* (na primjer, popis ocjena za jednog studenta ili za grupu studenata, količine padavina u toku svakog mjeseca u godini, itd.), i kojima se pristupa preko *indeksa* (tj. rednog broja). S druge strane, u praktičnim problemima se često pojavljuje i situacija kada treba čuvati skupinu povezanih podataka *različitog tipa*, na primjer ime (sa prezimenom), odjeljenje, platni broj i iznos plate nekog radnika. Pored toga, često se javlja i potreba za pamćenjem skupine podataka *istog tipa*, ali na koje nije prirodno primijeniti indeksiranje, jer nisu *iste prirode*. Na primjer, datum je određen sa tri cijelobrojna podatka (dan, mjesec i godina), ali koje nije prirodno čuvati u formi niza, jer se ne vidi jasno kakva bi bila veza između indeksa i značenja podatka određenog tim indeksom.

Struktura podataka koja može čuvati skupinu *raznorodnih podataka*, koji se mogu razlikovati kako po prirodi tako i po tipu, i kojima se ne pristupa po *indeksu* već po *imenu*, naziva se *slog* ili *zapis* (engl. *record*). Međutim, kako se ovi tipovi podataka u jezicima C i C++ definiraju pomoću ključne riječi “**struct**”, to se u ovim jezicima oni najčešće nazivaju prosto *strukture*, dok se nazivi *slog* odnosno *zapis* češće susreću u drugim programskim jezicima (pogotovo u *Pascalu*). Iza ključne riječi “**struct**” se u vitičastim zagradama navode *deklaracije individualnih elemenata* od kojih se struktura sastoji. Na primjer, sljedeća deklaracija uvodi strukturalni tip “*Radnik*“ koji opisuje radnika u preduzeću:

```
struct Radnik {  
    char ime[40];  
    char odjeljenje[20];  
    int platni_broj;  
    double plata;  
};
```

Obratimo pažnju na tačka-zarez koji se nalazi iza završne vitičaste zagrade, koji se veoma često zaboravlja, a čiju ćemo ulogu uskoro objasniti. Individualni elementi kao što su “ime”, “odjeljenje”, “platni_broj” i “plata” nazivaju se *polja* (engl. *fields*), *atributi* ili *podaci članovi strukture*.

Slično ključnoj riječi “**typedef**”, ključna riječ “**struct**” samo definira *novi tip* (koji ćemo zvati *strukturalni tip*), a ne i konkretnе primjerke objekata tog tipa. Tako, na primjer, tip “*Radnik*“ samo opisuje koja karakteristična svojstva (attribute) posjeduje bilo koji radnik. Tek nakon što deklariramo odgovarajuće promjenljive strukturalnog tipa, kao na primjer u deklaraciji

```
Radnik sekretar, sluzbenik, monter, portir;
```

imamo *konkretne promjenljive* “sekretar”, “sluzbenik”, “monter” i “portir” sa kojima

možemo manipulirati u programu. Napomenimo da je jezik C zahtijevao da se prilikom deklaracije strukturnih promjenljivih ponovi ključna riječ “**struct**”, kao u sljedećoj deklaraciji:

```
struct Radnik Sekretar, Sluzbenik, Monter, Portir;
```

Ovakva sintaksa dozvoljena je i u jeziku C++, ali se ne preporučuje.

Moguće je odmah prilikom deklaracije strukturnog tipa definirati i konkretne primjerke promjenljivih tog tipa, kao u sljedećem primjeru:

```
struct Radnik {
    char ime[40];
    char odjeljenje[20];
    int platni_broj;
    double plata;
} sekretar, sluzbenik, monter, portir;
```

Sada postaje jasna uloga tačka-zareza iza zatvorene vitičaste zagrade: njime govorimo da u tom trenutku ne želimo da definiramo ni jedan konkretan primjerak tog tipa (napomenimo još da se danas deklariranje konkretnih primjeraka strukture istovremeno sa deklariranjem strukture smatra lošim stilom pisanja).

Pojedinačna polja (atributi, podaci članovi) unutar sloganovnih promjenljivih ponašaju se poput *individualnih promjenljivih* odgovarajućeg tipa. Njima se pristupa tako što prvo navedemo *ime sloganove promjenljive*, zatim znak (zapravo operator) *tačka* (“.”) i, konačno, *ime polja unutar sloganove promjenljive*. Na primjer, sve sljedeće konstrukcije su potpuno korektne:

```
strcpy(sekretar.ime, "Ahmed Hodžić");
sekretar.plata = 580;
strcpy(sluzbenik.ime, "Dragan Kovačević");
cout << sekretar.plata;
cin >> sekretar.odjeljenje;
cin.getline(portir.ime, sizeof portir.ime);
portir.plata = sekretar.plata - 100;
```

Treba naglasiti da nije moguće *jednim iskazom* postaviti sva polja u nekoj strukturnoj promjenljivoj (npr. promjenljivoj “sekretar”), već je to potrebno učiniti nizom iskaza poput:

```
strcpy(sekretar.ime, "Ahmed Hodžić");
strcpy(sekretar.odjeljenje, "Marketing");
sekretar.platni_broj = 34;
sekretar.plata = 530;
```

Ipak, moguće je odjednom *inicijalizirati* sva polja u nekoj strukturnoj promjenljivoj, ali samo *prilikom deklaracije* te promjenljive (slično kao što vrijedi za nizove). Tako je moguće pisati:

```
Radnik sekretar = {"Ahmed Hodžić", "Marketing", 34, 530};
```

Vidimo da se inicijalizacija strukturnih promjenljivih vrši tako da se u vitičastim zagradaima navode inicijalizacije za *svako od polja posebno*, onim redom kako su definirana u deklaraciji sloga. Inače, tipovi podataka koji se sastoje od *fiksnog broja* individualnih komponenti, i koje se mogu inicijalizirati navođenjem vrijednosti pojedinih komponenti u vitičastim zagradaama, nazivaju se *agregati*. U jeziku C++ agregati su *nizovi i strukture* (ali ne i recimo *vektori*, koji se ne mogu inicijalizirati na takav način).

Bilo koja slogovna promjenljiva može se čitava dodijeliti drugoj slogovnoj promjenljivoj *istog tipa*. Na primjer, dozvoljeno je pisati:

```
monter = sluzbenik;
```

Na ovaj način se sva polja iz promjenljive “*sluzbenik*” prepisuju u odgovarajuća polja promjenljive “*monter*”. Dodjela je, zapravo, *jedina operacija* koja je inicijalno podržana nad *strukturama kao cjelinama* (zapravo, smije se još uzeti i *adresa strukture* pomoću operatora “&”). Sve druge operacije, uključujući čitanje sa tastature i ispis na ekran, inicijalno *nisu definirane*, nego se moraju obavljati isključivo nad *individualnim poljima unutar struktura*. Kasnije ćemo vidjeti da je korištenjem tzv. *preklapanja operatora* moguće *samostalno definirati* i druge operacije koje će se obavljati nad čitavim strukturama, ali inicijalno takve operacije nisu podržane. Stoga su, bez upotrebe preklapanja operatora, sljedeće konstrukcije *besmislene*, i dovode do prijave greške:

```
cout << sekretar;
cin >> portir;
```

Bitno je napomenuti da se polja koja čine strukturu ne mogu inicijalizirati unutar deklaracije same strukture. Naime, polja strukture ne postoje kao neovisni objekti, nego samo kao dijelovi neke kontretne promjenljive struktornog tipa, tako da se ne bi moglo znati na šta se takva inicijalizacija odnosi. Zbog toga su deklaracije poput sljedeće besmislene:

```
struct Radnik {
    char ime[40] = "Ahmed Hodžić";
    char odjeljenje[20] = "Marketing";
    int platni_broj(34);
    double plata = 530;
}
```

Od navedenog pravila postoji jedan izuzetak: polja strukture mogu se (i moraju) inicijalizirati unutar deklaracije strukture ukoliko su deklarirana sa kvalifikatorima “**const**” i “**static**” (preciznije, sa oba ova kvalifikatora istovremeno). O smislu takvih kvalifikacija govorićemo kasnije, kada se upoznamo sa pojmom klasa. Pitanje kako se uopće inicijaliziraju eventualna polja deklarirana sa kvalifikatorom “**const**” i kakav je njihov smisao, također ćemo ostaviti za kasnije.

Strukture se *mogu prenositi kao parametri u funkcije*, kako po vrijednosti, tako i po referenci. Međutim, interesantno je da se, za razliku od nizova, strukture mogu *vraćati kao rezultati iz funkcija*. Na primjer, pretpostavimo da želimo da definiramo tip podataka koji opisuje vektor u prostoru koji se, kao što znamo, može opisati sa tri koordinate *x*, *y* i *z*. U jednom od ranijih poglavlja za tu svrhu koristili smo običan niz od tri elementa. Međutim, kako funkcije *ne mogu da vraćaju nizove kao rezultate*, funkcija “*VektorskiProizvod*” koju smo pisali zahtjevala je tri parametra: dva vektora koji se množe, i treći vektor u koji će biti prihvaćen rezultat. Tom prilikom je demonstrirano i elegantnije rješenje, zasnovano na upotrebi izvedenog tipa “*vector*” (definiranog u istoimenoj biblioteci), čiji se primjeri mogu vraćati kao rezultati iz funkcija. S druge strane, vidjeli smo da ni to rješenje nije bez svojih nedostataka. Mnogo

bolje i elegantnije rješenje možemo dobiti ukoliko definiramo vlastiti strukturni tip podataka "Vektor" koji opisuje vektor u prostoru, na primjer pomoću sljedeće deklaracije:

```
struct Vektor {  
    double x, y, z;  
};
```

Kako se strukture mogu prenositi u funkcije i vraćati kao rezultati iz funkcije, sa ovakvim tipovima podataka se često može elegantnije manipulirati nego sa nizovima. Ovo je ilustrirano u sljedećem programu, koji čita koordinate dva vektora sa tastature, a zatim ispisuje njihovu sumu i njihov vektorski proizvod (pri čemu se vektor ispisuje u formi tri koordinate međusobno razdvojene zarezima, unutar vitičastih zagrada):

```
#include <iostream>  
using namespace std;  
  
struct Vektor {  
    double x, y, z;  
};  
  
void UnesiVektor(Vektor &v) {  
    cout << "X = "; cin >> v.x;  
    cout << "Y = "; cin >> v.y;  
    cout << "Z = "; cin >> v.z;  
}  
  
Vektor ZbirVektora(const Vektor &v1, const Vektor &v2) {  
    Vektor v3;  
    v3.x = v1.x + v2.x; v3.y = v1.y + v2.y; v3.z = v1.z + v2.z;  
    return v3;  
}  
  
Vektor VektorskiProizvod(const Vektor &v1, const Vektor &v2) {  
    Vektor v3;  
    v3.x = v1.y * v2.z - v1.z * v2.y;  
    v3.y = v1.z * v2.x - v1.x * v2.z;  
    v3.z = v1.x * v2.y - v1.y * v2.x;  
    return v3;  
}  
void IspisiVektor(const Vektor &v) {  
    cout << "{" << v.x << "," << v.y << "," << v.z << "}";  
}  
  
int main() {  
    Vektor a, b;  
    cout << "Unesi prvi vektor:\n";  
    UnesiVektor(a);  
    cout << "Unesi drugi vektor:\n";  
    UnesiVektor(b);  
    cout << "Suma ova dva vektora je: ";  
    IspisiVektor(ZbirVektora(a, b));  
    cout << endl << "Njihov vektorski proizvod je: ";  
    IspisiVektor(VektorskiProizvod(a, b));  
    return 0;  
}
```

Vidimo da ovaj program djeluje znatno prirodnije u odnosu na sličan program u kojem smo za predstavljanje vektora koristili niz od tri elementa (uskoro ćemo vidjeti da upotreba klase i

preoprerećivanja operatora nudi mogućnost još prirodnijeg rada sa složenim tipovima podataka). Također, treba primijetiti da su se funkcije "ZbirVektora" i "VektorskiProizvod" mogle napisati jednostavnije, tako da se iskoristi mogućost da se polja neke strukturne promjenljive mogu inicijalizirati odmah po njenoj deklaraciji:

```
Vektor ZbirVektora(const Vektor &v1, const Vektor &v2) {
    Vektor v3 = {v1.x + v2.x, v1.y + v2.y, v1.z + v2.z};
    return v3;
}

Vektor VektorskiProizvod(const Vektor &v1, const Vektor &v2) {
    Vektor v3 = {v1.y * v2.z - v1.z * v2.y, v1.z * v2.x - v1.x * v2.z,
                 v1.x * v2.y - v1.y * v2.x};
    return v3;
}
```

Razmotrimo malo detaljnije prirodu parametara u upotrijebljenim funkcijama. Jasno je da formalni parametar "v" u funkciji "UnesiVektor" mora biti referenca, s obzirom da ova funkcija mora promijeniti vrijednost svog stvarnog parametra. Međutim, u svim ostalim funkcijama, formalni parametri su *reference na konstantne objekte*. Kvalifikator "**const**" je upotrijebljen iz dva razloga. Prvo, na taj način je spriječeno da funkcije nehotice promijene sadržaj parametra. Drugi, važniji razlog je to što je na taj način omogućeno da se kao stvarni parametar ne mora upotrijebiti nužno promjenljiva, nego je moguće upotrijebiti *bilo koji legalni izraz* čiji je tip "Vektor" (npr. rezultat neke druge funkcije koja vraća objekat tipa "Vektor"), s obzirom da se reference na konstantne objekte mogu vezati na privremene objekte koji nastaju kao rezultat izračunavanja izraza (npr. na objekat koji nastaje vraćanjem vrijednosti iz funkcije). Da smo u funkciji "IspisiVektor" kao formalni parametar upotrijebili običnu referencu a ne referencu na konstantni objekat, konstrukcija poput

```
IspisiVektor(ZbirVektora(a, b));
```

ne bi bila moguća. Naravno, kao alternativna mogućnost formalni parametri ovih funkcija uopće nisu morali biti reference, odnosno mogao se koristiti prenos parametara *po vrijednosti*. Međutim, korištenjem prenosa po referenci sprečava se nepotrebitno kopiranje stvarnog parametra u formalni, koje bi se, u našem slučaju, svelo na kopiranje *tri realna podatka* po svakom parametru (s obzirom da se tip "Vektor" sastoji od tri realna polja). To i nije tako mnogo, ali pošto strukture mogu biti neuporedivo masivnije (tako da njihovo kopiranje može biti veoma zahtjevno), trebamo se odmah navikavati na to da prenos parametara struktturnog tipa *po vrijednosti* ne treba koristiti ukoliko to nije zaista neophodno.

Interesantno je spomenuti jedan neobičan detalj vezan za vraćanje struktura kao rezultata iz funkcije. Naime, standard jezika C++ propisuje da je rezultat funkcije koja vraća strukturu l-vrijednost, odnosno može se naći sa lijeve strane znaka jednakosti! To zapravo znači, da su konstrukcije poput sljedećih

```
ZbirVektora(a, b) = a;
VektorskiProdukt(a, b).x = 3;
```

sintaksno ispravne, bez obzira što su besmislene! Naime, u oba slučaja dodjela se vrši nad privremenim objektom koji predstavlja rezultat vraćen iz funkcije (odnosno njegovim poljem "x" u drugom slučaju), koji odmah nakon toga biva izgubljen, tako da izvršena dodjela nema efekta (naravno, sasvim drugačiju situaciju imamo u slučaju kada funkcija vrati *referencu* na neki objekat koji nastavlja postojati). Razlog za ovaku čudnu konvenciju leži u tome što je za strukturne tipove podataka smisao operatora dodjele "=" moguće *promijeniti* (o tome ćemo govoriti kasnije), tako da je naizgled besmislenim konstrukcijama poput navedenih moguće dati precizan smisao (ovo se koristi uglavnom samo za realizaciju nekih prilično prljavih trikova). Naravno, zbog ove konvencije se ne trebamo previše sekirati, s obzirom da ni jedan

programer zdrave pameti ne bi napisao konstrukcije poput gore navedenih, pri uobičajenom tretmanu operatora dodjele “=”. Međutim, ukoliko nam smeta čak i sama činjenica da su ovakve konstrukcije dozvoljene, jezik C++ nudi rješenje i za to. Naime, standard jezika C++ kaže da rezultat funkcije koja vraća strukturu neće biti l-vrijednost ukoliko se povratni tip označi kvalifikatorom “**const**” (ovim se efektivno zabranjuje upotreba povratne vrijednosti sa lijeve strane znaka jednakosti). Drugim riječima, ukoliko zaglavljemo funkcije “ZbirVektora” napišemo na sljedeći način:

```
const Vektor ZbirVektora(const Vektor &v1, const Vektor &v2) {
```

tada prva od dvije prethodno navedene besmislene konstrukcije više neće biti prihvaćena od strane kompjajlera.

Rad sa strukturnim tipovima ćemo ilustrirati na još jednom jednostavnom primjeru. Često se zahtijeva program koji štampa detalje u uplatama na predračunskom izvještaju, npr. za platne spiskove ili za kontrolu u samoposluzi. Ovdje je napisan jednostavan program koji koristi strukturu “Uplata” za čuvanje podataka o uplati. Strukturna promjenljiva “nova_uplata” se prvo prenosi (po referenci) u funkciju “CitajUplatu”, koja postavlja vrijednosti u sva njena polja. Nakon toga, ona se prenosi u funkciju “StampajUplatu”, koja ispisuje na ekran detalje o uplati u zahtijevanom obliku (u nekom realnijem programu ovi detalji bi se vjerovatno trebali ispisivati na štampač umjesto na ekran). Program za svaku upлатu traži da se unese datum uplate u obliku “*dan/mjesec/godina*” (npr. “23/7/1997”), a zatim ime uplatitelja, i iznos uplate.

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

using namespace std;

struct Uplata {
    int dan, mjesec, godina;
    char uplatitelj[50];
    double iznos;
};

void CitajUplatu(Uplata &uplata) {
    char crta; // Služi za preskakanje znaka '/' prilikom unosa datuma
    cout << "Unesi datum uplate u obliku DAN/MJ/GOD: ";
    cin >> uplata.dan >> crta >> uplata.mjesec >> crta >> uplata.godina;
    cin.ignore(1000, '\n'); // Obriši zaostalo "smeće"
    cout << "Unesi ime uplatitelja: ";
    cin.getline(uplata.uplatitelj, sizeof uplata.uplatitelj);
    cout << "Unesi iznos uplate: ";
    cin >> uplata.iznos;
}

void StampajUplatu(const Uplata &uplata) {
    cout << endl << uplata.uplatitelj << " je uplatio(la) iznos od "
        << setprecision(2) << uplata.iznos << " KM dana " << uplata.dan
        << "/" << uplata.mjesec << "/" << uplata.godina << endl << endl;
}

int main() {
    Uplata nova_uplata;
    char odgovor = 'D';
    while(odgovor == 'D' || odgovor == 'd') {
        CitajUplatu(nova_uplata);
        StampajUplatu(nova_uplata);
        cout << "Želite li još uplata (D/N)?";
    }
}
```

```

    cin >> odgovor;
    cin.ignore(1000, '\n');      // Obriši eventualni višak znakova
    cout << endl;
}
return 0;
}

```

Obratimo pažnju između tipa “Uplata” i formalnog parametra “uplata” koji predstavlja konkretni primjerak tipa “Uplata”. Mada se obično smatra da treba izbjegavati identifikatore koji se međusobno razlikuju samo po rasporedu malih i velikih slova, konvencija da se konkretnom primjerku nekog tipa daje ime identično imenu tipa samo bez velikog početnog slova dosta se često susreće u praksi, i može se smatrati izuzetkom u odnosu na konvencije o imenovanju identifikatora.

U prethodnom primjeru *zaista nije bilo neophodno* čuvati podatke koje sadrže ime uplatitelja, iznos uplate itd. u strukturnoj promjenljivoj, jer smo u jednom trenutku radili *samo sa jednom uplatom*. Stoga bi korištenje *posebnih promjenljivih* za pamćenje imena uplatitelja, iznosa uplate itd. sasvim lijepo poslužilo svrsi (mada je upotreba strukturnog tipa omogućila da prenosimo u funkcije sve podatke za obradu *odjedanput* koristeći samo *jedan parametar*). Ipak, u stvarnom životu, mnogo se česće javlja potreba za obradom *skupine slogova*, kao što je, na primjer, skupina *svih uplata* uplaćenih u proteklom mjesecu, ili podaci o *svim radnicima* u preduzeću, ili *svim studentima* na jednoj godini studija. Za modeliranje takvih podataka iz stvarnog života možemo koristiti *nizove struktura*, odnosno nizove *čiji je svaki element strukturnog tipa*. Na primjer, sljedeće definicije tipa i deklaracije promjenljivih omogućavaju nam da predstavimo skupinu radnika:

```

const int MaksimalniBrojRadnika(100);

struct Radnik {
    char ime[40];
    char odjeljenje[20];
    int platni_broj;
    double plata;
};

Radnik radnici[MaksimalniBrojRadnika];

```

Elementima ovakvog niza se pristupa na uobičajeni način, preko *indeksa*. *Svaki element niza predstavlja strukturu*. Tako se, na primjer, polju “ime“ trećeg elementa promjenljive “radnici“ (preciznije, elementa sa indeksom 3, s obzirom da numeracija indeksa počinje od nule) pristupa pomoću konstrukcije “radnici[3].ime”. Tako, na primjer, ukoliko pretpostavimo da cijelobrojna promjenljiva “broj_radnika” predstavlja stvarni broj radnika u preduzeću, možemo iskoristiti konstrukciju poput

```

for(int i = 0; i < broj_radnika; i++)
    cout << radnici[i].ime << " " << radnici[i].plata << endl;

```

da odštampamo ime i platu za svakog radnika u preduzeću.

Polja unutar struktura i sama mogu biti složeni tipovi podataka. Ona mogu biti nizovnog, pa čak i ponovo strukturnog tipa. Tako, kombinirajući nizove i slogove, možemo sagraditi vrlo složene strukture podataka koje su od koristi za modeliranje podataka sa kojima se susrećemo u realnim problemima. Na primjer, zamislimo da želimo opisati jedan razred u školi. Razred se sastoji od učenika, a svaki učenik opisan je imenom, prezimenom, datumom rođenja, spiskom ocjena, prosjekom i informacijom da li je učenik prošao ili nije. Spisak ocjena predstavlja niz cijelih brojeva (koji ima onoliko elemenata koliko ima predmeta), dok se datum rođenja sastoji od tri cjeline: dana, mjeseca i godine rođenja, koji su cijeli

brojevi. Stoga je najprirodnije definirati strukturalni tip "Ucenik", koji opisuje jednog učenika. Atributi strukture "Ucenik" mogu biti "ime", " prezime", "datum_rodjenja", "ocjene", "projek" i "prolaz". Atributi "ime" i " prezime" su očigledno nizovi znakova, atribut "projek" je realnog tipa, dok je "prolaz" logička promjenljiva. Atribut "ocjene" ćemo definirati kao niz cijelih brojeva, dok ćemo atribut "datum_rodjenja" definirati da bude tipa "Datum", pri čemu je "Datum" ponovo strukturalni tip koji se sastoji od tri atributa: "dan", "mjesec" i "godina". Na kraju, definiraćemo promjenljivu "ucenici", koja će biti niz čiji su elementi tipa "Ucenik". Na osnovu provedene analize možemo napisati sljedeće deklaracije:

```
const int MaxBrojUcenika(30), BrojPredmeta(12);

struct Datum {
    int dan, mjesec, godina;
};

struct Ucenik {
    char ime[20], prezime[20];
    Datum datum_rodjenja;
    int ocjene[BrojPredmeta];
    double projek;
    bool prolaz;
};
Ucenik ucenici[MaxBrojUcenika];
```

Primijetimo da smo tip "Datum" definirali prije nego što smo ga upotrijebili unutar tipa "Ucenik". Jezik C++ nikada ne dozvoljava korištenje bilo kojeg pojma koji prethodno nije definiran (ili bar najavljen, odnosno deklariran, kao što je to slučaj pri korištenju prototipova funkcija).

Pojedinim dijelovima ovakve složene strukture pristupamo kombinacijom indeksiranja i navođenja imena polja, na sasvim logičan način. Na primjer, ukoliko treba postaviti godinu rođenja trećeg učenika (odnosno učenika sa indeksom 3) na vrijednost "1988", i devetu ocjenu istog učenika na vrijednost "4", možemo pisati:

```
ucenici[3].datum_rodjenja.godina = 1988;
ucenici[3].ocjene[9] = 4;
```

Rad sa složenim tipovima podataka ilustriraćemo programom koji prvo zahtijeva unos osnovnih podataka o svim učenicima u razredu, zatim računa projek i utvrđuje prolaznost za svakog učenika, i na kraju, prikazuje na ekranu izvještaj o svim učenicima u razredu, sortiran u opadajući redoslijed po projeku. Izvještaj se sastoji od rečenica poput

Učenik Marko Marković, rođen 17.3.1989. ima projek 3.89

za one učenike koji su prošli, ili rečenica poput

Učenik Janko Janković, rođen 22.5.1989. mora ponavljati razred

za one učenike koji nisu prošli. Program je pisan strukturano, uz pomoć mnoštva funkcija, tako da je lakše pratiti logiku programa, i eventualno vršiti kasnije modifikacije. Također su korišteni prototipovi funkcija, što je omogućilo da prvo pišemo glavni program, pa onda potprograme. Na ovaj način se dosljednije prati logika razvoja *od vrha na niže*, po kojoj prvo pišemo kostur programa, pa tek onda njegove detalje, počevši od krupnijih ka sitnijim.

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

```

#include <algorithm>
using namespace std;
const int MaxBrojUcenika(30), BrojPredmeta(12);
struct Datum {
    int dan, mjesec, godina;
};
struct Ucenik {
    char ime[20], prezime[20];
    Datum datum_rodjenja;
    int ocjene[BrojPredmeta];
    double prosjek;
    bool prolaz;
};
int main() {
    void UnesiUcenike(Ucenik ucenici[], int broj_ucenika);
    void ObradiUcenike(Ucenik ucenici[], int broj_ucenika);
    void IspisiIzvjestaj(Ucenik ucenici[], int broj_ucenika);
    int broj_ucenika;
    cout << "Koliko ima učenika: ";
    cin >> broj_ucenika;
    Ucenik ucenici[MaxBrojUcenika];
    UnesiUcenike(ucenici, broj_ucenika);
    ObradiUcenike(ucenici, broj_ucenika);
    IspisiIzvjestaj(ucenici, broj_ucenika);
    return 0;
}
void UnesiUcenike(Ucenik ucenici[], int broj_ucenika) {
    void UnesiJednogUcenika(Ucenik &ucenik);
    for(int i = 0; i < broj_ucenika; i++) {
        cout << "Unesite podatke za " << i + 1 << ". učenika:\n";
        UnesiJednogUcenika(ucenici[i]);
    }
}
void UnesiJednogUcenika(Ucenik &ucenik) {
    void UnesiDatum(Datum &datum);
    void UnesiOcjene(int ocjene[], int broj_predmeta);
    cout << " Ime: "; cin >> ucenik.ime;
    cout << " Prezime: "; cin >> ucenik.prezime;
    UnesiDatum(ucenik.datum_rodjenja);
    UnesiOcjene(ucenik.ocjene, BrojPredmeta);
}
void UnesiDatum(Datum &datum) {
    cout << " Dan rođenja: "; cin >> datum.dan;
    cout << " Mjesec rođenja: "; cin >> datum.mjesec;
    cout << " Godina rođenja: "; cin >> datum.godina;
}
void UnesiOcjene(int ocjene[], int broj_predmeta) {
    for(int i = 0; i < broj_predmeta; i++) {
        cout << " Ocjena iz " << i + 1 << ". predmeta: ";
        cin >> ocjene[i];
    }
}
void ObradiUcenike(Ucenik ucenici[], int broj_ucenika) {

```

```

void ObradiJednogUcenika(Ucenik &ucenik);
bool DaLiJeBoljiProsjek(const Ucenik &u1, const Ucenik &u2);
for(int i = 0; i < broj_ucenika; i++)
    ObradiJednogUcenika(ucenici[i]);
    sort(ucenici, ucenici + broj_ucenika, DaLiJeBoljiProsjek);
}

void ObradiJednogUcenika(Ucenik &ucenik) {
    double suma_ocjena(0);
    ucenik.prosjek = 1; ucenik.prolaz = false;
    for(int i = 0; i < BrojPredmeta; i++) {
        if(ucenik.ocjene[i] == 1) return;
        suma_ocjena += ucenik.ocjene[i];
    }
    ucenik.prolaz = true;
    ucenik.prosjek = suma_ocjena / BrojPredmeta;
}

bool DaLiJeBoljiProsjek(const Ucenik &u1, const Ucenik &u2) {
    return u1.prosjek > u2.prosjek;
}

void IspisiIzvjestaj(Ucenik ucenici[], int broj_ucenika) {
    void IspisiJednogUcenika(const Ucenik &ucenik);
    cout << endl;
    for(int i = 0; i < broj_ucenika; i++)
        IspisiJednogUcenika(ucenici[i]);
}

void IspisiJednogUcenika(const Ucenik &ucenik) {
    void IspisiDatum(const Datum &datum);
    cout << "Učenik " << ucenik.ime << " " << ucenik.prezime << " rođen ";
    IspisiDatum(ucenik.datum_rodjenja);
    if(ucenik.prolaz)
        cout << " ima prosjek " << setprecision(2) << ucenik.prosjek;
    else
        cout << " mora ponavljati razred";
    cout << endl;
}

void IspisiDatum(const Datum &datum) {
    cout << datum.dan << "." << datum.mjesec << "." << datum.godina;
}

```

Za sortiranje je upotrijebljena funkcija "sort" iz biblioteke "algorithm". Za tu potrebu je definirana i funkcija kriterija "DaLiJeBoljiProsjek", koja je prenesena funkciji "sort" kao parametar. Funkcija kriterija je gotovo uvijek potrebna kada se sortiraju nizovi struktura, s obzirom da za strukture inicijalno nije definiran poredak, pa samim tim ni smisao operatara "<" (mada se, pomoću tehnikе preklapanja operatora, može dati smisao i ovom operatoru primijenjenom na strukture).

Primijetimo da se sortiranje u navedenom primjeru obavlja samo prema vrijednosti jednog polja strukture (polja "prosjek"). Kažemo da je polje "prosjek" *ključ* po kojem se obavlja sortiranje. Prirodno je postaviti šta se dešava prilikom sortiranja sa dva sloga kod kojih su ključevi *jednaki* (npr. sa dva učenika koji imaju jednak prosjek). Za takve slogove kažemo da su *neuporedivi*, i funkcija "sort" ne predviđa ništa precizno vezano za njihov međusobni poredak. Drugim riječima, svi učenici sa jednakim prosjekom zaista će biti grupirani jedan do drugog u sortiranom spisku, ali se ništa ne zna o tome kako će oni međusobno biti poredani. Čak se ne garantira da će međusobni poredak slogova sa jednakim ključevima ostati isti kao što je bio prije sortiranja. U slučaju potrebe, ovaj problem se može riješiti

proširivanjem funkcije kriterija (npr. funkcija kriterija može definirati da između dva učenika koji imaju jednak prosjek, na prvo mjesto treba doći onaj učenik čije prezime dolazi prije po abecedi). Alternativno se umjesto funkcije "sort" može koristiti funkcija "stable_sort" (iz iste biblioteke), koja je neznatno sporija, ali garantira da će elementi niza koji su neuporedivi sa aspekta funkcije kriterija ostati u sortiranom nizu u istom međusobnom poretku kakvi su bili prije sortiranja. Tako, na primjer, pretpostavimo da je niz učenika već bio sortiran po abecednom redoslijedu prezimena. Ukoliko sada sortiramo ovaj niz po prosjeku pomoću funkcije "stable_sort", učenici koji imaju jednak prosjek zadržće međusobni poredak po abecednom redoslijedu prezimena, dok taj poredak ne mora ostati sačuvan ukoliko umjesto "stable_sort" iskoristimo funkciju "sort".

Poput svih drugih promjenljivih, promjenljive strukturnog tipa se također mogu kreirati dinamički pomoću operatora "**new**". Na primjer, sljedeći programski isječak deklarira pokazivačku promjenljivu "pok_sekretar" kojoj se dodjeljuje adresa novokreirane dinamičke promjenljive tipa "Radnik". Ova pokazivačka promjenljiva se kasnije koristi za pristup novokreiranoj dinamičkoj promjenljivoj:

```
Radnik *pok_sekretar = new Radnik;
strcpy((*pok_sekretar).ime, "Ahmed Hodžić");
strcpy((*pok_sekretar).odjeljenje, "Marketing");
(*pok_sekretar).platni_broj = 34;
(*pok_sekretar).plata = 530;
```

Treba napomenuti da su u konstrukcijama poput "(*pok_sekretar).plata" zgrade *bitne*, s obzirom da operator pristupa "..." ima viši prioritet u odnosu na operator dereferenciranja "*". Stoga se izraz poput "*x.y" interpretira kao "* (x.y)". Ovakav izraz imao bi smisla kada bismo imali strukturu promjenljivu "x" koja ima polje "y" pokazivačkog tipa, koje želimo da dereferenciramo. S druge strane, u izrazu "(*x).y", imamo pokazivač "x" na neki strukturalni tip, koji želimo da dereferenciramo, i da nakon toga pristupimo atributu "y" tako dereferenciranog objekta. S obzirom da se u jeziku C++ izuzetno često susreće jezička konstrukcija oblika "(*x).y", za nju je uvedena posebna i preglednija sintaksa oblika "x->y". Stoga, umjesto konstrukcije poput "(*pok_sekretar).plata" možemo pisati i konstrukciju "pok_sekretar->plata". Slijedi da smo prethodnu sekvencu mogli preglednije napisati ovako:

```
Radnik *pok_sekretar = new Radnik;
strcpy(pok_sekretar->ime, "Ahmed Hodžić");
strcpy(pok_sekretar->odjeljenje, "Marketing");
pok_sekretar->platni_broj = 34;
pok_sekretar->plata = 530;
```

Znak "->" ponaša se također kao neovisan operator koji se naziva *operator indirektnog pristupa*, za razliku od *operatora direktnog pristupa* "...". Na ovaj način, dinamičkim strukturalnim promjenljivim možemo pristupati preko pokazivača koji na njih pokazuju na potpuno isti način kao da se radi o običnim strukturalnim promjenljivim, samo što umjesto operatora direktnog pristupa "..." trebamo koristiti operator indirektnog pristupa "->".

Može se postaviti pitanje šta se postiže kreiranjem dinamičkih strukturalnih promjenljivih. Ranije smo govorili da nema previše smisla kreirati individualne dinamičke promjenljive jednostavnih tipova, kao što je recimo tip "**int**". Međutim, kod strukturalnih promjenljivih ne mora biti tako. Strukturne promjenljive mogu biti prilično masivne, tako da manipulacije sa njima mogu biti zahtjevne. Na primjer, uz pretpostavku da jedan cijeli broj zauzima 4 bajta, a realni broj 8 bajtova, jedan konkretan primjerak promjenljive tipa "Ucenik" zauzima 109 bajtova u memoriji, što nije teško izračunati. Sada, ukoliko na primjer treba kopirati promjenljivu "u1" tipa "Ucenik" u promjenljivu "u2" istog tipa, potrebno je

kopirati svih 109 bajtova. S druge strane, ukoliko su “pu1” i “pu2” *pokazivači* na dvije dinamičke promjenljive tipa “Ucenik”, umjesto da kopiramo samu dinamičku promjenljivu “*pu1” u dinamičku promjenljivu “*pu2”, isti efekat možemo ostvariti kopirajući samo pokazivač “pu1” u pokazivač “pu2”, pri čemu se kopiraju svega 4 bajta (uz pretpostavku da pokazivač zauzima toliko). Slijedi da se sve manipulacije sa struktturnim promjenljivim mogu učiniti mnogo efikasnije ukoliko umjesto sa samim struktturnim promjenljivim manipuliramo sa *pokazivačima koje na njih pokazuju*. Izmjene koje za ovu svrhu treba učiniti uglavnom su posve minorne.

Opisanu ideju iskoristićemo za poboljšanje efikasnosti prethodnog programa koji manipulira sa strukturama koje opisuju učenike. Pošto je sortiranje postupak koji zahtijeva intenzivnu razmjenu elemenata, možemo mnogo dobiti na efikasnosti ukoliko umjesto niza elemenata tipa “Ucenik” upotrijebimo niz pokazivača na dinamički kreirane objekte tipa “Ucenik”. Ova ideja iskorištena je u sljedećem programu:

```
#include <iostream>
#include <iomanip>
#include <algorithm>

using namespace std;

const int MaxBrojUcenika(30), BrojPredmeta(12);

struct Datum {
    int dan, mjesec, godina;
};

struct Ucenik {
    char ime[20], prezime[20];
    Datum datum_rodjenja;
    int ocjene[BrojPredmeta];
    double prosjek;
    bool prolaz;
};

int main() {
    void UnesiUcenike(Ucenik *ucenici[], int broj_ucenika);
    void ObradiUcenike(Ucenik *ucenici[], int broj_ucenika);
    void IspisiIzvjestaj(Ucenik *ucenici[], int broj_ucenika);
    void OslobodiMemoriju(Ucenik *ucenici[], int broj_ucenika);
    int broj_ucenika;
    cout << "Koliko ima učenika: ";
    cin >> broj_ucenika;
    Ucenik *ucenici[MaxBrojUcenika] = {};// Inicijalizacija nulama
    try {
        UnesiUcenike(ucenici, broj_ucenika);
        ObradiUcenike(ucenici, broj_ucenika);
        IspisiIzvjestaj(ucenici, broj_ucenika);
    }
    catch(...) {
        cout << "Problemi sa memorijom...\n";
    }
    OslobodiMemoriju(ucenici, broj_ucenika);
    return 0;
}

void UnesiUcenike(Ucenik *ucenici[], int broj_ucenika) {
    void UnesiJednogUcenika(Ucenik *ucenik);
    for(int i = 0; i < broj_ucenika; i++) {
        cout << "Unesite podatke za " << i + 1 << ". učenika:\n";
        UnesiJednogUcenika(ucenik);
    }
}
```

```

        ucenici[i] = new Ucenik;
        UnesiJednogUcenika(ucenici[i]);
    }
}

void UnesiJednogUcenika(Ucenik *ucenik) {
    void UnesiDatum(Datum &datum);
    void UnesiOcjene(int ocjene[], int broj_predmeta);
    cout << " Ime: "; cin >> ucenik->ime;
    cout << " Prezime: "; cin >> ucenik->prezime;
    UnesiDatum(ucenik->datum_rodjenja);
    UnesiOcjene(ucenik->ocjene, BrojPredmeta);
}
void UnesiDatum(Datum &datum) {
    cout << " Dan rodenja: "; cin >> datum.dan;
    cout << " Mjesec rodenja: "; cin >> datum.mjesec;
    cout << " Godina rodenja: "; cin >> datum.godina;
}
void UnesiOcjene(int ocjene[], int broj_predmeta) {
    for(int i = 0; i < broj_predmeta; i++) {
        cout << " Ocjena iz " << i + 1 << ". predmeta: ";
        cin >> ocjene[i];
    }
}
void ObradiUcenike(Ucenik *ucenici[], int broj_ucenika) {
    void ObradiJednogUcenika(Ucenik *ucenik);
    bool DaLiJeBoljiProsjek(const Ucenik *u1, const Ucenik *u2);
    for(int i = 0; i < broj_ucenika; i++)
        ObradiJednogUcenika(ucenici[i]);
    sort(ucenici, ucenici + broj_ucenika, DaLiJeBoljiProsjek);
}
void ObradiJednogUcenika(Ucenik *ucenik) {
    double suma_ocjena();
    ucenik->prosjek = 1; ucenik->prolaz = false;
    for(int i = 0; i < BrojPredmeta; i++) {
        if(ucenik->ocjene[i] == 1) return;
        suma_ocjena += ucenik->ocjene[i];
    }
    ucenik->prolaz = true;
    ucenik->prosjek = suma_ocjena / BrojPredmeta;
}
bool DaLiJeBoljiProsjek(const Ucenik *u1, const Ucenik *u2) {
    return u1->prosjek > u2->prosjek;
}
void IspisiIzvjestaj(Ucenik *ucenici[], int broj_ucenika) {
    void IspisiJednogUcenika(const Ucenik *ucenik);
    cout << endl;
    for(int i = 0; i < broj_ucenika; i++)
        IspisiJednogUcenika(ucenici[i]);
}
void IspisiJednogUcenika(const Ucenik *ucenik) {
    void IspisiDatum(const Datum &datum);
    cout << "Ucenik " << ucenik->ime << " " << ucenik->prezime
        << " roden ";
}

```

```

IspisiDatum(ucenik->datum_rodjenja);
if(ucenik->prolaz)
    cout << " ima prosjek " << setprecision(2) << ucenik->prosjek;
else
    cout << " mora ponavljati razred";
    cout << endl;
}

void IspisiDatum(const Datum &datum) {
    cout << datum.dan << "." << datum.mjesec << "." << datum.godina;
}

void OslobodiMemoriju(Ucenik *ucenici[], int broj_ucenika) {
    for(int i = 0; i < broj_ucenika; i++) delete ucenici[i];
}

```

Može se primijetiti da su izmjene koje su izvršene u odnosu na prethodnu verziju minorne, i uglavnom se svode na zamjenu operatora “.” operatorom “->” tamo gdje je to neophodno. Pored toga, unutar funkcije ”UnesiUcenike” vrši se dinamička alokacija memorije za svakog učenika posebno, prije nego što se pokazivač na novokreiranog učenika prosljedi funkciji ”UnesiJednogUcenika”. Napomenimo da se ovoj funkciji nije morao prosljediti *pokazivač* na novokreiranog učenika, već sam novokreirani učenik, konstrukcijom poput ”UnesiJednogUcenika(*ucenici[i])”, dakle uz jedno dodatno dereferenciranje. Naravno, u tom slučaju funkcija ”UnesiJednogUcenika” ne bi kao svoj formalni parametar trebala imati *pokazivač* na objekat tipa ”Ucenik”, već sam objekat tipa učenik (ili referencu na njega). Drugim riječima, funkcija ”UnesiJednogUcenika” bi tada izgledala potpuno isto kao u prethodnom programu. Koja će se od ove dvije varijante koristiti, stvar je stila. Iste primjedbe vrijede i za funkcije ”ObradiJednogUcenika” i ”IspisiJednogUcenika”. Naravno, ovaj program predviđa i funkciju za brisanje svih alociranih učenika, kao i hvatanje izuzetaka koji bi eventualno mogli nastati uslijed memorijskih problema. Čitatelju odnosno čitateljki se savjetuje da temeljito analiziraju i međusobno uporede prethodna dva programa, s obzirom da je razumijevanje izloženih koncepata od ključne važnosti za razumijevanje materije koja slijedi dalje u ovom i narednim poglavljima.

U prethodna dva primjera, koristili smo *staticke* nizove ”ucenici” čiji su elementi strukture tipa ”Ucenik”, odnosno pokazivači na strukture tipa ”Ucenik”. Nema nikakvog razloga da i nizovi struktura odnosno nizovi pokazivača na strukture ne mogu biti dinamički. U slučaju da želimo koristiti dinamički niz struktura tipa ”Ucenik”, za pristup elementima takvog dinamičkog niza promjenljivu ”ucenici” bi trebalo definirati kao *pokazivač na strukturu* ”Ucenik”, odnosno kao

```
Ucenik *ucenici;
```

nakon čega bismo, kada saznamo koliko zaista ima učenika, trebali izvršiti alokaciju dinamičkog niza učenika i dodijeliti promjenljivoj ”ucenici” adresu prvog elementa takvog niza pomoću naredbe

```
ucenici = new Ucenik[broj_ucenika];
```

U slučaju da želimo koristiti dinamički niz pokazivača na strukture tipa ”Ucenik”, promjenljivu ”ucenici” bi trebalo deklarirati kao *pokazivač na pokazivač na strukturu* ”Ucenik” (tj. kao dvojni pokazivač) pomoću deklaracije

```
Ucenik **ucenici;
```

dok bismo samu dinamičku alokaciju niza pokazivača izvršili pomoću naredbe

```
ucenici = new Ucenik*[broj_ucenika];
```

Uz ove izmjene, sve ostalo bi u ova dva programa, zahvaljujući činjenici da se na pokazivače može primjenjivati indeksiranje, moglo ostati isto, osim što bi na kraj programa trebalo dodati i naredbu

```
delete[] ucenici;
```

Pored toga, trebalo bi predvidjeti i hvatanje izuzetka koji eventualno može baciti operator “**new**“ u slučaju da dinamička alokacija ovog niza ne uspije.

Činjenica da se, za razliku od nizova, promjenljive strukturnog tipa mogu *dodjeljivati jedna drugoj* pomoću operatora dodjele “=”, zatim prenosi u funkcije *po vrijednosti* (uz potpuno kopiranje sadržaja), kao i da se strukture *mogu vraćati kao rezultati iz funkcije*, dovodi do zanimljive ideje da se nizovi “umotaju” u strukture, tj. da se napravi struktura čiji je jedini atribut niz. Tako “umotan” niz može se koristiti poput strukture, tj. prenosi u funkcije po vrijednosti, vraćati kao rezultat iz funkcije, itd. Na primjer, pretpostavimo da imamo sljedeće deklaracije:

```
struct Niz {  
    int elementi[10];  
};  
  
Niz a, b;
```

Činjenica da je “Niz“ zapravo struktura a ne niz, donekle komplicira pristup elementima ovog, nazovimo ga tako, kvazi-niza. Na primjer, da popunimo sve elemente kvazi-niza “a“ nulama, morali bismo pisati konstrukciju poput

```
for(int i = 0; i < 10; i++) a.elementi[i] = 0;
```

Drugim riječima, morali bismo eksplisitno naglašavati da pristupamo atributu “elementi“ strukture “a“ (zapravo, tehnika preklapanja operatora omogućava da se i ovaj problem prevaziđe, međutim još nije vrijeme da o tome govorimo). S druge strane, ovako kompliciranje se ponekad isplati. Na primjer, sada nam je omogućeno da pišemo konstrukcije poput “b = a“ sa ciljem kopiranja *svih* elemenata iz kvazi-niza “a“ u kvazi-niz “b“. Također, moguće je “a“ i “b“ prenijeti u neku funkciju *po vrijednosti*, tako da u slučaju da funkcija *izmjeni* sadržaj formalnog parametra tipa “Niz“, ta promjena ne ostavi nikakav utjecaj na sadržaj strukturnih promjenljivih “a“ i “b“. Sa običnim nizovima, koji se uvijek ponašaju kao da se prenose po referenci, ovako nešto ne bi bilo moguće. Naime, ukoliko funkcija kvari niz prenesen kao parametar, jedini način da sačuvamo niz koji je prenesen kao stvarni parametar je da formiramo pomoćni niz, i da unutar funkcije radimo sa njim umjesto sa izvornim nizom. Dalje, moguće je *vratiti iz funkcije kao rezultat* neki objekat tipa “Niz“. Ovim smo došli do osnovne ideje za povećanje fleksibilnosti nizovnih tipova podataka. U nastavku teksta slijede dalja produbljivanja ove ideje.

Moguće je formirati i *generičke strukture*, odnosno strukture kod kojih tipovi određenih atributa nisu unaprijed poznati. Ovakve strukture se također deklariraju uz pomoć *šablonu*, odnosno ključne riječi “**template**“. Slijedi jedan vrlo jednostavan primjer deklaracije generičke strukture, kao i konkretnih primjeraka objekata ovakvih struktura:

```
template <typename Tip>  
struct GenerickiNiz {  
    Tip elementi[10];  
};  
  
GenerickiNiz<double> a, b;  
GenerickiNiz<int> c;
```

Primijetimo da se, prilikom deklariranja konkretnih primjeraka “*a*”, “*b*” i “*c*” generičke strukture “*GenerickiNiz*”, obavezno unutar šiljastih zagrada “*<>*” treba specificirati značenje formalnih parametara šablonu, odnosno smisao nepoznatih tipova upotrijebljenih unutar strukture. Drugim riječima, za razliku od generičkih funkcija, ovdje nisu moguće automatske dedukcije tipova. Također je bitno napomenuti da samo ime generičke strukture (u ovom primjeru “*GenerickiNiz*”) ne predstavlja tip, odnosno ne postoje objekti tipa “*GenerickiNiz*”. Tip dobijamo tek nakon specifikacije parametara šablonu, tako da konstrukcije poput “*GenerickiNiz<double>*” ili “*GenerickiNiz<int>*” jesu tipovi. Pri tome, konkretni tipovi dobijeni iz istog šablonu uz različite parametre šablonu predstavljaju različite tipove. Tako su, u prethodnom primjeru, promjenljive “*a*” i “*b*” istog tipa, ali koji je različit od tipa promjenljive “*c*”. Stoga je, na primjer, dodjela poput “*a = b*” legalna, a dodjela “*a = c*” nije.

Do sada smo uvijek kada smo radili sa nizovima, u funkciju kao parametar obavezno prenosili ne samo niz, nego i broj elemenata niza (koji funkcija ne bi drugačije mogla saznati). U slučaju matrica, morali smo pored same matrice prenositi i broj njenih redova, odnosno kolona. Sve ovo se može izbjegći ukoliko formiramo strukturu čiji će atributi biti ne samo sam niz, nego i stvarni broj njegovih elemenata (odnosno, u slučaju matrice, stvarni broj redova i kolona). Na ovaj način, sve informacije koje opisuju niz (njegovi elementi i dimenzije) upakovane su u jednu strukturu, koju možemo prenijeti kao jedinstven parametar u funkciju, pa čak i vratiti kao rezultat iz funkcije. Ovaj pristup ilustriran je u sljedećem programu, koji od korisnika traži da unese dvije matrice istog formata (broj redova i kolona također zadaje korisnik), a zatim račina i ispisuje zbir dvije unesene matrice. Radi veće fleksibilnosti, struktura “*Matrica*” deklarirana je kao generička struktura, sa neodređenim tipom elemenata matrice. Posljedica ove činjenice je da sve funkcije koje rade sa ovakvom strukturom moraju biti ili generičke funkcije (ukoliko žele da rade sa najopštijom formom generičke strukture “*Matrica*”), ili se moraju ograničiti isključivo na rad sa nekim konkretnim tipom izvedenim iz generičke strukture “*Matrica*” (npr. tipom “*Matrica<double>*”). Ovdje je prikazana univerzalnija varijanta, koja koristi generičke funkcije. Uporedite ovaj program sa sličnim programom u poglavljju o višedimenzionalnim nizovima. Ova dva programa su, zanemarimo li neke sitne detalje, funkcionalno ekvivalentna, samo što je lako uočiti da se u verziji koja slijedi funkcije pozivaju i koriste na mnogo elegantniji način.

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

using namespace std;

const int MaxDimenzija(10);

template <typename Tip>
struct Matrica {
    int broj_redova, broj_kolona;
    Tip elementi[MaxDimenzija][MaxDimenzija];
};

template <typename Tip>
void PostaviDimenzije(Matrica<Tip> &m, int br_redova, int br_kolona) {
    if(br_redova < 1 || br_kolona < 1)
        throw "Dimenzije matrice moraju biti pozitivne!\n";
    if(br_redova > MaxDimenzija || br_kolona > MaxDimenzija)
        throw "Dimenzije matrice su prevelike!\n";
    m.broj_redova = br_redova; m.broj_kolona = br_kolona;
}

template <typename Tip>
void UnesiMatricu(const char ime_matrice[], Matrica<Tip> &mat) {
    for(int i = 0; i < mat.broj_redova; i++)
        for(int j = 0; j < mat.broj_kolona; j++) {
            cout << ime_matrice << "(" << i + 1 << ", " << j + 1 << ") = ";
            cin >> mat.elementi[i][j];
        }
}
```

```

        cin >> mat.elementi[i][j];
    }
}

template <typename Tip>
void IspisiMatricu(const Matrica<Tip> &mat, int sirina_ispisa) {
    for(int i = 0; i < mat.broj_redova; i++) {
        for(int j = 0; j < mat.broj_kolona; j++)
            cout << setw(sirina_ispisa) << mat.elementi[i][j];
        cout << endl;
    }
}

template <typename Tip>
Matrica<Tip> ZbirMatrica(const Matrica<Tip> &m1,
    const Matrica<Tip> &m2) {
    if(m1.broj_redova != m2.broj_redova
        || m1.broj_kolona != m2.broj_kolona)
        throw "Matrice nemaju jednake dimenzije!\n";
    Matrica<Tip> m3;
    PostaviDimenzije(m3, m1.broj_redova, m1.broj_kolona);
    for(int i = 0; i < m1.broj_redova; i++)
        for(int j = 0; j < m1.broj_kolona; j++)
            m3.elementi[i][j] = m1.elementi[i][j] + m2.elementi[i][j];
    return m3;
}

int main() {
    Matrica<double> a, b;
    int m, n;
    cout << "Unesi broj redova i kolona za matrice:\n";
    cin >> m >> n;
    try {
        PostaviDimenzije(a, m, n); PostaviDimenzije(b, m, n);
        cout << "Unesi matricu A:\n";
        UnesiMatricu("A", a);
        cout << "Unesi matricu B:\n";
        UnesiMatricu("B", b);
        cout << "Zbir ove dvije matrice je:\n";
        IspisiMatricu(ZbirMatrica(a, b), 7);
    }
    catch(const char poruka[]) {
        cout << poruka;
    }
    return 0;
}
}

```

U ovom programu je za pamćenje elemenata matrice unutar generičke strukture “`Matrica`“ definiran statički dvodimenzionalni niz “`elementi`”, sa fiksnim dimenzijama, dok se stvarne dimenzije pamte u atributima “`broj_redova`” i “`broj_kolona`”. Postavljanje ovih atributa povjerenje je funkciji “`PostaviDimenzije`”. Prednost njihovog postavljanja pomoću ove funkcije umjesto direktnog postavljanja je u činjenici da ova funkcija može da provjeri da li su željene dimenzije matrice veće od deklariranih dimenzija, i da baci izuzetak ukoliko jesu. Na taj način se sprečava da se greškom postave dimenzije matrice koje su veće od prostora rezerviranog za smještanje elemenata matrice.

Izloženi primjer jasno ilustrira zbog čega nije preporučljivo parametre strukturnog tipa prenositi u funkcije po vrijednosti, ukoliko to nije zaista neophodno. Naime, u prethodnom primjeru, uz prepostavku da se za pamćenje realnih brojeva koristi 8 bajta memorijskog prostora, samo za pamćenje elemenata

matrice u strukturi "Matrica" troši se $8 \cdot 10 \cdot 10 = 800$ bajta, a nekoliko bajtova se troši i na pamćenje dimenzija matrice (tako da vjerovatno memorijsko zauzeće jednog objekta tipa "Matrica" iznosi 808 bajta). Kada bi se koristio prenos parametra po vrijednosti, prilikom svakog pozivanja funkcije, svih 808 bajta stvarnog parametra bi se bespotrebno kopirali u formalni parametar funkcije, što je svakako trošak vremena, a i memorije, s obzirom da se ista informacija čuva na dva mesta (u formalnom i stvarnom parametru). Utrošak vremena i memorije bio bi još veći da su deklarirane dimenzije matrice bile veće.

U prethodnom primjeru, struktura "Matrica" zauzima mnogo memorije, i to bez obzira na stvarne dimenzije matrice, zbog toga što je prostor za pamćenje elemenata matrice definiran *statički*, sa fiksnim dimenzijama. Bolju varijantu prethodnog programa možemo dobiti ukoliko se odlučimo da matrice definiramo *dinamički*. Na taj način, struktura "Matrica" ne treba da pamti sve elemente matrice, već samo *pokazivač na njih*, dok će se prostor za smještanje elemenata matrice dodijeliti dinamički, na zahtjev, onog trenutka kada postanu poznate stvarne dimenzije matrice. U tom slučaju je veličina matrica sa kojima možemo raditi ograničena samo količinom raspoložive memorije.

Opisana ideja iskorištena je u sljedećem programu. Većina funkcija u njemu su identične ili veoma slične funkcijama iz prethodnog programa, samo se umjesto funkcije "PostaviDimenzije" koristi funkcija "StvoriMaticu" koja kreira matricu dinamički, i vraća kao rezultat strukturu "Matrica", koja pored podataka o dimenzijama matrice sadrži pokazivač pomoću kojeg se može pristupiti elementima dinamički kreirane matrice. Funkcija "StvoriMaticu" je nešto složenija, zbog toga što vodi brigu o tome da ne dovede do curenja memorije ukoliko u procesu kreiranja matrice ponestane memorije, o čemu smo ranije detaljno govorili. Pored funkcije "StvoriMaticu", u ovom programu se koristi i funkcija "UnistiMaticu", čiji je zadatak oslobađanje memorije. Primijetimo da u ovom slučaju, za razliku od slične funkcije koju smo napisali prilikom demonstracije dinamičke alokacije i dealokacije višedimenzionalnih nizova, nije potrebno u funkciju prenosititi podatke o dimenzijama matrice, s obzirom da su ti podaci već sadržani u strukturi "Matrica" koja se prosljeđuje ovoj funkciji kao parametar.

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

using namespace std;

template <typename Tip>
struct Matrica {
    int broj_redova, broj_kolona;
    Tip **elementi;
};

template <typename Tip>
void UnistiMaticu(Matrica mat) {
    if(mat.elementi == 0) return;
    for(int i = 0; i < mat.broj_redova; i++) delete[] mat.elementi[i];
    delete[] mat.elementi;
}

template <typename Tip>
Matrica<Tip> StvoriMaticu(int broj_redova, int broj_kolona) {
    Matrica<Tip> mat;
    mat.broj_redova = broj_redova; mat.broj_kolona = broj_kolona;
    mat.elementi = new Tip*[broj_redova];
    for(int i = 0; i < broj_redova; i++) mat.elementi[i] = 0;
    try {
        for(int i = 0; i < broj_redova; i++)
            mat.elementi[i] = new Tip[broj_kolona];
    }
}
```

```

    }
    catch(...) {
        UnistiMatricu(mat);
        throw;
    }
    return mat;
}

template <typename Tip>
void UnesiMatricu(const char ime_matrice[], Matrica<Tip> &mat) {
    for(int i = 0; i < mat.broj_redova; i++)
        for(int j = 0; j < mat.broj_kolona; j++) {
            cout << ime_matrice << "(" << i + 1 << "," << j + 1 << ") = ";
            cin >> mat.elementi[i][j];
        }
}

template <typename Tip>
void IspisiMatricu(const Matrica<Tip> &mat, int sirina_ispisa) {
    for(int i = 0; i < mat.broj_redova; i++) {
        for(int j = 0; j < mat.broj_kolona; j++)
            cout << setw(sirina_ispisa) << mat.elementi[i][j];
        cout << endl;
    }
}

template <typename Tip>
Matrica<Tip> ZbirMatrica(const Matrica<Tip> &m1,
    const Matrica<Tip> &m2) {
    if(m1.broj_redova != m2.broj_redova
        || m1.broj_kolona != m2.broj_kolona)
        throw "Matrice nemaju jednake dimenzije!\n";
    Matrica<Tip> m3 = StvoriMatricu<Tip>(m1.broj_redova, m1.broj_kolona);
    for(int i = 0; i < m1.broj_redova; i++)
        for(int j = 0; j < m1.broj_kolona; j++)
            m3.elementi[i][j] = m1.elementi[i][j] + m2.elementi[i][j];
    return m3;
}

int main() {
    Matrica<double> a = {0, 0, 0}, b = {0, 0, 0}, c = {0, 0, 0};
    int m, n;
    cout << "Unesi broj redova i kolona za matrice:\n";
    cin >> m >> n;
    try {
        a = StvoriMatricu<double>(m, n);
        b = StvoriMatricu<double>(m, n);
        cout << "Unesi matricu A:\n";
        UnesiMatricu("A", a);
        cout << "Unesi matricu B:\n";
        UnesiMatricu("B", b);
        cout << "Zbir ove dvije matrice je:\n";
        IspisiMatricu(c = ZbirMatrica(a, b), 7);
    }
    catch(...) {
        cout << "Nema dovoljno memorije!\n";
    }
    UnistiMatricu(a); UnistiMatricu(b); UnistiMatricu(c);
    return 0;
}

```

}

Vidimo da je dobijeni program također vrlo elegantan, poput prethodnog programa. Međutim, u njemu se javljaju neki detalji koji nisu postojali u prethodnom programu. Prvo, ovdje smo pored matrica "a" i "b" definirali i matricu "c", a umjesto naredbe

```
IspisiMatricu(ZbirMatrica(a, b), 7);
```

koju smo imali u prethodnom programu, ovdje se javlja naredba

```
IspisiMatricu(c = ZbirMatrica(a, b), 7);
```

koja je funkcionalno ekvivalentna slijedu od dvije naredbe

```
c = ZbirMatrica(a, b);
IspisiMatricu(c, 7);
```

Postavlja se pitanje zašto smo uopće morali definirati promjenljivu "c". Problem je u tome što funkcija "ZbirMatrica" dinamički kreira novu matricu (pozivom funkcije "StvoriMatricu") i kao rezultat vraća strukturu koja sadrži pokazivač na zauzeti dio memorije. Ova struktura bi se mogla neposredno prenijeti u funkciju "IspisiMatricu" bez ikakve potrebe za pamćenjem vraćene strukture u promjenljivoj "c". Međutim, na taj način bismo *izgubili pokazivač* na zauzeti dio memorije, i ne bismo kasnije imali priliku da oslobodimo zauzeti dio memorije (pozivom funkcije "UnistiMatricu"). Naime, problem je u tome što se sav dinamički zauzet prostor mora eksplisitno obrisati upotrebom operatora "**delete**". Ova potreba da se eksplisitno brinemo o brisanju svakog dinamičkog objekta koji je kreiran, može nam zadati mnogo glavobolja ako ne želimo (a svakako ne bismo trebali da želimo) da uzrokujemo neprestano curenje memorije. Kasnije ćemo vidjeti da se ovaj problem može riješiti primjenom tzv. *destruktora* koji na sebe mogu automatski preuzeti brigu za brisanje memorije koju je neki objekat zauzeo onog trenutka kada taj objekat više nije potreban.

Drugi detalj koji upada u oči je da smo u ovom programu na samom početku inicijalizirali sva polja u matricama "a", "b" i "c" na nule (zapravo, neophodno je samo da polje "elementi" bude inicijalizirano na *nul-pokazivač*). Ovo je urađeno zbog sljedećeg razloga. Na kraju programa potrebno je eksplisitno *uništiti* sve tri matrice "a", "b" i "c" (tj. osloboditi prostor koji je rezerviran za smještanje njihovih elemenata). Pretpostavimo sada da stvaranje matrice "a" uspije, ali da prilikom stvaranja matrice "b" dođe do bacanja izuzetka (npr. zbog nedovoljne količine raspoložive memorije). Izuzetak će biti uhvaćen u "**catch**" bloku, ali stvorenu matricu "a" treba obrisati. Njeno brisanje će se ionako desiti nakon "**catch**" bloka pozivom funkcije "UnistiMatricu". Međutim, šta je sa matricama "b" i "c"? Naredna dva poziva funkcije "UnistiMatricu" trebaju da unište i njih, ali one nisu ni stvorene! Međutim, ukoliko pažljivije pogledamo funkciju "UnistiMatricu", vidjećemo da ona *ne radi ništa* u slučaju da polje "elementi" u matrici sadrži *nul-pokazivač*. Kako smo na početku polja "elementi" u sve tri matrice inicijalizirali na nulu, one matrice koje nisu ni stvorene i dalje će imati nul-pokazivač u ovom polju, tako da funkcija "UnistiMatricu" neće nad njima ništa ni uraditi. Da nismo prethodno izvršili inicijalizaciju polja "elementi" na nul-pokazivač, mogli bi nastati veliki problemi ukoliko funkciji "UnistiMatricu" prosljedimo matricu koja nije ni stvorena (tj. za čije elemente nije alociran prostor). Naime, pokazivač "elementi" bi imao neku *slučajnu vrijednost* (jer sve klasične promjenljive koje nisu inicijalizirane imaju slučajne početne vrijednosti), pa bi unutar funkcije "UnistiMatricu" operator "**delete**" bio primijenjen nad pokazivačima za koje je potpuno neizvjesno na šta pokazuju (najvjerojatnije ni na šta smisleno). Stoga je ishod ovakvih akcija posve nepredvidljiv (i, najvjerojatnije, ne vodi ničemu dobrom). Stoga, da nismo ručno inicijalizirali polja "elementi" na nul-pokazivače, morali bismo koristiti ugniježdene "**try**" – "**catch**" strukture, što je, kao što smo već vidjeli, veoma nelegantno i nezgrapno.

Očigledno je da svi problemi ovog tipa nastaju zbog činjenice da promjenljive, uključujući i polja unutar struktura, na početku imaju nedefinirane vrijednosti. Uskoro ćemo vidjeti kako se uz pomoć tzv. *konstruktora* može ovaj problem riješiti kreiranjem strukturnih tipova čija se polja automatski inicijaliziraju pri deklaraciji odgovarajućih promjenljivih, bez potrebe da programer eksplisitno vodi računa o propisnoj inicijalizaciji.

Ovom prilikom je neophodno ukazati na još jednu pojavu koja može nastati kod upotrebe struktura koje kao svoja polja sadrže pokazivače. Prepostavimo da imamo strukturu “*Matrica*“ deklariranu kao u prethodnom programu, i da smo izvršili sljedeću sekvencu naredbi:

```
Matrica<double> a, b;
a = StvoriMatricu<double>(10, 10);
b = a;
a.elementi[5][5] = 13;
b.elementi[5][5] = 18;
cout << a.elementi[5][5];
```

Mada bi se moglo očekivati da će ovaj program ispisati broj 13, on će zapravo ispisati broj 18! Ovo se ne bi desilo da je polje “elementi“ u strukturi “*Matrica*“ deklarirano kao običan dvodimenzionalni niz. Šta se zapravo desilo? Kada se jedna struktura kopira u drugu pomoću znaka dodjeljivanja “=”, kopiraju se sva polja iz jedne strukture u drugu (i ništa drugo). Međutim, treba obratiti pažnju da polje “elementi“ nije *niz* nego *pokazivač*, tako da se prilikom kopiranja polja iz strukture “*a*“ u strukturu “*b*“ kopira samo *pokazivač*, a ne ono na šta on pokazuje. Stoga, nakon obavljenog kopiranja, obje strukture “*a*“ i “*b*“ sadrže polja nazvana “elementi“ koja sadrže *istu vrijednost*, odnosno pokazuju na *istu adresu* tj. *isti dinamički niz*! Drugim riječima, kopiranjem polja iz “*a*“ u “*b*“ ne stvara se novi dinamički niz, nego imamo *jedan dinamički niz sa dva pokazivača koja pokazuju na njega* (jedan u strukturi “*a*”, a drugi u strukturi “*b*”). Stoga je jasno da se bilo koji pristup dinamičkom nizu preko pokazivača “*a.elementi*“ i “*b.elementi*“ odnose na *isti dinamički niz*! Dinamički niz *nije element strukture* nego se nalazi *izvan nje*, i ne kopira se zajedno sa strukturom!

Činjenica da se prilikom kopiranja struktura koje sadrže pokazivače iz jedne u drugu kopiraju samo pokazivači, a ne i ono na šta oni pokazuju naziva se *plitko kopiranje*, a dobijene kopije nazivamo *plitkim kopijama*. Plitko kopiranje obično ne pravi neke probleme ukoliko ga imamo u vidu (tj. sve dok imamo u vidu činjenicu da dobijamo plitke kopije), ali djeluje donekle suprotno intuitivnom poimanju kako bi dodjeljivanje trebalo da radi. Naime, nakon obavljenog plitkog kopiranja strukturne promjenljive “*a*“ u promjenljivu “*b*“, promjenljiva “*b*“ se ponaša više poput *reference* na promjenljivu “*a*“ nego poput njene kopije. Ovakvo ponašanje je u nekim programskim jezicima posve uobičajeno (npr. u jeziku Java dodjeljivanjem strukturne promjenljive “*a*“ promjenljivoj “*b*“, promjenljiva “*b*“ zapravo postaje referenca na promjenljivu “*a*“), ali ne i u jeziku C++. Doduše, mnogi teoretičari objektno orijentiranog programiranja (o kojem ćemo govoriti u narednim poglavljima) smatraju da je za strukturne tipove plitko kopiranje prirodnije (o ovome ćemo diskutirati kasnije), ali autor ovog materijala ne dijeli to mišljenje. Kasnije ćemo vidjeti da se problem plitkog kopiranja može riješiti uz pomoć preklapanja operatora tako da se operatoru “=” promijeni značenje tako da obavlja kopiranje ne samo pokazivača unutar strukture, nego i dinamičkih elemenata na koje pokazivači pokazuju (tzv. *duboko kopiranje*).

Na ovom mjestu trebamo ipak ukazati na jednu potencijalnu opasnost uslijed korištenja plitkih kopija (koja će posebno doći do izražaja kada se upoznamo sa pojmom destruktora). Posmatrajmo sljedeći programski isječak:

```
Matrica<double> a, b;
a = StvoriMatricu<double>(10, 10);
b = a;
```

```
UnistiMatricu(b);
```

Mada je cilj poziva “`UnistiMatricu(b)`” vjerovatno trebao biti uništavanje matrice “`b`” (tj. oslobođanje prostora zauzetog za njene elemente), ovaj poziv će se odraziti i na matricu “`a`”. Naime, kako pokazivač “`elementi`” u obje matrice pokazuje na isti memorijski prostor, nakon oslobođanja zauzetog prostora pozivom “`UnistiMatricu(b)`”, pokazivač “`elementi`” u matrici “`a`” će pokazivati na upravo oslobođeni prostor, odnosno postaće divlji pokazivač. Time je uništavanje matrice “`b`” efektivno uništilo i matricu “`a`”, što je posljedica činjenice da “`b`” zapravo nije prava kopija matrice “`a`” (već nešto nalik na referencu na nju). Ovaj primjer pokazuje da pri radu sa strukturama koje kao svoje elemente imaju pokazivače trebamo uvijek biti na oprezu.

Plitko kopiranje ne nastaje samo prilikom dodjeljivanja jedne strukturne promjenljive drugoj, nego i prilikom prenosa *po vrijednosti* struktturnih promjenljivih kao parametara u funkcije, kao i prilikom *vraćanja struktura* kao rezultata iz funkcije. Na primjer, posmatrajmo sljedeću funkciju:

```
template <typename Tip>
void AnulirajMatricu(Matrica<Tip> mat) {
    for(int i = 0; i < mat.broj_redova; i++)
        for(int j = 0; j < mat.broj_kolona; j++)
            mat.elementi[i][j] = 0;
}
```

Ova funkcija će postaviti sve elemente matrice koja joj se proslijedi kao parametar na nulu. Međutim, na prvi pogled, ova funkcija *ne bi trebala to da uradi*, s obzirom da joj se parametar prenosi *po vrijednosti*. Zar formalni parametar “`mat`” nije samo *kopija* stvarnog parametra prenesenog u funkciju? Naravno da jeste, ali razmotrimo šta je zapravo ta kopija. Ona sadrži kopiju dimenzija matrice proslijedene kao stvarni argument, i kopiju pokazivača na njene elemente. Međutim, ta kopija pokazivača pokazuje na *iste elemente* kao i izvorni pokazivač, odnosno formalni parametar “`mat`” predstavlja *plitku kopiju* stvarnog argumenta. Zbog toga je pristup elementima matrice preko polja “`elementi`” unutar parametra “`mat`” ujedno i pristup elementima izvorne matrice. Drugim riječima, mada je parametar zaista prenesen po vrijednosti, izgleda kao da funkcija *mijenja* sadržaj stvarnog parametra (mada ona zapravo mijenja elemente matrice koji u suštini uopće nisu sastavni dio stvarnog parametra, već se nalaze izvan njega). U ovom slučaju ponovo imamo situaciju koja intuitivno odudara od očekivanog ponašanja pri prenosu parametara po vrijednosti (ovdje je ponašanje skoro istovjetno kao da je parametar prenesen *po referenci*). Time dolazimo u sličnu situaciju kao prilikom prenosa nizova kao parametara – nizovi preneseni u funkcije *po vrijednosti* ponašaju se kao da su preneseni *po referenci*. Ovakvo ponašanje uzrokovan je plitkim kopiranjem, odnosno činjenicom da ovdje parametar zapravo ne sadrži u sebi elemente matrice (mada izgleda kao da ih sadrži) – oni se nalaze *izvan njega*.

Ovakav neintuitivan tretman prilikom prenosa po vrijednosti parametara struktturnog tipa koji sadrže pokazivače, također ne dovodi do većih problema, sve dok smo ga svjesni (kao što smo se navikli da se parametri nizovnog tipa ponašaju kao da se radi o prenosu po referenci). Međutim, ovakvo ponašanje je moguće promijeniti uz pomoć tzv. *konstruktora kopije*, koji preciziraju način kako će se tačno vršiti kopiranje struktturnih parametara prilikom prenosa po vrijednosti, i prilikom vraćanja rezultata iz funkcije. Na taj način je moguće realizirati duboko kopiranje, odnosno prenos koji će biti u skladu sa intuicijom. O ovome ćemo detaljno govoriti u kasnijim poglavljima.

Čitatelj odnosno čitateljka se sada mogu sa pravom zapitati zbog čega stalno navodimo primjere raznih probematičnih situacija uz napomenu da će problem biti riješen kasnije, umjesto da odmah ponudimo rješenje u kojem se navedeni problem ne javlja. Razlog za ovo je sljedeći: kako je teško shvatiti zbog čega nešto treba raditi onako kako bi se trebalo raditi ukoliko se prethodno ne shvati šta bi se desilo kada bi se radilo drugačije, odnosno ako bi se radilo onako kako se ne treba raditi. Također, prilično je

teško shvatiti razloge za upotrebu nekih naprednijih tehnika koji na prvi pogled djeluju komplikirano (kao što su konstruktori, destruktori, konstruktori kopije, preklapanje operatora itd.) ukoliko prethodno ne shvatimo kakvi se problemi javljaju ukoliko se ove tehnike ne koriste.

Razumije se da ni jedan strukturni tip ne može sadržavati polje koje je istog strukturnog tipa kao i struktura u kojoj je sadržano, jer bi to bila “rekurzija bez izlaza” (imali bismo “strukturu koja kao polje ima strukturu koja kao polje ima strukturu koja kao polje ima...” i tako bez kraja). Međutim, struktura može sadržavati polja koja su pokazivači na isti strukturni tip. Takve strukture nazivaju se *čvorovi* (engl. *nodes*) a odgovarajući pokazivači unutar njih koji pokazuju na primjerke istog strukturnog tipa nazivaju se *veze* (engl. *links*). Na primjer, neki čvor može biti deklariran na sljedeći način:

```
struct Cvor {
    int element;
    Cvor *vez;
};
```

Čvorovi predstavljaju osnovni gradivni element za formiranje tzv. *dinamičkih struktura podataka*, o kojima ćemo govoriti kasnije. Na ovom mjestu ćemo ilustrirati samo osnovnu ideju koja ilustrira smisao čvorova. Prepostavimo da želimo unijeti i zapamtiti slijed brojeva koji se završava nulom, ali da nam broj brojeva koji će biti uneseni nije unaprijed poznat. Prepostavimo dalje da ne želimo zauzeti više memorije od one količine koja je zaista neophodna za pamćenje unesenih brojeva (odnosno, ne želimo na primjer deklarirati neki veliki niz koji će sigurno moći prihvatići sve unesene brojeve, ali i znatno više od toga). Kako ne možemo znati unaprijed kada će biti unesena nula, ne možemo koristiti niti statičke nizove, niti vektore sa veličinom zadatom u trenutku deklaracije, niti dinamički alocirane nizove. Tip “vector” doduše nudi elegantno rješenje: možemo prvo deklarirati *prazan vektor*, a zatim pozivom operacije “push_back” dodavati na kraj unesene brojeve, jedan po jedan, i tako *proširivati* veličinu vektora. Ovo rješenje je zaista lijepo, ali dovodi do jednog suštinskog pitanja. Tip “vector” je *izvedeni tip* definiran u istoimenoj biblioteci, i on je na neki način implementiran koristeći fundamentalna svojstva jezika C++ (tj. svojstva koja nisu definirana u bibliotekama, nego koja čine samo jezgro jezika). Stoga se prirodno postavlja pitanje kako bi se slična funkcionalnost mogla ostvariti *bez korištenja tipa “vector”*. To je očigledno moguće, jer je sam tip “vector” napravljen korištenjem onih svojstava koja postoje i bez njega!

Osnovna ideja zasniva se upravo na korištenju *čvorova*. Neka, na primjer, imamo čvor deklariran kao u prethodnom primjeru. Razmotrimo sada sljedeći programski isječak:

```
Cvor *pocetak(0), *prethodni;
for(;;) {
    int broj;
    cin >> broj;
    if(broj == 0) break;
    Cvor *novi = new Cvor;
    novi->element = broj; novi->vez = 0;
    if(pocetak != 0) prethodni->vez = novi;
    else pocetak = novi;
    prethodni = novi;
}
```

U ovom isječku, pri svakom unosu novog broja, dinamički kreiramo *novi čvor*, i u njegovo polje “element” upisujemo uneseni broj. Polje “vez” čvora koji sadrži *prethodni uneseni broj* (ukoliko takav postoji, odnosno ukoliko novokreirani čvor nije prvi čvor) usmjeravamo tako da pokazuje na novokreirani čvor (za tu svrhu uveli smo pokazivačku promjenljivu “*prethodni*” koja pamti adresu čvora koji sadrži prethodno uneseni broj). Prilikom kreiranja *prvog čvora*, njegovu adresu pamtimo u pokazivaču

“*pocetak*”. Polje “*veza*” novokreiranog čvora postavljamo na *nul-pokazivač*, čime zapravo signaliziramo da iza njega ne slijedi nikakav drugi čvor. Kao ilustraciju, pretpostavimo da smo unijeli slijed brojeva 3, 6, 7, 2, 5 i 0. Nakon izvršavanja prethodnog programskog isječka, u memoriji će se formirati struktura podataka koja se može shematski prikazati kao na sljedećoj slici:

Ovako formirana struktura podataka u memoriji naziva se *jednostruko povezana lista* (engl. *single linked list*), iz očiglednog razloga. Ovim smo zaista *smjestili* sve unesene brojeve u memoriju, ali kako im možemo pristupiti? Primijetimo da pokazivač “*pocetak*” sadrži adresu prvog čvora, tako da preko njega možemo pristupiti *prvom elementu*. Međutim, ovaj čvor sadrži pokazivač na sljedeći (drugi) čvor, pa koristeći ovaj pokazivač možemo pristupiti *drugom elementu*. Dalje, drugi čvor sadrži pokazivač na treći čvor, pa preko njega možemo pristupiti i *trećem elementu*, itd. Na primjer, da ispišemo prva tri unesena elementa, možemo koristiti sljedeće konstrukcije:

```
cout << pocetak->element;
cout << pocetak->veza->element;
cout << pocetak->veza->veza->element;
```

Sada se postavlja pitanje kako ispisati *sve unesene elemente*? Za tu svrhu nam je očigledno potreban neki sistematičniji način od gore prikazanog. Nije teško vidjeti da obična “**for**” petlja u kojoj se koristi pomoći pokazivač “*p*” koji u svakom trenutku pokazuje na čvor koji sadrži *tekući element* (tj. element koji upravo treba ispisati), i koji se u svakom prolazu kroz petlju pomjera tako da pokazuje na sljedeći čvor, rješava traženi zadatak:

```
for(Cvor *p = pocetak; p != 0; p = p->veza)
    cout << p->element << endl;
```

Veoma je bitno napomenuti da fizički raspored čvorova u memoriji *uopće nije bitan*, već je bitna samo *logička veza* između čvorova, ostvarena pomoći pokazivača. Naime, mi nikada ne znamo *gdje* će tačno u memoriji biti smješten čvor konstruisan operatorom “**new**”. Mi ćemo dobiti kao rezultat ovog operatorka *adresu* gdje je čvor kreiran, ali ne postoji nikakve garancije da će se čvorovi u memoriji stvarati tako da uvijek čine rastući slijed adresa (mada je takav ishod najvjerojatniji). Tako je principijelno sasvim moguće (ali ne i mnogo vjerovatno) da stvarna memorijска slika povezane liste prikazane na prethodnoj slici zapravo izgleda kao na sljedećoj slici:

Međutim, kako se pristup čvorovima ostvaruje isključivo prateći veze, povezana lista čija fizička organizacija u memoriji izgleda ovako ponaša se isto kao i povezana lista čiji čvorovi prirodno slijede jedan drugog u memoriji.

Povezane liste su veoma fleksibilne i korisne strukture podataka, i o njima ćemo detaljnije govoriti kasnije. Međutim, već na ovom mjestu treba uočiti jedan njihov bitan nedostatak u odnosu na nizove. Naime, elementima smještenim u ovako kreiranu listu može se pristupati isključivo *sekvenčijalno*, jedan po jedan, u redoslijedu kreiranja, tako da nije moguće direktno pristupiti recimo petom čvoru a da prethodno ne pročitamo prva četiri čvora (s obzirom da se adresa petog čvora nalazi u četvrtom, adresa četvrtog u trećem, itd.). Na primjer, imali bismo velikih nevolja ukoliko bismo unesene elemente trebali ispisati recimo u *obrnutom poretku*. Također, dodatni utrošak memorije za pokazivače koji se čuvaju u svakom čvoru mogu biti nedostatak, pogotovo ukoliko sami elementi ne zauzimaju mnogo prostora. Bez obzira na ova ograničenja, postoji veliki broj primjena u kojima se elementi obrađuju upravo sekvenčijalno, i u kojima dodatni utrošak memorije nije velika smetnja, tako da u takvim primjenama ovi nedostaci ne predstavljaju bitniju prepreku.

Izrazite prednosti korištenja povezanih listi umjesto nizova nastaje u primjenama u kojima je potrebno često ubacivati nove elemente *između* do tada ubaćenih elemenata, ili izbacivati elemente koji se nalaze između postojećih elemenata. Poznato je da su ove operacije u slučaju nizova veoma neefikasne. Na primjer, za ubacivanje novog elementa usred niza, prethodno je potrebno sve elemente niza koji slijede iza pozicije na koju želimo ubaciti novi element pomjeriti za jedno mjesto naviše, da bi se stvorilo prazno mjesto za element koji želimo da ubacimo. Ovim se troši mnogo vremena, pogotovo ukoliko je potrebno pomjeriti mnogo elemenata niza. Slično, da bismo uklonili neki element iz niza, potrebno je sve elemente niza koji se nalaze iza elementa koji izbacujemo pomjeriti za jedno mjesto unazad. Međutim, ubacivanje elemenata unutar liste može se izvesti znatno efikasnije, uz mnogo manje trošenja vremena. Naime, dovoljno je kreirati novi čvor koji sadrži element koji umećemo, a zatim izvršiti uvezivanje pokazivača tako da novokreirani čvor logički dođe na svoje mjesto. Sve ovo se može izvesti veoma efikasno (potrebno je promijeniti samo dva pokazivača). Također, brisanje elementa se može izvesti samo promjenom jednog pokazivača (tako da se u lancu povezanih čvorova "zaobiđe" čvor koji sadrži element koji želimo izbrisati), i brisanjem čvora koji sadrži suvišan element primjenom operatora "**delete**". Ovdje su date samo osnovne ideje, a kompletna implementacija izloženih ideja uslijediće u kasnijim poglavljima.

Treba napomenuti da jednostruko povezane liste nisu jedine strukture podataka za čiju se realizaciju koriste čvorovi. Čvorovi su također neizostavan gradivni element drugih složenijih struktura podataka kako što su *višestruko povezane liste, stabla i grafovi*. Zbog velike važnosti ovih struktura podataka u praksi, o njima ćemo detaljnije govoriti u kasnijim poglavljima, kada upoznamo još neke elemente jezika C++ koji olakšavaju njihovu efikasnu i fleksibilnu implementaciju.

30. Klase i objektno orijentirana filozofija

U prethodnom poglavlju smo razmotrili strukture koje predstavljaju složene tipove podataka koje mogu čuvati čitavu skupinu raznorodnih podataka, od kojih neki mogu biti ponovo struktturnog tipa. Strukture praktično omogućavaju jednostavno modeliranje svih tipova podataka koji se susreću u realnom životu. Međutim, strukture definiraju samo *podatke* kojom se opisuje neka jedinka (npr. neki student), ali ne i *postupke* koji se mogu primijeniti nad tom jedinkom. Na taj način, strukture su na izvjestan način *preslobodne*, u smislu da se nad njihovim poljima mogu neovisno izvoditi sve operacije koje su dozvoljene sa tipom podataka koji polje predstavlja, bez obzira toga da li ta operacija ima smisla nad strukturu *kao cjelinom*. Na primjer, neka je definirana struktura “*Datum*”, koja se sastoji od tri polja (atributa) nazvana “*dan*”, “*mjesec*” i “*godina*”:

```
struct Datum {  
    int dan, mjesec, godina;  
};
```

Polja “*dan*”, “*mjesec*” i “*godina*” očigledno su namijenjena da čuvaju dan, mjesec i godinu koji tvore neki stvarni datum. Međutim, kako su ova polja praktično obične cjelobrojne promjenljive, ništa nas ne sprečava da napišemo nešto poput

```
Datum d;  
d.dan = 35;  
d.mjesec = 14;  
d.godina = 2004;
```

bez obzira što je datum 35. 14. 2004. očigledno liшен svakog smisla. Međutim, kako se “*d.dan*”, “*d.mjesec*” i “*d.godina*” ponašaju kao obične cjelobrojne promjenljive, ne postoji nikakav mehanizam koji bi nas sprječio da ovaku dodjelu učinimo. Sa aspekta izvršavanja operacija, ne vidi se da “*dan*”, “*mjesec*” i “*godina*” predstavljaju dio jedne nerazdvojive cjeline, koji se ne mogu postavljati na proizvoljan način, i neovisno jedan od drugog.

Razmotrimo još jedan primjer koji ilustrira “opasnost” rada sa strukturama, zbog preslobodne mogućnosti manipulacije sa njenim poljima. Neka je, na primjer, data sljedeća struktura:

```
struct Student {  
    char ime[30], prezime[30];  
    int indeks;  
    int ocjene[50];  
    double prosjek;  
};
```

Očigledno je da bi polje “prosjek” trebalo da bude “vezano” za polje “ocjene”. Međutim, ništa nas ne sprečava da sve ocjene nekog studenta postavimo npr. na 6, a prosjek na 10. Ova polja se ponašaju kao da su potpuno odvojena jedna od drugog, a ne tjesno povezane komponente jedne cjeline.

Jedna od mogućnosti kojima možemo *djelimično* riješiti ovaj problem je da definiramo izvjesne funkcije koje će pristupati poljima strukture na strogo kontroliran način, a da zatim pri radu sa strukturu koristimo isključivo napisane funkcije za pristup poljima. Na primjer, mogli bismo postaviti funkciju “PostaviDatum“ koja bi postavljala polja “dan”, “mjesec“ i “godina“ unutar strukture koja se prenosi kao prvi parametar u funkciju na vrijednosti zadane drugim, trećim i četvrtim parametrom. Pri tome bi funkcija mogla provjeriti smislenost parametara i preduzeti neku akciju u slučaju da oni nisu odgovarajući (npr. baciti izuzetak). Na primjer, takva funkcija bi mogla izgledati ovako (provjera da li je godina prestupna izvedena je u skladu sa gregorijanskim kalendarom):

```
void PostaviDatum(Datum &d, int dan, int mjesec, int godina) {
    int broj_dana[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if(godina % 4 == 0 && godina % 100 != 0 || godina % 400 == 0)
        broj_dana[1]++;
    if(godina < 1 || mjesec < 1 || mjesec > 12 || dan < 1
       || dan > broj_dana[mjesec - 1])
        throw "Neispravan datum!\n";
    d.dan = dan; d.mjesec = mjesec; d.godina = godina;
}
```

Napisanu funkciju “PostaviDatum“ možemo koristiti za postavku datuma na konzistentan način (dakle, sva polja odjedanput), uz provjeru legalnosti. Tako će, u narednom primjeru, prva postavka proći bez problema, dok će druga baciti izuzetak:

```
Datum d1, d2;
PostaviDatum(d1, 14, 5, 2004);
PostaviDatum(d2, 35, 14, 2004);
```

Međutim, ovim je problem samo djelomično riješen. Nas *ništa ne prisiljava* da moramo koristiti funkciju “PostaviDatum”. Još uvijek je moguće direktno pristupati poljima strukture “Datum“ i tako u nju unijeti nekonzistentan sadržaj. Problem je što je funkcija “PostaviDatum“ potpuno odvojena od strukture “Datum”, ne predstavlja njen dio, i ničim se ne nameće njen korištenje.

Čitatelj odnosno čitateljka se prirodnog mogu zapitati *zašto bi nam uopće moralo biti nametano šta možemo raditi a šta ne*, odnosno zašto se ne bismo jednostavno pridržavali discipline i pristupali elementima strukture na propisani način. Problem je u tome što je lako držati se discipline u manjim programima, koje razvija jedan pojedinac. U složenim programima, koje razvija čitav tim ljudi, i čija dužina može iznositi i više stotina hiljada redova, nemoguće je održavati disciplinu programiranja ukoliko se ne uvede neki mehanizam *koji nas na nju prisiljava*. Stoga je potreba za timskim razvojem velikih programa dovela do razvoja potpuno nove metodologije za razvoj programa, koja je nazvana *objektno orijentirano programiranje (OOP)*, kod koje se mnogo veći značaj daje samim podacima i

tipovima podataka nego što je to do sada bio slučaj u klasičnoj metodologiji programiranja, koja je poznata i pod nazivom *proceduralno programiranje*.

Suštinski nedostatak proceduralnog pristupa sastoji se u tome što se previše pažnje posvećuje *postupcima* odnosno *procedurama* koji se obavljaju nad podacima (obično izvedenim u formi *funkcija*), a pre malo samim podacima. Podaci se posmatraju kao sirova hrpa činjenica, čija je jedina svrha postojanja da budu proslijedeni nekoj funkciji koja ih obrađuje. Dakle, smatra se da je *funkcija* nešto što je važno, a da podaci samo služe kao "hrana" za funkciju. Međutim, da li je to zaista ispravno gledište? Da li podaci služe samo zato da bi funkcije imale šta da obrađuju? Prije bi se moglo reći da je tačno obrnuto gledište: *funkcije služe da bi opsluživale podatke*. Funkcije postoje radi podataka, a ne podaci radi funkcija!

Mada gore navedeno razmatranje djeluje čisto filozofski, ono je dovelo do drastične promjene u načinu razmišljanja kako se programi zapravo trebaju pisati, i do uvođenja pojma *klasa* odnosno *razreda*, koje predstavljaju složene tipove podataka slične strukturama, ali koje ne objedinjuju samo sirove podatke, nego i *kompletan opis akcija koje se mogu primjenjivati nad tim podacima*. Pri tome je nad podacima moguće izvoditi samo akcije koje su navedene u klasi, i nikakve druge. Na taj način se ostvaruje konzistencija odnosno integritet podataka unutar klase. Prije nego što pređemo na sam opis kako se kreiraju klase, razmotrimo još neke primjere filozofske prirode koji ilustriraju razliku između proceduralnog i objektno orijentiranog razmišljanja. Posmatrajmo na primjer izraz " $\log 1000$ ". Da li je ovdje u središtu pažnje funkcija " \log " ili njen argument " 1000 "? Ukoliko razmišljate da se ovdje radi o funkciji " \log " kojoj se šalje argument " 1000 " i koja daje kao rezultat broj " 3 ", razmišljate na proceduralni način. S druge strane, ukoliko smatraste da se u ovom slučaju na broj " 1000 " primjenjuje operacija (funkcija) " \log " koja ga transformira u broj " 3 ", tada razmišljate na objektno orijentirani način. Dakle, proceduralno razmišljanje forsira *funkciju*, kojoj se šalje podatak koji se obrađuje, dok objektno orijentirano razmišljanje forsira *podatak* nad kojim se funkcija *primjenjuje*. Navedimo još jedan primjer. Posmatrajmo izraz " $5 + 3$ ". Proceduralna filozofija ovdje stavlja središte pažnje na *operaciju* sabiranja (" $+$ ") kojoj se kao argumenti šalju podaci " 5 " i " 3 ", i koja daje kao rezultat " 8 ". Međutim, sa aspekta objektno orijentirane filozofije, u ovom slučaju se nad podatkom " 5 " primjenjuje akcija " $+ 3$ " (koju možemo tumačiti kao "povećaj se za 3 ") koja ga transformira u novi podatak " 8 ". Ne može se reći da je bilo koja od ove dvije filozofije (proceduralna odnosno objektno orijentirana) ispravna a da je druga neispravna. Obje su ispravne, samo imaju različita gledišta. Međutim, praksa je pokazala da se za potrebe razvoja složenijih programa objektno orijentirana filozofija pokazala znatno efikasnijom, produktivnijom, i otpornijom na greške u razvoju programa.

Osnovu objektno orijentiranog pristupa programiranju čine *klase* odnosno *razredi* (engl. *class*), za koje smo već rekli da predstavljaju objedinjenu cjelinu koja objedinjuje skupinu podataka, kao i postupke koji se mogu primjenjivati nad tim podacima. Klasu dobijamo tako što unutar strukture dodamo i prototipove funkcije koje se mogu *primjenjivati* nad tim podacima. Na primjer, sljedeća deklaracija definira *klasu* "Datum", koja pored podataka "dan", "mjesec" i "godina" sadrži i prototip funkcije "Postavi", koja će se primjenjivati nad promjenljivim tipa "Datum":

```
struct Datum {  
    int dan, mjesec, godina;  
    void Postavi(int d, int m, int g);  
};
```

Treba naglasiti da se promjenljive čiji je tip klasa obično nazivaju *primjercima klase*, *instancama klase*, ili prosto *objektima* (mada smo mi do sada objektima zvali i konkretne primjerke promjenljivih bilo kojeg tipa). U tom smislu, klasa predstavlja apstrakciju koja definira zajednička svojstva i zajedničke postupke koji su svojstveni svim objektima nekog tipa.

Funkcije čiji su prototipovi navedeni unutar same klase, nazivaju se *funkcije članice klase* ili *metode*, i one su namijenjene da se *primjenjuju* nad konkretnim objektima te klase, a ne da im se kao parametri *šalju* objekti te klase. Zbog toga je sintaksa pozivanja funkcija članica *drugačija* nego sintaksa pozivanja običnih funkcija. One se uvjek primjenjuju *nad nekim konkretnim objektom* korištenjem operatora “.” (tačka), dakle istim operatorom koji se koristi i za pristup atributima unutar klase. Naravno, funkcija članica “Postavi” mora negdje biti i *implementirana*, kao i sve ostale funkcije. Ostavimo za trenutak njenu implementaciju, i pogledajmo na koji bi se način ova funkcija pozvala nad dva konkretna objekta tipa “Datum” (drugi poziv bi trebao da baci izuzetak, s obzirom na besmislen datum):

```
Datum d1, d2;
d1.Postavi(14, 5, 2004);
d2.Postavi(35, 14, 2004);
```

Nemojte misliti da je ovim izvršena samo promjena sintakse u odnosu na raniji primjer u kojem smo koristili (običnu) funkciju “PostaviDatum”. Do promjene u sintaksi je zaista došlo, međutim ta promjena treba da nas uputi na promjenu načina razmišljanja. Funkciji “PostaviDatum” se kao argument *šalju* objekti “d1” i “d2”, dok se funkcija “Postavi” *primjenjuje* nad objektima “d1” i “d2”. Ubrzo ćemo shvatiti kakvu nam korist donosi ta promjena načina razmišljanja.

Razmotrimo sada kako bi se trebala implementirati funkcija članica “Postavi”. Njena implementacija mogla bi, na primjer, izgledati ovako:

```
void Datum::Postavi(int d, int m, int g) {
    int broj_dana[] = {31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31};
    if(g % 4 == 0 && g % 100 != 0 || g % 400 == 0) broj_dana[1]++;
    if(g < 1 || d < 1 || m < 1 || m > 12 || d > broj_dana[m - 1])
        throw "Neispravan datum!\n";
    dan = d; mjesec = m; godina = g;
}
```

Ovdje možemo uočiti nekoliko detalja koji odudaraju od implementacije klasičnih funkcija. Prvo, ispred imena funkcije nalazi se dodatak “Datum::”. Ova konstrukcija ukazuje da se ne radi o običnoj funkciji, nego o funkciji članici klase “Datum”. Na taj način je omogućeno da mogu postojati obična funkcija i funkcija članica istog imena, kao i da više različitih klasa imaju funkcije članice istih imena. Znak “::” (dupla dvotačka) također spada u operatore, i obično se naziva *operator razlučivosti, vidokruga* ili *dosega* (engl. *scope operator*). Preciznije, sa ovim operatorma smo se već susreli ranije, ali u *unarnoj formi* (npr. u izrazu poput “::a”) dok se ovdje susrećemo sa njegovom *binarnom formom* (zapravo, i sa binarnom formom operatora “::” smo se već susreli na samom početku u konstrukcijama poput “std::cout”). Upotrijebljen kao binarni operator, operator “::” se najčešće koristi u obliku “klasa::identifikator”, i označava da se identifikator “identifikator” odnosi na atribut odnosno metodu koja pripada klasi “klasa”. Drugo, primijetimo da se unutar funkcije članice direktno pristupa atributima “dan”, “mjesec” i “godina” *bez navođenja na koji se konkretni objekat ovi atributi odnose*, što do sada nije bilo moguće. Ovakvu privilegiju direktnom pristupanju atributima klase imaju isključivo funkcije članice te klase. Ovo je moguće zbog toga što se funkcije članice klase nikad ne pozivaju samostalno, nego se uvjek primjenjuju nad nekim konkretnim objektom (uz izuzetak kada se neka funkcija članica poziva iz druge funkcije članice iste klase, a koji ćemo uskoro objasniti), tako da se samostalno upotrijebljeni atributi unutar funkcije članice zapravo odnose na atribute *onog objekta nad kojim je funkcija članica pozvana*. Tako, ukoliko npr. izvršimo poziv

```
d1.Postavi(14, 5, 2004);
```

atributi “dan”, “mjesec” i “godina” unutar funkcije članice “Postavi” odnosiće se na odgovarajuće attribute objekta “d1”, tj. interpretiraće se kao “d1.dan”, “d1.mjesec” i “d1.godina”.

Prirodno je postaviti pitanje *kako funkcija članica može znati nad kojim je objektom primijenjena*, odnosno kako može znati na koji se objekat samostalno upotrijebljeni atributi trebaju odnositi. Kao i kod svake druge indirekcije, i ovdje se u pozadini igre koriste *pokazivači*. Tajna je u tome što se prilikom poziva ma koje metode (funkcije članice) nad nekim objektom, adresa samog objekta nad kojim se metoda primjenjuje smješta u jedan pokazivač koji se naziva “**this**” (ovo “**this**” je zapravo ključna riječ, ali se u svemu ponaša kao klasična konstantna pokazivačka promjenljiva), dok se svim atributima unutar metode za koje nije specificirano na koji se objekat odnose pristupa indirektno preko pokazivača “**this**”. Tako se, na primjer, samostalna upotreba atributa “*dan*” interpretira kao “*(*this).dan*”, odnosno “**this->dan**”. Stoga smo funkciju članicu “Postavi” sasvim legalno mogli napisati i ovako:

```
void Datum::Postavi(int d, int m, int g) {
    int broj_dana[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31};
    if(g % 4 == 0 && g % 100 != 0 || g % 400 == 0) broj_dana[1]++;
    if(g < 1 || d < 1 || m < 1 || m > 12 || d > broj_dana[m - 1])
        throw "Neispravan datum!\n";
    this->dan = d; this->mjesec = m; this->godina = g;
}
```

Možemo reći da se u jeziku C++ kada god unutar funkcije članice neke klase upotrijebimo samostalno ime nekog atributa te klase (vidjećemo da to isto vrijedi i za metode) podrazumijeva da ispred navedenog atributa (metode) piše prefiks “**this->**”.

Postojanje pokazivača “**this**” zapravo znači da se svaka funkcija članica interno interpretira kao obična funkcija koja ima pokazivač “**this**” kao skriveni parametar. Stoga se metoda “Postavi” interno ponaša kao obična funkcija (nazovimo je “Postavi_obicna”) sa sljedećim protoipom:

```
void Postavi_obicna(Datum *this, int d, int m, int g);
```

dok se njen poziv nad objektom “*d1*” interpretira kao

```
Postavi_obicna(&d1, 14, 5, 2004);
```

Mada ovakva interpretacija može pomoći onome ko se prvi put susreće sa objektno orijentiranim pristupom da shvati šta se zaista dešava, o njoj ne treba previše misliti, s obzirom da odvraća programera od objektno orijentiranog razmišljanja i okreće ga nazad ka proceduralnom razmišljanju, tj. posmatranju funkcija kao cjeline posve odvojenih od objekata na koje se one primjenjuju.

Primijetimo da smo u funkciji članici “Postavi” parametre nazvali prosto “*d*”, “*m*” i “*g*”, sa ciljem da izbjegnemo konflikt sa imenima atributa klase “*Datum*”, koji se u funkciji članici mogu koristiti samostalno, bez vezanja na konkretan objekat. Da smo parametre funkcije također nazvali “*dan*”, “*mjesec*” i “*godina*”, bilo bi nejasno na šta se npr. identifikator “*dan*” odnosi, da li na *formalni parametar* “*dan*” ili na *atribut* “*dan*”. Za razrješavanje ovog konflikta, u jeziku C++ je usvojena konvencija da u slučaju pojave istih imena formalni parametri ili eventualne lokalne promjenljive definirane unutar funkcije članice *imaju prioritet* u odnosu na istoimene atribute ili metode, dok s druge strane atributi i metode imaju prioritet u odnosu na eventualne istoimene globalne promjenljive ili klasične funkcije. Stoga bi se samostalno upotrijebljen identifikator “*dan*” odnosio na formalni parametar “*dan*”, a ne na atribut “*dan*”. Ukoliko ipak želimo da pristupimo atributu “*dan*”, moguća su dva načina. Jedan je da eksplicitno koristimo pokazivač “**this**”. Na primjer, funkciju članicu “Postavi” mogli smo napisati i ovako:

```
void Datum::Postavi(int dan, int mjesec, int godina) {
```

```

int broj_dana[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
if(godina % 4 == 0 && godina % 100 != 0 || godina % 400 == 0)
    broj_dana[1]++;
if(godina < 1 || dan < 1 || mjesec < 1 || mjesec > 12
    || dan > broj_dana[mjesec - 1])
    throw "Neispravan datum!\n";
this->dan = dan; this->mjesec = mjesec; this->godina = godina;
}

```

Drugi način, koji se češće koristi, je upotreba operatora razlučivosti “::”. Tako, ukoliko napišemo “Datum::dan”, eksplisitno naglašavamo da mislimo na atribut “dan” koji pripada klasi “Datum”. Stoga smo metodu “Postavi“ mogli napisati i na sljedeći način, u kojem izbjegavamo upotrebu pokazivača “this”:

```

void Datum::Postavi(int dan, int mjesec, int godina) {
    int broj_dana[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31};
    if(godina % 4 == 0 && godina % 100 != 0 || godina % 400 == 0)
        broj_dana[1]++;
    if(godina < 1 || dan < 1 || mjesec < 1 || mjesec > 12
        || dan > broj_dana[mjesec - 1])
        throw "Neispravan datum!\n";
    Datum::dan = dan; Datum::mjesec = mjesec; Datum::godina = godina;
}

```

Bitno je istaći veliku razliku između operatora “.” i “::”. Operator “.” se uvijek primjenjuje na neki *konkretan objekat*. Tako, izraz “d1.dan” označava atribut “dan” konkretnog objekta “d1”. S druge strane, operator “::” se primjenjuje nad samom *klasom*, kao apstraktnim tipom podataka. Tako izraz “Datum::dan” označava atribut “dan” koji pripada klasi “Datum”, ali bez specifikacije na koji se konkretan objekat klase “Datum” taj atribut odnosi. Stoga, ovakva upotreba atributa “dan” može imati smisla jedino unutar definicije neke metode, jer će prilikom poziva te metode naknadno postati jasno na koji se konkretan objekat atribut odnosi.

S obzirom da se svim atributima neke klase unutar funkcija članica može pristupati i neposredno, bez eksplisitne upotrebe pokazivača “this”, ovaj pokazivač se dosta rijetko koristi eksplisitno. Međutim, postoje situacije kada je eksplisitna upotreba ovog pokazivača veoma korisna, pa čak i nužna. To se javlja u situacijama kada zbog nekog razloga iz funkcije članice želimo pristupiti objektu nad kojim je funkcija članica pozvana, ali *kao cjelini* a ne njenim pojedinačnim atributima. Zamislimo, na primjer, da klasa “Datum” ima neku metodu koja se zove “NekaMetoda”, i koja treba da pozove neku drugu funkciju ili metodu nazvanu “NekaFunkcijaKojaPrimaDatumKaoParametar” koja kao parametar prima objekat tipa “Datum”, i kojoj je potrebno kao stvarni parametar prenijeti objekat nad kojim je metoda “NekaMetoda” pozvana. Tada bi implementacija metode “NekaMetoda” mogla izgledati ovako:

```

void Datum::NekaMetoda() {
    ...
    NekaFunkcijaKojaPrimaDatumKaoParametar(*this);
    ...
}

```

Potreba za eksplisitnom upotrebotom pokazivača “this” može se još javiti i u slučaju kada metoda treba da vrati kao rezultat objekat nad kojim je pozvana (nakon što je eventualno izvršila neke izmjene nad njim), ili ukoliko je potrebno da unutar metode imamo lokalnu promjenljivu tipa klase u koju želimo da iskopiramo objekat nad kojim je metoda pozvana. Tada bismo mogli pisati nešto poput:

```
Datum pomocna = *this;
```

Pažljiviji čitatelj ili čitateljka će primijetiti da i nakon mnogo priče koja je ispisana na prethodnim stranicama, još uvijek nismo riješili osnovni problem koji nas je motivirao da uopće uvedemo pojam klase! Još nas niko ne sprečava da prosto zaobiđemo metodu "Postavi" i direktnom postavkom atributa "dan", "mjesec" i "godina" upišemo besmislen datum. Potreban nam je mehanizam pomoću kojeg ćemo prisiliti korisnika klase da koristi klasu isključivo na način kako je to propisao projektant klase (što je naročito važno u slučaju kada projektant i korisnik klase nisu ista osoba, što je tipičan slučaj u većim projektima). Jezik C++ omogućava elegantno rješavanje navedenog problema uvođenjem ključnih riječi "**private**" i "**public**". Ove ključne riječi koriste se *isključivo unutar deklaracija klase*, a iza njih uvijek slijedi znak ":" (dvotačka). One specificiraju koji dio klase je *privatan*, a koji *javan*. Na primjer, prepostavimo da smo izmijenili deklaraciju klase "Datum" tako da izgleda ovako:

```
struct Datum {  
    private:  
        int dan, mjesec, godina;  
    public:  
        void Postavi(int d, int m, int g);  
};
```

Na taj način smo atribute "dan", "mjesec" i "godina" proglašili *privatnim*, dok smo metodu "Postavi" proglašili *javnom*. Privatnim atributima i metodama može se pristupiti *samo iz tijela funkcija članica (metoda) te iste klase* i iz tzv. *prijateljskih funkcija klase* koje ćemo razmotriti malo kasnije. Bilo kakav pristup privatnim metodama i atributima iz ostalih dijelova programa je *zabranjen* i dovodi do prijave greške od strane kompjajlera. Zbog toga naredbe poput sljedećih *postaju zabranjene*:

```
d1.dan = 20;  
cout << d1.dan;
```

Primijetimo da je ovim postalo zabranjeno ne samo direktno *mijenjati* vrijednost atributa "dan", nego čak i *čitati* njegovu vrijednost. Privatni atributi su jednostavno *nedostupni* za ostatak programa, osim za funkcije članice klase i prijateljske funkcije klase (preciznije, oni su i dalje *vidljivi* u smislu da ostatak programa zna za njihovo postojanje, ali im se ne može *pristupiti*)! Ipak, na ovaj način smo klasu učinili *potpuno neupotrebljivom*. Naime, funkcija "Postavi" nam omogućava da postavimo sadržaj nekog objekta (i to samo na legalan datum), ali nakon toga ti podaci ostaju *zarobljeni u objektu*, i ne možemo ništa uraditi sa njima! Izlaz iz ovog problema je posve jednostavan: *treba dodati još metoda u objekat, koji će raditi nešto korisno sa objektom*. Na primjer, moguće je dodati metodu "Ispisi" koja ispisuje datum na ekran. Stoga ćemo promijeniti deklaraciju klase "Datum" da izgleda ovako:

```
struct Datum {  
    private:  
        int dan, mjesec, godina;  
    public:  
        void Postavi(int d, int m, int g);  
        void Ispisi();  
};
```

Naravno, metodu "Ispisi" je potrebno i implementirati, što je posve jednostavno:

```
void Datum::Ispisi() {  
    cout << dan << ". " << mjesec << ". " << godina << ". ";
```

Sada već sa objektima tipa "Datum" možemo raditi dvije stvari: možemo im *postavljati sadržaj* i *ispisivati ih*. Stoga je legalno napisati sljedeću sekvencu naredbi:

```
Datum d;  
d.Postavi(14, 5, 2004);  
cout << "Unesen je datum ";  
d.Ispisi();
```

Primijetimo da smo na ovaj način oštro ograničili šta se može raditi sa promjenljivim tipa "Datum": *postavka*, *ispis* i *ništa drugo*. Tako mi kao projektanti klase jasno specificiramo šta je moguće raditi sa instancama te klase. Princip dobrog objektno orijentiranog programiranja obično predviđa da se svi atributi klase definiraju isključivo kao *privatni*, dok bi sav pristup atributima trebao da se vrši preko funkcija članica. Ovaj princip naziva se *sakrivanje informacija* (engl. *information hiding*). Na taj način ostvarena je stroga kontrola nad načinom korištenja klase. Sakrivanje informacija čini *prvi postulat objektno orijentiranog programiranja* (od ukupno četiri). Ostala tri postulata čine *enkapsulacija*, *nasljeđivanje* i *polimorfizam*. Pri tome, pod enkapsulacijom ili učahurivanjem (engl. *encapsulation*) podrazumijevamo već rečeno objedinjavanje podataka i postupaka koji se mogu primjenjivati nad tim podacima u jednu cjelinu, dok ćemo o nasljeđivanju i polimorfizmu govoriti kasnije. Trebalo bi naglasiti da o objektno orijentiranom programiranju možemo govoriti tek u slučajevima u kojima koristimo *sva četiri postulata* objektno orijentiranog programiranja. Nešto manje radikalna metodologija programiranja koja se oslanja na sakrivanje informacija i enkapsulaciju, ali ne i na nasljeđivanje i polimorfizam, naziva se *objektno zasnovano* odnosno *objektno bazirano programiranje (OBP)*, koju mnogi brkaju sa pravim objektno orijentiranim programiranjem.

Mada se klase mogu definirati pomoću ključne riječi "**struct**", danas je dio programerskog bontona da se klase definiraju isključivo pomoću ključne riječi "**class**", dok se riječ "**struct**" ostavlja isključivo za strukture (tj. složene tipove podataka koji sadrže isključivo atribute, a ne i metode). Inače, ključna riječ "**class**" djeluje veoma slično kao ključna riječ "**struct**", ali uz jednu malu razliku. Naime, kod upotrebe ključne riječi "**struct**", svi atributi i metode za koje se eksplicitno ne kaže da su privatni smatraju se javnim. S druge strane, ključna riječ "**class**" sve atribute i metode za koje se eksplicitno ne kaže da su javni smatra privatnim, i na taj način podstiče sakrivanje informacija (sve je privatno dok se eksplicitno ne kaže drugačije). Stoga bismo klasu "Datum" u skladu sa programerskim bontonom trebali deklarirati ovako:

```
class Datum {  
    int dan, mjesec, godina;  
public:  
    void Postavi(int d, int m, int g);  
    void Ispisi();  
};
```

Veoma često je potrebno omogućiti *čitanje* ali ne i *izmjenu* izvjesnih atributa klase. Na primjer, prethodna klasa "Datum" je prilično ograničena, s obzirom da postavljeni datum možemo samo *ispisati*. Ne možemo čak ni saznati koji je datum upisan (bez da vršimo njegov ispis na ekran). Nije nikakav problem dodati u našu klasu novu metodu "Ocitaj" koja bi u tri parametra prenesena po referenci smjestila vrijednosti atributa "dan", "mjesec" i "godina", i koja bi se mogla koristiti za očitavanje atributa klase. U skladu sa tim, nova definicija klase "Datum" mogla bi izgledati ovako:

```
class Datum {  
    int dan, mjesec, godina;  
public:  
    void Postavi(int d, int m, int g);
```

```

void Ocitaj(int &d, int &m, int &g);
void Ispisi();
};

```

Implementacija metode “Ocitaj“ je veoma jednostavna. S obzirom na kratkoću njene implementacije, definiraćemo je kao umetnutu funkciju pomoću kvalifikatora “**inline**”, čime dobijamo na efikasnosti:

```

inline void Datum::Ocitaj(int &d, int &m, int &g) {
    d = dan; m = mjesec; g = godina;
}

```

Sada bismo mogli napisati programski isječak poput sljedećeg:

```

Datum dat;
dat.Postavi(30, 12, 2002);
int dan, mj, god;
dat.Ocitaj(dan, mj, god);
cout << "Dan = " << dan << " Mjesec = " << mj << "Godina = " << god;

```

Moguće je napisati još fleksibilnije rješenje. Naime, ako malo razmislimo, vidjećemo da nema nikakve štete od mogućnosti zasebnog čitanja atributa “dan”, “mjesec“ i “godina”, dok mogućnost nekontroliranog mijenjanja ovih atributa nije poželjna (napomenimo da je sasvim moguće zamisliti situacije u kojima nije poželjno ni čitati izvjesne attribute klase). Čitanje ovih atributa možemo omogućiti tako što ćemo dodati tri trivijalne funkcije članice “VratiDan”, “VratiMjesec“ i “VratiGodinu“ bez parametara, koje će prosto vraćati kao rezultat vrijednosti atributa “dan”, “mjesec“ i “godina“:

```

class Datum {
    int dan, mjesec, godina;
public:
    void Postavi(int d, int m, int g);
    void Ocitaj(int &d, int &m, int &g);
    int VratiDan();
    int VratiMjesec();
    int VratiGodinu();
    void Ispisi();
};

```

Implementacija ove tri metode je trivijalna:

```

inline int Datum::VratiDan() {
    return dan;
}

inline int Datum::VratiMjesec() {
    return mjesec;
}

inline int Datum::VratiGodinu() {
    return godina;
}

```

Sada možemo pisati i ovakve programske isječke:

```

Datum d;
d.Postavi(30, 12, 2002);
cout << "Dan = " << d.VratiDan() << " Mjesec = " << d.VratiMjesec()
    << "Godina = " << d.VratiGodinu();

```

Na ovaj način omogućili smo čitanje atributa “dan”, “mjesec” i “godina”, doduše ne posve direktno, već pomoću pristupnih funkcija članica “VratiDan”, “VratiMjesec” i “VratiGodinu”. Onima koji se prvi put susreću sa objektno orijentiranim pristupom, ova indirekcija djeluje kao nepotrebna komplikacija. Međutim, dugoročno gledano, ovaj pristup donosi mnoge koristi, jer je ponašanje klase pod punom kontrolom projektanta klase. Na primjer, funkcije za pristup pojedinim atributima mogu se napisati tako da pod izvjesnim okolnostima vrate *lažnu vrijednost atributa* ili čak da *odbiju da vrate vrijednost* (npr. bacanjem izuzetka). U praksi se često javljaju situacije gdje ovakvo ponašanje može da ima smisla.

Mnogi programeri imaju odbojnost ka definiranju trivijalnih metoda poput “VratiMjesec” itd. jer smatraju da je na taj način potrebno mnogo pisana. Naime, svaku takvu metodu potrebno je kako *deklarirati*, tako i *implementirati*. Da bi se smanjila potreba za količinom pisana, C++ omogućava da se implementacija metoda obavi na istom mjestu kao i deklaracija (slično kao kod običnih funkcija). Na primjer, sljedeća deklaracija klase “Datum” implementira sve metode osim metode “Postavi” odmah na mjestu njihove deklaracije (u tom slučaju, metoda “Postavi” se mora implementirati negdje drugdje da bi klasa bila kompletna):

```
class Datum {
    int dan, mjesec, godina;
public:
    void Postavi(int d, int m, int g);
    void Ocitaj(int &d, int &m, int &g) {
        d = dan; m = mjesec; g = godina;
    }
    int VratiDan() { return dan; }
    int VratiMjesec() { return mjesec; }
    int VratiGodinu() { return godina; }
    void Ispisi() {
        cout << dan << ". " << mjesec << ". " << godina << ".";
    }
};
```

Ipak, važno je napomenuti da nije svejedno da li se metoda implementira *unutar deklaracije klase* (tj. odmah pri deklaraciji) ili *izvan deklaracije klase*. U slučaju kada se implementacija metode izvodi unutar deklaracije klase, automatski se podrazumijeva da se metoda izvodi kao *umetnuta funkcija*, odnosno da je implementirana sa kvalifikatorom “**inline**”. Stoga vrijedi praktično pravilo: metode koje se sastoje od svega jedne ili dvije naredbe (eventualno, do tri naredbe) najbolje je implementirati *odmah unutar deklaracije klase* (može se tolerirati i više naredbi, ukoliko se radi samo o naredbama dodjeljivanja), čime dobijamo na brzini (eventualno, ukoliko ih ipak implementiramo izvan deklaracije klase, poželjno ih je označiti kvalifikatorom “**inline**”). Sve ostale metode treba implementirati *izvan deklaracije klase* (ovo pogotovo vrijedi za metode koje sadrže petlje i druge složenije programske konstrukcije).

Iz navedenih primjera može se primijetiti da se sve metode mogu podijeliti u dvije kategorije. Metode iz prve kategorije samo *čitaju* attribute klase, ali ih *ne mijenjaju*. Takve su, u prethodnom primjeru, metode “Ocitaj”, “VratiDan”, “VratiMjesec”, “VratiGodinu” i “Ispisi”. Takve metode nazivaju se *inspektori*. Metode iz druge kategorije, u koje spada recimo metoda “Postavi”, *mijenjaju* attribute klase nad kojom su primjenjene. Takve funkcije nazivaju se *mutatori*.

U jeziku C++ se smatra dobrom praksom posebno označiti funkcije inspektore ključnom riječi “**const**”, koja se navodi nakon zatvorene zagrade koja definira popis parametara metode. Tako označene funkcije nazivaju se *konstantne funkcije članice*. Od ovakvog označavanja imamo dvije koristi. Prvo, na taj način se omogućava kompjuleru prijava greške u slučaju da unutar metode za koju konceptualno

znamo da treba da bude inspektor pokušamo promijeniti vrijednost nekog od atributa. Drugo, pomoću ključne riječi “**const**“ moguće je definirati konstantne objekte, na isti način kao i npr. cjelobrojne konstante. Prirodno je da je nad takvim objektima moguće pozivati samo metode inspektore, ali ne i mutatore. Zbog toga je usvojena konvencija da se nad konstantnim objektima smiju pozivati samo metode koje su ekplizitno deklarirane kao konstantne funkcije članice. Stoga bismo deklaraciju klase “Datum“ mogli još bolje izvesti ovako:

```
class Datum {
    int dan, mjesec, godina;
public:
    void Postavi(int d, int m, int g);
    void Sljedeci();
    void Ocitaj(int &d, int &m, int &g) const {
        d = dan; m = mjesec; g = godina;
    }
    int VratiDan() const { return dan; }
    int VratiMjesec() const { return mjesec; }
    int VratiGodinu() const { return godina; }
    void Ispisi() const {
        cout << dan << ". " << mjesec << ". " << godina << ".";
    }
};
```

Ovom prilikom smo u klasu “Datum“ dodali još jednu zanimljivu metodu mutator nazvanu “**Sljedeci**“ koja uvećava objekat na koji je primjenjena tako da sadrži sljedeći dan po kalendaru (vodeći računa o prebacivanju na naredni mjesec i godinu u slučaju potrebe). Njena implementacija bi mogla izgledati na primjer ovako:

```
void Datum::Sljedeci() {
    int broj_dana[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if(godina % 4 == 0 && godina % 100 != 0 || godina % 400 == 0)
        broj_dana[1]++;
    dan++;
    if(dan > broj_dana[mjesec - 1]) {
        dan = 1; mjesec++;
    }
    if(mjesec > 12) {
        mjesec = 1; godina++;
    }
}
```

Način njene upotrebe ilustrira sljedeći primjer:

```
Datum d;
d.Postavi(31, 12, 2004);
d.Sljedeci();
d.Ispisi();
```

Formalno gledano, razlika između običnih i konstantnih funkcija članica je u tome što se u konstantnim funkcijama članicama skriveni pokazivač “**this**” ponaša kao pokazivač na konstantan objekat.

Naglasimo da je ključna riječ “**const**” kojom funkciju članicu proglašavamo konstantnom smatra *dijelom zaglavlja funkcije*, tako da se u slučaju da konstantnu funkciju članicu implementiramo izvan deklaracije klase, ključna riječ “**const**” mora ponoviti, inače će se smatrati da se radi o implementaciji

druge funkcije članice koja *nije konstantna*, sa istim imenom i istim parametrima. **Odavde očigledno slijedi da klasa može imati dvije različite funkcije članice istog imena i sa istim parametrima, od kojih je jedna konstantna a druga nije.** Pri tome se koristi konvencija da se u takvima slučajevima konstantna funkcija članica poziva samo u slučaju da je primijenjena nad konstantnim objektom, dok se u slučaju nekonstantnih objekata poziva istoimena nekonstantna funkcija članica. Potreba za ovakvim razdvajanjem može se javiti kod funkcija članica koje kao rezultat vraćaju reference ili pokazivače, o čemu ćemo govoriti kasnije.

Nekome obaveza deklariranja inspektora kao konstantnih funkcija članica može djelovati kao suvišna komplikacija, s obzirom da na prvi pogled izgleda da se deklaracija konstantnih objekata ne susreće tako često. Međutim, nije baš tako. Pretpostavimo, na primjer, da imamo neku funkciju nazvanu "NekaFunkcija", sa sljedećim prototipom (pri tome ovdje nije bitno šta funkcija tačno radi):

```
void NekaFunkcija(const Datum &d);
```

Ova funkcija prima parametar tipa "Datum" koji se prenosi po referenci na konstantu, s obzirom da je njen formalni parametar "d" deklariran kao referenca na konstantni objekat tipa "Datum" (a već smo vidjeli da se parametri strukturalnih tipova najčešće upravo tako prenose). Međutim, kako se referenca u potpunosti identificira sa objektom koji predstavlja, formalni parametar "d" se ponaša kao *konstantni objekat*, i nad njim se mogu pozivati *isključivo konstantne funkcije članice!*

Napomenimo da je moguće definirati i takve specijalne atribute koji će se moći mijenjati čak i iz metoda deklariranih kao konstantne funkcije članice. Oni se kreiraju tako da se ispred deklaracije samog atributa doda ključna riječ "**mutable**" (npr. "**mutable int** izmjenjivi;"). Pošto ovakvi atributi na neki način kvare sam koncept metoda inspektora, ne treba ih koristiti u slučaju da zaista ne postoji dobar razlog za to (takvi razlozi se mogu pojaviti, inače ključna riječ "**mutable**" ne bi ni bila uvedena).

Klasu "Datum" možemo proširiti sa još dvije korisne metode. Dodaćemo metodu "VratiImeMjeseca" koja vraća puni naziv mjeseca pohranjenog u datumu (preciznije, pokazivač na prvi znak naziva) kao i metodu "IspisiLijepo" koja ispisuje datum sa tekstualno ispisanim nazivom mjeseca umjesto brojčanog ispisa. Obe metode su naravno inspektori, a implementiraćemo ih izvan deklaracije klase:

```
class Datum {
    int dan, mjesec, godina;
public:
    void Postavi(int d, int m, int g);
    void Sljedeci();
    void Ocitaj(int &d, int &m, int &g) const {
        d = dan; m = mjesec; g = godina;
    }
    int VratiDan() const { return dan; }
    int VratiMjesec() const { return mjesec; }
    char *VratiImeMjeseca() const;
    int VratiGodinu() const { return godina; }
    void Ispisi() const {
        cout << dan << ". " << mjesec << ". " << godina << ".";
    }
    void IspisiLijepo() const;
};
```

Implementacija metoda "VratiImeMjeseca" i "IspisiLijepo" mogla bi izgledati ovako:

```

char *Datum::VratiImeMjeseca() const {
    char *imena_mjeseci[] = {"Januar", "Februar", "Mart", "April",
        "Maj", "Juni", "Juli", "August", "Septembar", "Oktobar",
        "Novembar", "Decembar"};
    return imena_mjeseci[mjesec - 1];
}

void Datum::IspisiLijepo() const {
    cout << dan << ". " << VratiImeMjeseca() << " " << godina << ".";
}

```

Interesantno je razmotriti metodu “`IspisiLijepo()`”. Ona unutar sebe poziva metodu “`VratiImeMjeseca()`”. Međutim, interesantno je da se ova metoda ne poziva niti nad jednim objektom. Ovo je dozvoljeno, s obzirom da se unutar ma koje metode ime atributa ili metode upotrijebljeno samo za sebe tretira preko pokazivača “`this`”. Stoga se poziv “`VratiImeMjeseca()`” interpretira kao “`this->VratiImeMjeseca()`” odnosno “`(*this).VratiImeMjeseca()`”. Drugim riječima, ukoliko iz neke metode pozovemo neku drugu metodu iste klase bez eksplicitnog navođenja nad kojim se objektom ta metoda poziva, podrazumijeva se pozivanje nad istim objektom nad kojim je pozvana i prva metoda.

Sada smo već dobili pristojno razvijenu klasu “`Datum`”, sa kojom se može raditi dosta stvari. Očigledno, skup akcija koje klasa podržava definirane su u javnom dijelu klase, odnosno dijelu obilježenom ključnom riječju “`public`”. Stoga se opis javnog dijela klase naziva *sučelje* ili *interfejs klase* (engl. *class interface*).

Kao što smo već nagovijestili ranije, u objektno orijentiranom programiranju moguće je potpuno razdvojiti *projektanta klase* od *korisnika klase*, koji pri timskom razvoju složenijih programa gotovo nikada nisu ista osoba. **Da bi korisnik klase koristio neku klasu, sve što o njoj treba znati je njen interfejs (tj. šta sa njom može raditi), dok je za njega implementacija potpuno nebitna.** Dobro napisana klasa mogla bi se koristiti u mnoštvu različitih programa, bez ikakve potrebe za njenim izmjenama. Tako, razvijenu klasu “`Datum`” bismo mogli koristiti u ma kojem programu u kojem postoji potreba da radimo sa datumima, bez ikakve potrebe da uopće znamo kako su njene metode implementirane. Ovaj princip nazivamo *princip ponovne iskoristivosti* (engl. *reusability*), koji je postao jedan od najvećih aduta objektno orijentiranog programiranja.

Interesantno je napomenuti da smo mi i do sada mnogo puta koristili ovaj princip, mada posve nesvjesno. Recimo, objekti ulaznog i izlaznog toka “`cin`” i “`cout`” koje smo koristili od samog početka nisu nikakve magične naredbe, već obične promjenljive! Preciznije, “`cin`” je u zagлавju “`iostream`” deklariran kao instanca klase “`istream`”, a “`cout`” kao instanca klase “`ostream`”. Klasa “`istream`” definira skup atributa i metoda za pristupanje ulaznom toku (tipično tastaturi), dok klasa “`ostream`” definira skup atributa i metoda za pristupanje izlaznom toku (tipično ekranu). Na primjer, jedna od metoda klase “`istream`” je metoda “`getline`”, dok je jedna od metoda klase “`ostream`” metoda “`width`”. Ovo objašnjava upotrebu sintaksnih konstrukcija poput “`cin.getline(recenica, 50)`” ili “`cout.width(10)`”. Vidimo da se ovdje zapravo radi o pozivanju metoda “`getline`” odnosno “`width`” nad objektima “`cin`” odnosno “`cout`” respektivno. Naravno, ove metode negdje moraju biti implementirane (i jesu implementirane negdje unutar biblioteke “`iostream`”). Međutim, činjenica je da smo mi ove objekte od samog početka koristili ne znajući kako su uopće ove metode implementirane (niti je potrebno da znamo). Ovo i jeste osnovni cilj: razdvojiti interfejs klase od njene implementacije. Također, tip podataka “`string`” sa kojim smo se ranije upoznali nije ništa drugo nego posebno definirana klasa (čije su metode recimo “`length`” i “`c_str`”), a isto vrijedi i za tipove podataka “`complex`” i “`vector`”, koji su takođe definirani kao klase (preciznije, kao generičke klase koje ćemo uskoro upoznati). Na primjer, “`push_back`” nije ništa drugo nego jedna od metoda klase “`vector`”.

Izloženi primjeri ilustriraju u kojoj mjeri korisnik klase ne mora da brine o njenoj implementaciji. Mi smo, zapravo, od samog početka koristili izvjesne klase, koje doduše nismo mi napisali, nego koje su definirane unutar odgovarajućih standardnih biblioteka. Međutim, upravo u tome i jeste poenta. Da bismo koristili klase, mi ne moramo znati njihovu implementaciju. Doduše, ostaju još neke tajne vezane za ove "gotove" klase koje smo do sada koristili. Na primjer, uz objekte "cin" i "cout" koristili smo "magične" operatore "<<" i ">>", objekte tipa "string" i "complex" mogli smo sabirati pomoću operatora "+", dok smo objekte tipa "string" ili "vector" mogli indeksirati pomoću uglastih zagrada. Sve ovo ne možemo uraditi sa primjercima klase "Datum" (a pravo da kažemo, sa primjercima ove klase takve operacije i nemaju smisla). S druge strane, u kasnijim poglavljima ćemo vidjeti da je također moguće definirati smisao gotovo svih operatora primijenjenim nad primjercima klase koju definiramo. Očigledno, ima još dosta stvari koje treba da naučimo o klasama, ali je na ovom mjestu bitno shvatiti da klase poput "string", "vector" ili "istream" nisu ni po čemu posebne u odnosu na bilo koju klasu koju možemo samostalno razviti, niti su objekti poput "cin" i "cout" i po čemu posebni u odnosu na bilo koji drugi objekat!

Već je rečeno da je dobra praksa sve atribute držati isključivo u privatnom dijelu klase. S druge strane, metode su uglavnom u javnom dijelu klase. Može se postaviti pitanje ima li smisla definirati neku od metoda kao privatnu metodu. Odgovor je potvrđan, ukoliko smatramo da neka od metoda može biti od koristi da se poziva iz neke druge metode, ali *ne želimo da dopustimo da se ta metoda poziva iz ostatka programa*. Na primjer, pretpostavimo da smo u prethodnom primjeru klase "Datum" metodu "VratiImeMjeseca" deklarirali u privatnom dijelu klase, kao u sljedećoj deklaraciji:

```
class Datum {
    int dan, mjesec, godina;
    char *VratiImeMjeseca() const;
public:
    void Postavi(int d, int m, int g);
    void Sljedeci();
    void Ocitaj(int &d, int &m, int &g) const {
        d = dan; m = mjesec; g = godina;
    }
    int VratiDan() const { return dan; }
    int VratiMjesec() const { return mjesec; }
    int VratiGodinu() const { return godina; }
    void Ispisi() const {
        cout << dan << ". " << mjesec << ". " << godina << ".";
    }
    void IspisiLijepo() const;
};
```

U tom slučaju, metoda "VratiImeMjeseca" bi se sasvim normalno mogla pozivati iz metode "IspisiLijepo" (kao i iz ma koje druge metode klase "Datum"), ali se ne bi mogla pozivati ni odakle drugdje (za ostatak programa ona praktično ne bi ni postojala).

Razumije se da ukoliko imamo više primjeraka iste klase, tada svaki primjerak ima *zasebne primjerke* svih svojih atributa. Na primjer, ukoliko su "d1" i "d2" dvije promjenljive tipa "Datum", tada se "d1.dan" i "d2.dan" ponašaju kao dvije *potpuno odvojene promjenljive*, što je i logično. Međutim, nekada je potrebno imati atribute koji će zajednički dijeliti *svi primjeri iste klase*. Takvi atribute nazivaju se *statički atributi*, i deklariraju se uz pomoć kvalifikatora "**static**". Neka, na primjer, imamo klasu "Student" koja je karakterizirana atributima "ime", "prezime", "indeks", "ocjene" i "prosjek" (sa očiglednim značenjima), i neka je dalje potrebno imati informaciju o *ukupnom broju studenata*.

Prepostavimo da ovoj informaciji trebaju pristupati svi primjeri klase “Student”, bez obzira gdje se oni nalazili unutar programa. Očigledno rješenje je definirati *globalnu promjenljivu* “broj_studenata” koja će čuvati traženu informaciju. Međutim, već znamo da upotreba globalnih promjenljivih tipično donosi više problema nego što rješava. Naime, prepostavimo da samo metode klase “Student” trebaju pristupati ovoj informaciji. Ukoliko “broj_studenata” deklariramo kao globalnu promjenljivu, njoj će moći pristupati svako, i ko treba i ko ne treba (što je poznato svojstvo globalnih promjenljivih), tako da postoji nehotična mogućnost neovlaštenog pristupa. Bolja ideja je definirati “broj_studenata” kao atribut klase “Student”. Međutim, na taj način, svaki primjerak klase “Student” mogao bi postaviti različite vrijednosti ovog atributa, bez obzira što je jasno da je ukupan broj studentata svojstvo koje je očigledno zajedničko za sve primjerke klase “Student”. Pravo rješenje je definirati “broj_studenata” kao *statički atribut klase* “Student”, kao u sljedećoj deklaraciji (deklaracije eventualnih metoda klase nam u ovom primjeru nisu interesantne):

```
class Student {
    static int broj_studenata;
    char ime[30], prezime[30];
    int indeks;
    int ocjene[50];
    double prosjek;
public:
    ...
    // Nebitno
};
```

Uz ovakvu deklaraciju, ukoliko prepostavimo da su “s1” i “s2” dva primjerka klase “Student”, tada se “s1.broj_studenata” i “s2.broj_studenata” ne ponašaju kao dvije različite, nego kao *jedna te ista promjenljiva*. Drugim riječima, svi primjeri klase “Student” dijele istu vrijednost atributa “broj_studenata”, i njegova eventualna promjena izvršena iz recimo neke metode klase “Student” primijenjene nad nekim konkretnim objektom biće vidljiva svim drugim primercima iste klase. Atribut “broj_studenata” se u potpunosti ponaša kao *globalna promjenljiva*, osim što se može koristiti samo na mjestima gdje su privatni atributi klase “Student” dostupni (tj. unutar metoda ove klase, i njenih prijateljskih funkcija). Međutim, deklaracija statičkog atributa unutar klase samo *najavljuje* njegovo postojanje, ali ga i ne *definira* (odnosno, ne rezervira memoriski prostor za njegovo pamćenje). Svaki statički atribut se mora na nekom mjestu u programu na globalnom nivou (dakle, izvan definicija funkcija ili klase) posebno definirati (pri čemu se tom prilikom može eventualno izvršiti njegova inicijalizacija) deklaracijom poput sljedeće:

```
static int Student::broj_studenata = 10;
```

Razlog za ovu zavrzelam vezan je za činjenicu da se C++ programi mogu podijeliti u više različitih programskih datoteka koje se mogu neovisno kompajlirati i na kraju povezati u jedinstven program, o čemu ćemo kasnije posebno govoriti. Ipak, bez obzira na činjenicu da se statički atributi definiraju izvan deklaracije klase, poput definicije običnih globalnih promjenljivih, pravo pristupa im je strogo ograničeno, kao što smo već objasnili.

Statički atributi se gotovo isključivo deklariraju u privatnoj sekciji klase. U načelu, nije zabranjeno izvršiti njihovu deklaraciju u javnoj sekciji klase, ali se na taj način gubi smisao njihovog uvođenja. Na primjer, prepostavimo da je statički atribut “broj_studenata” deklariran u javnoj sekciji klase “Student”, i da su “s1” i “s2” dva konkretna primjerka klase “Student”. Tada bismo atributu

“broj_studenata” mogli pristupiti iz *bilo kojeg dijela programa* iza mesta njegove deklaracije konstrukcijama poput “s1.broj_studenata” ili “s2.broj_studenata” (u oba slučaja pristup se odnosi na *istu jedinku*). Čak bi i pristup pomoću izraza “Student::broj_studenata” izvan neke od metoda klase “Student” bio posve legalan, s obzirom da je *potpuno svejedno* na koji se konkretni primjerak klase “Student” ovaj atribut odnosi (s obzirom da je on zajednički za sve primjerke klase). Drugim riječima, ovaj atribut bi se mogao koristiti na identičan način kao i na kakva globalna promjenljiva, jedino bi nas sintaksa upućivala da se radi o podatku koji je na neki način logički vezan za klasu “Student”.

Funkcije članice također mogu biti statičke. Kao što su statički atributi u suštini klasične globalne promjenljive, samo sa ograničenim pravima pristupa, tako su i statičke funkcije članice zapravo klasične funkcije sa ograničenim pravima pristupa. Unutar statičkih funkcija članica ne postoji pokazivač “**this**”, tako da statičke funkcije članice *ne mogu znati* nad kojim su objektom pozvane. Zbog toga, statičke funkcije članice ne mogu direktno pristupati atributima objekta nad kojim su pozvane, osim eventualno statičkim atributima (koji su zajednički za sve primjerke iste klase), niti mogu pozivati druge funkcije članice klase, osim eventualno drugih statičkih funkcija članica. Za statičke funkcije članice je zapravo *potpuno nebitno* nad kojim se objektom pozivaju. One se koriste u situacijama kada želimo implementirati neku akciju koja ne ovisi od toga nad kojim je konkretnim objektom pozvana, a ne želimo za tu svrhu upotrijebiti klasičnu funkciju, s obzirom da smatramo da je ta akcija po svojoj prirodi ipak tjesno vezana za samu klasu.

Razmotrimo statičke funkcije članice na konkretnom primjeru klase “Datum” koju smo razvili. Ukoliko uporedimo metode “Postavi” i “Sljedeci”, primjećujemo da se u obje metode javlja potreba za utvrđivanjem broja dana u određenom mjesecu, zbog čega se u obje metode javlja ista sekvenca naredbi. Dupliranje kôda bismo mogli izbjegći ukoliko bismo definirali posebnu funkciju nazvanu recimo “BrojDana” koja bi vraćala broj dana u mjesecu koji se prenosi kao parametar (morali bismo kao parametar prenositi i godinu, jer od nje može zavisiti koliko dana ima februar). Funkcija “BrojDana” bi, sasvim lijepo, mogla biti klasična funkcija. Međutim, kako je ona razvijena za potrebe klase “Datum”, prirodnije bi bilo da ona bude *sastavni dio klase*, odnosno da bude njena funkcija članica. Dalje, s obzirom da ova funkcija ne treba pristupati atributima klase (ona sve neophodne informacije dobija putem parametara), možemo je učiniti *statičkom funkcijom članicom*. I zaista, potpuno je svejedno nad kojim će se objektom funkcija “BrojDana” pozvati (npr. broj dana u martu je uvijek 31, neovisno od razmatranog datuma). Stoga ćemo izmijeniti deklaraciju klase “Datum” da izgleda ovako (pri tome, dio deklaracije koji ostaje isti nismo ponovo prepisivali, radi uštede prostora):

```
class Datum {
    int dan, mjesec, godina;
    char *VratiImeMjeseca() const;
    static int BrojDana(int mjesec, int godina);
public:
    ...
    // Javni dio ostaje isti kao i ranije...
};
```

Metodu “BrojDana” smo stavili u privatnu sekciju klase, čime je onemogućeno njen pozivanje izvan funkcija članica klase. Naravno, ovu metodu treba i implementirati. Njena implementacija, kao i izmijenjene implementacije metoda “Postavi” i “Sljedeci”, koje se oslanjaju na njen postojanje, mogće biti izgledati recimo ovako:

```
int Datum::BrojDana(int mjesec, int godina) {
    int broj_dana[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if(godina % 4 == 0 && godina % 100 != 0 || godina % 400 == 0)
        broj_dana[1]++;
    return broj_dana[mjesec - 1];
```

```

}

void Datum::Postavi(int dan, int mjesec, int godina) {
    if(godina < 1 || dan < 1 || mjesec < 1 || mjesec > 12
        || dan > BrojDana(mjesec, godina)
        throw "Neispravan datum!\n";
    Datum::dan = dan; Datum::mjesec = mjesec; Datum::godina = godina;
}

void Datum::Sljedeci() {
    dan++;
    if(dan > BrojDana(mjesec, godina)) {
        dan = 1; mjesec++;
    }
    if(mjesec > 12) {
        mjesec = 1; godina++;
    }
}

```

U suštini, sa aspekta funkcioniranja, sve bi isto radilo i da metoda “BrojDana” nije bila statička. Međutim, prirodnije ju je definirati kao statičku, s obzirom da ona zaista ne mora znati nad kojim objektom djeluje. Kasnije ćemo vidjeti i da postoje neki konteksti u kojima je moguće primijeniti samo statičke metode, a ne i nestatičke, tako da je dobro da se odmah naviknemo na njihovu upotrebu. Pored toga, činjenica da je metoda “BrojDana” proglašena za statičku, čini je i neznatno efikasnijom, jer se u tom slučaju njoj ne prenosi pokazivač “**this**” kao skriveni parametar.

S obzirom da je metoda “BrojDana” definirana kao privatna, njen pozivanje izvan funkcija članice klase nije dozvoljeno. Da je ova metoda definirana u javnoj sekciji klase, ona bi se mogla pozivati iz bilo kojeg dijela programa konstrukcijama poput “`d.BrojDana(2, 2005)`” gdje je “`d`” neka konkretna promjenljiva tipa “`Datum`” (koja zapravo uopće ne utiče na samo izvršavanje funkcije), odnosno konstrukcijama poput “`Datum::BrojDana(2, 2005)`” s obzirom da je posve nebitno nad kojim je konkretnim primjerkom klase “`Datum`” metoda “`BrojDana`” pozvana. Dakle, metoda “`BrojDana`” bi se mogla koristiti kao i obična funkcija, samo bi nas sintaksni prefiks “`Datum::`” upućivao da se radi o funkciji čija je svrha postojanja tijesno vezana za klasu “`Datum`”.

Unutar deklaracija klase mogu se naći i deklaracije *korisnički imenovanih tipova* (“**typedef**” deklaracije), *pobrojanih tipova* (“**enum**” deklaracije), pa čak i deklaracije drugih struktura ili klasa. U slučaju deklaracije jedne klase unutar deklaracije druge klase govorimo o tzv. *ugniježdenim klasama* (engl. *nested classes*). Svim ovakvim tvorevinama unutar funkcija članica klase možemo pristupati na uobičajeni način, kao da su deklarirane izvan klase. Međutim, njihovo korištenje izvan funkcija članica klase (ili prijateljskih funkcija) moguće je samo ukoliko su deklarirane u javnoj sekciji klase, i to uz obavezno navođenje imena klase i operatora “`::`”. Na primjer, djeluje razumno u klasu “`Datum`” dodati i sljedeće deklaracije:

```

class Datum {
    ...
    // Isto kao i ranije...
public:
    enum Mjeseci {Januar, Februar, Mart, April, Maj, Juni, Juli, August,
        Septembar, Oktobar, Novembar, Decembar};
    enum Dani {Ponedjeljak, Utorak, Srijeda, Cetvrtak, Petak, Subota,
        Nedjelja};
    ...
    // Isto kao i ranije...
};

```

Uz ovakvu deklaraciju, tipove "Mjeseci" i "Dani", kao i pobrojane konstante ovih tipova (poput "Maj", "Petak", itd.) unutar funkcija članica klase možemo koristiti na isti način kao i da su deklarirani izvan klase. Međutim, ukoliko im želimo pristupiti iz drugih dijelova programa, moramo koristiti konstrukcije poput "Datum::Mjeseci", "Datum::Dani", "Datum::Maj", "Datum::Petak", itd. Pri tome je pristup iz drugih dijelova programa moguć samo zahvaljujući činjenici da smo navedene deklaracije smjestili u javnu sekciju klase. U suprotnom, svi ovi identifikatori bili bi nedostupni izvan metoda i funkcija prijatelja klase. Slijedi da je deklariranje korisnički imenovanih i pobrojanih tipova koji se samo internu koriste za potrebe implementacije klase, a za koje ostatak programa ne treba da zna, najbolje izvesti upravo u privatnoj sekciji klase.

Objektno zasnovani pristup razvoju demonstriraćemo na još jednoj interesantnoj klasi koju ćemo razviti. Pretpostavimo da nam je potrebna klasa "Vektor" koja će opisivati vektor u prostoru. Poznato je da je svaki prostorni vektor definiran sa tri koordinate "x", "y" i "z" koje će ujedno biti i atributi ove klase. U skladu sa principima dobrog objektno orijentiranog dizajna, ovi atributi će biti *privatni* (bez obzira što u ovom slučaju ima smisla postaviti *bilo koji atribut* na *bilo koju vrijednost*), a uvećemo metode "Postavi", "Ocitaj", "Vrati_x", "Vrati_y" i "Vrati_z" koje su analogne sličnim metodama klase "Datum" (na ovaj način onemogućavamo da se koordinate vektora postavljaju odvojeno jedna od druge, već samo sve tri skupa, što je mnogo smislenije). Pored toga, uvećemo i metodu "Ispisi" koja ispisuje vektor kao skupinu od tri koordinate unutar vitičastih zagrada razdvojene zarezima (npr. kao "{3, 4, 2}"), metodu "Duzina" koja vraća kao rezultat dužinu vektora, kao i metode "MnoziSaSkalarom" i "SaberiSa" koje množe vektor sa skalarom (zadanim kao parametar) odnosno sabiraju vektor sa drugim vektorom (također zadanim kao parametar). Na taj način dobijamo sljedeću deklaraciju klase, u kojoj su sve metode dovoljno jednostavne da mogu biti implementirane unutar same deklaracije klase:

```
class Vektor {
    double x, y, z;
public:
    void Postavi(double x, double y, double z) {
        Vektor::x = x; Vektor::y = y; Vektor::z = z;
    }
    void Ocitaj(double &x, double &y, double &z) const {
        x = Vektor::x; y = Vektor::y; z = Vektor::z;
    }
    void Ispisi() const {
        cout << "{" << x << "," << y << "," << z << "}";
    }
    double Vrati_x() const { return x; }
    double Vrati_y() const { return y; }
    double Vrati_z() const { return z; }
    double Duzina() const { return sqrt(x * x + y * y + z * z); }
    void PomnoziSaSkalarom(double s) { x *= s; y *= s; z *= s; }
    void SaberiSa(const Vektor &v) { x += v.x; y += v.y; z += v.z; }
};
```

Primjer je dovoljno jasan da ne traži posebna objašnjenja. Interesantan je jedino način na koji se koriste metode "PomnoziSaSkalarom" i "SaberiSa", koji je ilustriran u sljedećem primjeru upotrebe klase "Vektor" (koji će na ekranu ispisati "7.07106", "{15,20,25}" i "{22,23,27}"), iz čega vidimo da se ove dvije metode na izvjestan način ponašaju poput operatora "*=" i "+=" (što bi, između ostalog, trebalo da bude jasno i iz načina kako su one definirane):

```

Vektor v1, v2;
v1.Postavi(3, 4, 5);
v2.Postavi(7, 3, 2);
cout << v1.Duzina() << endl;
v1.PomnoziSaSkalarom(5);
v1.Ispisi();
cout << endl;
v1.SaberisSa(v2);
v1.Ispisi();

```

Veoma je bitno da princip sakrivanja podataka i enkapsulacije strogo naređuje da, osim ukoliko ne postoji veoma jak razlog za suprotno, sve atribute klase treba čuvati kao privatne. Na primjer, čak i ukoliko želimo da imamo mogućnost neovisnog postavljanja “x”, “y” i “z” koordinata vektora, to ne treba učiniti tako što ćemo atribute “x”, “y” i “z” prosto proglašiti javnim. Umjesto toga, potrebno je uvesti tri nove trivijalne metode “Postavi_x”, “Postavi_y” i “Postavi_z” koje će respektivno postavljati vrijednosti atributa “x”, “y” odnosno “z” na vrijednost zadatu parametrom. Tako ćemo ukoliko želimo da postavimo “x” koordinatu vektora “v” na vrijednost “5”, umjesto “v.x = 5” pisati “v.Postavi_x(5)”. Mada ovo djeluje kao potpuno suvišna komplikacija, postoje veoma jaki razlozi za ovaku praksu. Pretpostavimo da se u budućnosti pojavi potreba da u potpunosti promijenimo internu organizaciju koordinata vektora “v”, i da npr. umjesto u tri cjelobrojna atributa koordinate čuvamo u atributu tipa cjelobrojnog niza (nazvanog npr. “koordinate”). Tada bismo u slučaju da su atributi bili javni, svaki pristup tipa “v.x” morali zamijeniti pristupom tipa “v.koordinate[0]” (na primjer, “v.koordinate[0] = 5”). S druge strane, ukoliko imamo metode poput “Postavi_x”, dovoljno je samo promijeniti njihovu implementaciju. U ostatku programa se i dalje može bez problema koristiti konstrukcija poput “v.Postavi_x(5)”. Tačno je da objektno zasnovani pristup traži više pisanja, ali takav pristup nije ni namijenjen za pisanje *malih programi*. U ovom pristupu se mnogo više misli na budućnost i na činjenicu da će se svaki program koji ičemu vrijedi prije ili kasnije morati prepravljati, nadograđivati i usavršavati. Objektno zasnovano i objektno orijentirano programiranje nam priprema teren sa ciljem da eventualne kasnije intervencije na programu budu što je god moguće bezbolnije.

U klasi “Vektor”, metode “PomnoziSaSkalarom” i “SaberisSa” napisane su kao metode koje ne vraćaju nikakav rezultat, što može djelovati prirodno. Ipak, zbog te činjenice, neke također prirodne konstrukcije poput “v2 = v1.PomnoziSaSkalarom(5)” nisu izvodljive (metoda ne vraća ništa kao rezultat što bismo mogli pridružiti drugom vektoru). Stoga bismo veoma interesantan efekat mogli ostvariti ukoliko bismo izmijenili ove dvije metode tako da kao rezultat vraćaju *izmijenjeni vektor*. Tako bi konstrukcije poput “v1.PomnoziSaSkalarom(5)” i dalje ostale legalne (s obzirom da vraćeni rezultat uvijek možemo ignorirati), ali bi time konstrukcije “v2 = v1.PomnoziSaSkalarom(5)” pa čak i lančane (kaskadne) konstrukcije poput “v1.PomnoziSaSkalarom(5).Ispisi()” postale posve ispravne. Naime, u ovom drugom primjeru, “v1.PomnoziSaSkalarom(5)” će kao rezultat vratiti objekat tipa “Vektor”, na koji se onda može primijeniti metoda “Ispisi”. Tako dobijamo sljedeću verziju klase “Vektor”, u kojoj radi kratkoće nećemo ponovo pisati definicije atributa i ostalih metoda klase koje su ostale iste:

```

class Vektor {
    ... // Ovdje treba umetnuti definicije atributa i preostalih metoda
    Vektor PomnoziSaSkalarom(double s) {
        x *= s; y *= s; z *= s;
        return *this;
    }
    Vektor SaberiSa(const Vektor &v) {
        x += v.x; y += v.y; z += v.z;
        return *this;
    }
}

```

```
};
```

Primijetimo kako je ovom prilikom pokazivač “**this**“ iskorišten za vraćanje izmijenjenog objekta iz metode (ne zaboravimo da dereferencirani “**this**“ pokazivač predstavlja upravo objekat nad kojim je metoda pozvana). Sljedeća sekvenca naredbi dovodi do očekivanog ispisa (tj. ispisa “{15, 20, 25}“ i “{22, 23, 27}”), tako da *izgleda* da prethodno rješenje radi korektno:

```
Vektor v1, v2;
v1.Postavi(3, 4, 5);
v2.Postavi(7, 3, 2);
v1.PomnoziSaSkalarom(5).Ispisi();
cout << endl;
v1.SaberisSa(v2).Ispisi();
```

Međutim, sljedeći primjer će nas uvjeriti da sve nije baš u potpunosti u skladu sa očekivanjima. Naime, mada bi se moglo očekivati da će sljedeća sekvenca naredbi dva puta ispisati “{22, 23, 27}”, biće jedanput ispisano “{22, 23, 27}“, a drugi put “{15, 20, 25}”:

```
Vektor v1, v2;
v1.Postavi(3, 4, 5);
v2.Postavi(7, 3, 2);
v1.PomnoziSaSkalarom(5).SaberisSa(v2).Ispisi();
cout << endl;
v1.Ispisi();
```

Problem nastaje kod vraćanja rezultata iz funkcija. Naime, u normalnim okolnostima kada god sa “**return**“ naredbom vratimo neku vrijednost iz funkcije, stvara se novi objekat čiji tip odgovara tipu povratne vrijednosti, u koji se kopira izraz koji je naveden iza naredbe “**return**”, i koji predstavlja rezultat funkcije. Ovdje je važno da shvatimo da kada se iza naredbe “**return**“ nade *ime nekog objekta*, ono što će biti vraćeno iz funkcije nije taj objekat, nego *njegova identična kopija*. Ovo je neophodno zbog toga što stvarni objekat može i da prestane postojati nakon završetka funkcije, npr. ukoliko on predstavlja lokalnu promjenljivu deklariranu unutar funkcije. Ovakvo ponašanje nije svojstveno samo klasama i funkcijama članicama klase, nego predstavlja *opće pravilo kako se ma kakve vrijednosti vraćaju iz ma kakve funkcije*. Mada o ovakovom ponašanju nismo do sada posebno razmišljali, ono nam nikada do sada nije smetalo. Međutim, razmotrimo u ovom kontekstu značenje sljedeće konstrukcije:

```
v1.PomnoziSaSkalarom(5).SaberisSa(v2).Ispisi();
```

U ovom slučaju, konstrukcija “`v1.PomnoziSaSkalarom(5)`“ množi vektor “`v1`“ sa brojem “5“ (što se odražava na sadržaj vektora “`v1`“ koji poprima vrijednost “{15, 20, 25}”), i vraća kao rezultat *vrijednost* izmijenjenog vektora. Međutim, vraćeni vektor nije sam objekat “`v1`“, nego neki privremeni objekat tipa “`Vektor`”, koji je njegova potpuno identična bezimena kopija! Na tu kopiju (koja također ima vrijednost “{15, 20, 25}”) primjenjuje se metoda “`SaberisSa(v2)`“ koja mijenja vrijednost tog bezimenog vektora (koja sad postaje “{22, 23, 27}”), i vraća kao rezultat *novi bezimeni vektor* koji ima vrijednost “{22, 23, 27}”. Nad ovim vektorm primjenjuje se metoda “`Ispisi()`“ koja naravno ispisuje “{22, 23, 27}”. Međutim, samo se prva metoda “`PomnoziSaSkalarom(5)`“ zaista izvršila nad vektrom “`v1`”, dok su se sve ostale metode izvršile nad nekim bezimenim vektorima koje predstavljaju kopije objekata koje su vraćene iz funkcije! Stoga vektor “`v1`“ zadržava vrijednost kakvu je imao nakon primjene metode “`PomnoziSaSkalarom(5)`”, odnosno “{15, 20, 25}”. Zbog toga će sljedeći poziv “`v1.Ispisi()`“ ispisati upravo “{15, 20, 25}”.

Činjenica da se iz funkcije uvijek vraća *kopija* objekta koji je naveden iza “**return**“ naredbe a ne sam objekat, nije nam do sada smetala (naprotiv, bila je veoma korisna), s obzirom da nikada do sada nismo imali potrebu da nad objektom vraćenim iz funkcije primijenimo neku akciju koja bi trebala da promijeni

sam vraćeni objekat. Međutim, kao što vidimo, uvođenjem *klasa* omogućeno je definiranje takvih objekata nad kojima se mogu primjenjivati akcije koje mogu *mijenjati* sadržaj objekta (tj. metode mutatori). Stoga, ovakav način vraćanja vrijednosti iz funkcija može napraviti probleme ukoliko se metode mutatori pozivaju *ulančano* (tj. kad se na rezultat jedne metode ponovo primjenjuje druga metoda). Šta da se radi? Izlaz iz ove situacije je veoma jednostavan: metode “*PomnoziSaSkalarom*“ i “*SaberisiSa*“ treba da umjesto samog objekta tipa “*Vektor*”, koji je *kopija* objekta koji želimo da vratimo, vrate *referencu* na objekat tipa “*Vektor*” koja predstavlja *alternativno ime* za objekat koji vraćamo (odnosno *prikriveni pokazivač* na objekat koji vraćamo). Stoga je u klasi “*Vektor*“ dovoljno izvršiti samo neznatnu izmjenu, kao u sljedećoj deklaraciji:

```
class Vektor {
    ...
    // Ovdje treba umetnuti definicije atributa i preostalih metoda
    Vektor &PomnoziSaSkalarom(double s) {
        x *= s; y *= s; z *= s;
        return *this;
    }
    Vektor &SaberisiSa(const Vektor &v) {
        x += v.x; y += v.y; z += v.z;
        return *this;
    }
};
```

Sada će ulančani poziv poput “*v1.PomnoziSaSkalarom(5).SaberisiSa(v2)*“ zaista raditi u skladu sa očekivanjima. Prvi poziv metode “*PomnoziSaSkalarom(5)*“ izmijeniće vektor “*v1*“ i vratiti kao rezultat referencu na izmijenjeni vektor “*v1*“ koja samo predstavlja *njegovo alternativno ime* (treba li reći *prerušeni pokazivač na njega?*) tako da će sljedeći poziv metode “*SaberisiSa(v2)*“ zaista djelovati na vektor “*v1*”, što je u skladu sa očekivanjima.

U prethodnim primjerima smo koristili sintaksu poput “*v1.SaberisiSa(v2)*“ da saberemo vektore “*v1*“ i “*v2*” (ne zaboravimo da ova operacija *mijenja* sam vektor “*v1*“). Mada je ova sintaksa u potpunosti u duhu objektno orijentirane filozofije (sjetimo se kakvo je objektno orijentirano gledište na interpretaciju izraza “*5 + 3*”), samo će najradikalniji zagovornici striktnog objektno orijentiranog pristupa insistirati na upotrebi ovakve sintakse čak i u slučaju takvih operacija kao što je sabiranje, koje bi bilo prirodnije izraziti čisto proceduralnom sintaksom poput “*ZbirVektora(v1, v2)*“ (sjetimo se proceduralnog gledišta na interpretaciju izraza “*5 + 3*”), pogotovo u kontekstu dodjele rezultata nekom trećem vektoru (npr. “*v3 = ZbirVektora(v1, v2)*“). Međutim, bitno je naglasiti da time što smo definirali tip “*Vektor*“ kao klasu nismo izgubili pravo da pišemo *obične funkcije* (koje nisu funkcije članice) koje kao parametre primaju objekte tipa “*Vektor*”, i koje vraćaju objekte tipa “*Vektor*“ kao rezultat. Stoga nam niko ne brani da napišemo običnu funkciju “*ZbirVektora*“ koju ćemo upravo pozivati na gore navedeni način (tj. kao standardnu funkciju, a ne primjenjenu nad nekim objektom). Mada izvjesni radikalisti smatraju da pisanje ovakvih funkcija nije u duhu objektno orijentiranog programiranja, većina ipak ima mišljenje da pisanje ovakvih funkcija u *umjerenoj količini* ne šteti objektno orijentiranoj filozofiji, pogotovo ukoliko vodi ka intuitivno prirodnijoj sintaksi. Stoga ćemo napisati funkciju “*ZbirVektora*“ koja će koristiti gore opisanu sintaksu. Međutim, moramo voditi računa da tako napisana funkcija neće imati privilegiju pristupa privatnim atributima klase “*Vektor*“ (upravo zbog toga što nije funkcija članica nego obična funkcija). Stoga ćemo ovu funkciju morati napisati na indirektan način, koristeći funkcije članice “*Postavi*”, “*Vrati_x*”, “*Vrati_y*“ i “*Vrati_z*“ pomoću kojih ipak možemo posredno pristupiti atributima klase “*Vektor*” (podsjetimo se da smo kvalifikator “**const**” uz povratnu vrijednost stavili sa ciljem da spriječimo besmislene konstrukcije poput “*ZbirVektora(v1, v2) = v3*” u kojima se privremeni objekat vraćen kao rezultat funkcije koristi kao l-vrijednost):

```

const Vektor ZbirVektora(const Vektor &v1, const Vektor &v2) {
    Vektor v3;
    v3.Postavi(v1.Vrati_x() + v2.Vrati_x(), v1.Vrati_y() + v2.Vrati_y(),
    v1.Vrati_z() + v2.Vrati_z());
    return v3;
}

```

Neko bi mogao prigovoriti da i sintaksa poput “`v.Duzina()`“ za nalaženje dužine vektora `v` nije posve prirodna, već da bi bilo prirodnije koristiti sintaksu poput “`Duzina(v)`“. Međutim, ova primjedba je manje osnovana nego u slučaju sabiranja, i već donekle kvari duh objektno orijentirane filozofije. Naime, već smo vidjeli da informacije o objektu dobijamo pozivom metoda nad njim, tako da informaciju o “`x`“ koordinati vektora “`v`“ dobijamo pozivom metode “`Vrati_x`“ nad vektorom “`v`”, tj. pozivom “`v.Vrati_x()`“. Kako je i dužina vektora izvjesna informacija o objektu vektor (koja doduše nije atribut ali se izvodi iz atributa), nema bitnog razloga da za dobijanje ove informacije koristimo drugačiju sintaksu. Korištenjem jedinstvene sintakse ne pravimo konceptualnu razliku između npr. koordinate vektora i njegove dužine (oba pojma predstavljaju informacije o vektoru koje dobijamo na isti način, tj. primjenom istih sintaksnih konstrukcija). Ipak, radi demonstracije nekih činjenica napisaćemo i *običnu funkciju* “`Duzina`“ koja prima kao parametar vektor, a vraća njegovu dužinu, tako da se može koristiti sintaksa “`Duzina(v)`“. Ovdje želimo istaći činjenicu da može postojati funkcija članica i obična funkcija sa istim imenom, bez opasnosti da ih kompjajler pomiješa, s obzirom da se one *razlikuju po načinu poziva* (funkcije članice se uvijek pozivaju nad nekim objektom, osim eventualno iz druge funkcije članice iste klase). Stoga bi (običnu) funkciju “`Duzina`“ mogli napisati ovako:

```

double Duzina(Vektor v) {
    return v.Duzina();
}

```

Primijetimo da je definicija funkcije “`Duzina`“ trivijalna, jer smo prosto iskoristili (istoimenu) metodu “`Duzina`“ nad objektom koji je proslijeden kao parametar, koja upravo radi ono što nam i treba.

Vratimo se na implementaciju funkcije “`ZbirVektora`“. Možemo reći da smo imali sreće što klasa “`Vektor`” ima metode “`Vrati_x`”, “`Vrati_y`“ i “`Vrati_z`“, koje ipak omogućavaju da jednostavno pristupimo vrijednostima koordinata vektora. Da nismo imali ove metode nego da smo morali koristiti metodu “`Ocitaj`”, implementacija funkcije “`ZbirVektora`“ bila bi mnogo komplikovanija:

```

const Vektor ZbirVektora(const Vektor &v1, const Vektor &v2) {
    double v1_x, v1_y, v1_z, v2_x, v2_y, v2_z;
    v1.Ocitaj(v1_x, v1_y, v1_z);
    v2.Ocitaj(v2_x, v2_y, v2_z);
    Vektor v3;
    v3.Postavi(v1_x + v2_x, v1_y + v2_y, v1_z + v2_z);
    return v3;
}

```

Da nije postojala ni metoda “`Ocitaj`”, ruke bi nam bile potpuno vezane, i uopće ne bismo bili u stanju da napišemo ovu funkciju. S druge strane, jasno je da bi se najprirodnija verzija funkcije “`ZbirVektora`“ mogla napisati kada bi ova funkcija *imala pravo pristupa privatnim atributima* klase “`Vektor`”. Ona bi se tada mogla napisati identično kao i istoimena funkciji iz prethodnog poglavlja (dok je tip “`Vektor`“ bio samo *struktura*, a ne *klasa sa privatnim atributima*):

```

const Vektor ZbirVektora(const Vektor &v1, const Vektor &v2) {
    Vektor v3;
    v3.x = v1.x + v2.x; v3.y = v1.y + v2.y; v3.z = v1.z + v2.z;
    return v3;
}

```

}

Na svu sreću, tako nešto je *moguće*. Naime, unutar deklaracije klase moguće je izvjesne obične funkcije (koje nisu funkcije članice klase) proglašiti *prijateljima klase*, čime one dobijaju pravo pristupa privatnim atributima i metodama te klase. Neku funkciju je moguće proglašiti prijateljem klase tako što se unutar deklaracije klase navede ključna riječ “**friend**”, a zatim prototip funkcije koju proglašavamo prijateljem. Ovim funkcija ne postaje funkcija članica (tj. i dalje se poziva kao obična funkcija), samo dobija pravo pristupa privatnim atributima klase. Slijedi deklaracija klase “Vektor” koja proglašava funkciju “ZbirVektora” prijateljem klase “Vektor”, što omogućava njenu implementaciju na gore navedeni način:

```
class Vektor {
    double x, y, z;
public:
    void Postavi(double x, double y, double z) {
        Vektor::x = x; Vektor::y = y; Vektor::z = z;
    }
    void Ocitaj(double &x, double &y, double &z) const {
        x = Vektor::x; y = Vektor::y; z = Vektor::z;
    }
    void Ispisi() const {
        cout << "{" << x << "," << y << "," << z << "}";
    }
    double Vrati_x() const { return x; }
    double Vrati_y() const { return y; }
    double Vrati_z() const { return z; }
    double Duzina() const { return sqrt(x * x + y * y + z * z); }
    Vektor &PomnoziSaSkalarom(double s) {
        x *= s; y *= s; z *= s;
        return *this;
    }
    Vektor &SaberisiSa(const Vektor &v) {
        x += v.x; y += v.y; z += v.z;
        return *this;
    }
    friend const Vektor ZbirVektora(const Vektor &v1, const Vektor &v2);
};
```

Bitno je uočiti smisao ključne riječi “**friend**”. Bez navođenja ove ključne riječi, prototip funkcije “ZbirVektora” bio bi shvaćen kao prototip *funkcije članice* koja se primjenjuje nad nekim objektom tipa “Vektor”, a kojoj se pored toga još prenose dva vektora kao parametri (tj. funkcije članice koja bi se koristila u formi poput “v3.ZbirVektora(v1, v2)” gdje su “v1”, “v2” i “v3” neki objekti tipa “Vektor”). Tako, ključna riječ “**friend**” zapravo govori da se ne radi o funkciji članici nego o običnoj funkciji, ali kojoj se daju *sva prava po pitanju pristupa privatnoj sekciji klase* kakvu imaju i funkcije članice klase (takve funkcije nazivamo *prijateljske funkcije*). Prijateljske funkcije se također mogu implementirati odmah unutar deklaracije klase, pri čemu se tada, slično kao u slučaju funkcija članica klase, one automatski proglašavaju za umetnute funkcije (tj. podrazumijeva se kvalifikator “**inline**”).

Neka klasa može bilo koju funkciju proglašiti svojim prijateljem. Naravno, to “prijateljstvo” ima smisla jedino ukoliko ta funkcija ima neke veze sa tom klasom, što se tipično dešava u tri slučaja: ako funkcija *prima parametre* koji su tipa te klase (ili pokazivača na tu klasu, ili reference na tu klasu), ako funkcija *vraća rezultat* koji je tipa te klase, ili ako funkcija *deklariira lokalnu varijablu* tipa te klase (naravno, moguće je da se dešava i sve ovo skupa, kao u primjeru navedene funkcije “ZbirVektora”).

Dugo vremena su postojale rasprave o tome da li prijateljske funkcije štete objektno orijentiranoj filozofiji. Jedan argument je bio da one *narušavaju sakrivanje podataka*. To mišljenje je odbačeno zbog činjenice da nijedna funkcija sama sebe ne može proglašiti prijateljem neke klase (i na taj način neovlašteno pristupiti njenim atributima), već upravo sama klasa definira kojim funkcijama "može vjerovati". Ako klasa "ima povjerenje" u svoje funkcije članice, zašto ne bi mogla imati povjerenje i u obične funkcije koje sama specificira? Drugi argument je da prijateljske funkcije *podstiču korištenje sintakse koja nije u duhu objektno orijentiranog programiranja*. Ovo je doduše tačno, ali je činjenica da objektno orijentiranu filozofiju ne čini *sintaksa*, već četiri osnovna postulata: *sakrivanje podataka, enkapsulacija, nasljeđivanje i polimorfizam*. Objektno orijentirana sintaksa može nam *pomoći da razmišljamo na objektno orijentiran način*, ali sama za sebe ne čini objektno orijentiranu filozofiju. Program koji poštuje postulata objektno orijentirane filozofije neće postati neobjektno orijentiran ukoliko se s vremenom na vrijeme koristi sintaksa koja nije strogo objektno orijentirana. Što se tiče postulata objektno orijentirane filozofije, prijateljske funkcije ne narušavaju ni enkapsulaciju, jer se proglašenje funkcije prijateljskom također može smatrati *dijelom interfejsa klase* (tj. opisom šta se sa klasom može raditi) a ne nečim odvojenim od klase, dok ćemo kasnije vidjeti da prijateljske funkcije ne narušavaju ni preostala dva koncepta: nasljeđivanje i polimorfizam. Dakle, nema mesta tvrdnjama da su prijateljske funkcije ikakva smetnja objektno orijentiranom pristupu programiranju.

Na ovom mjestu treba ukazati na neke slučajeve u kojima se uglavnom svi slažu da upotreba funkcija članica dovodi do zabune, te da je u slučaju potrebe za takvim funkcijama bolje koristiti klasične funkcije. Razmotrimo, na primjer, funkcije članice "PomnoziSaSkalarom" i "Saberisa" klase "Vektor". Ove funkcije su klasične metode mutatori koje mijenjaju objekat na koji djeluju, i vraćaju kao rezultat referencu na izmijenjeni objekat (uglavnom sa ciljem da se podrže ulančani pozivi). Međutim, pretpostavimo da smo ove dvije metode izveli tako da *ne mijenjaju* vektor na koji djeluju, nego da samo *vraćaju kao rezultat* vektor koji je nastao množenjem vektora na koji djeluju sa skalarom, odnosno sabiranjem vektora na koji djeluju sa drugim vektorom. Na taj način, ove metode bi promijenile svoj karakter, odnosno umjesto mutatora postale bi inspektori. Njihove implementacije uz takvu modifikaciju moglo bi izgledati recimo ovako:

```
class Vektor {
    ... // Osim naredne dvije funkcije, sve ostaje isto...
    Vektor PomnoziSaSkalarom(double s) const {
        Vektor v;
        v.x = x * s; v.y = y * s; v.z = z * s;
        return v;
    }
    Vektor SaberiSa(const Vektor &v) const {
        Vektor v1;
        v1.x = x + v.x; v1.y = y + v.y; v1.z = z + v.z;
        return v1;
    }
};
```

Međutim, ovakve metode se ne smatraju dobrim, zbog toga što mogu biti zbumujuće. Na primjer, uz pretpostavku da imamo ovako definiranu klasu "Vektor" i dva primjerka "v1" i "v2" ove klase, pogledajmo sljedeću naredbu:

```
v1.SaberiSa(v2);
```

Šta radi ova naredba? Efektivno, ama baš ništa. Metoda "SaberiSa" ne utiče na vektor "v1" na koji je primijenjena, već samo vraća kao rezultat vektor koji je nastao sabiranjem vektora "v1" sa vektorom "v2". Međutim, vraćena vrijednost iz ove metode se ne koristi ni zašto (tj. ignorira se), tako da, na kraju, ova naredba ne ostavlja nikakav efekat. Ovako definirana metoda ima smisla jedino u slučaju da njen

rezultat na neki način *iskoristimo*, na primjer u dodjeli poput “`v1 = v1.SaberisA(v2)`“. Naravno, vraćeni rezultat smo mogli dodijeliti i nekom drugom vektoru, a ne ponovo vektoru “`v1`”. Ipak, ovakva metoda “`SaberisA`” je više zbumujuća nego korisna. Ranije napisana funkcija “`ZbirVektora`” je mnogo jasnija. Ista primjedba bi vrijedila za izmijenjenu verziju metode “`PomnoziSaSkalarom`”.

Razmotrimo još jedan primjer. Klasa “`Datum`” imala je metodu mutator “`Sljedeci`” koja *modificira* objekat na koji je primjenjena, tako da nakon modifikacije on sadrži sljedeći datum po kalendaru. Ukoliko bismo ovu metodu izmijenili tako da ne utiče na datum na koji je primjenjena nego da umjesto toga *vraća kao rezultat* sljedeći datum po kalendaru, ona bi postala zbumujuća zbog istog razloga kao u prethodnom primjeru. Ukoliko nam treba tako nešto, bolje je za tu svrhu upotrijebiti *običnu funkciju*, koju možemo deklarirati kao prijateljsku (sa ciljem dobijanja prava pristupa privatnim atributima klase). Takva funkcija mogla bi se implementirati recimo na sljedeći način:

```
const Datum Sljedeci(const Datum &d) {
    Datum d1 = d;
    d1.dan++;
    if(d1.dan > Datum::BrojDana(mjesec, godina)) {
        d1.dan = 1; d1.mjesec++;
    }
    if(d1.mjesec > 12) {
        d1.mjesec = 1; d1.godina++;
    }
    return d1;
}
```

Iz izloženih primjera možemo zaključiti sljedeće: *metode inspektori koje kao rezultat vraćaju objekat tipa klase u kojoj su definirane, mogu biti zbumujuće*. Ukoliko ipak želimo definirati i takve metode, zbrku možemo smanjiti ukoliko im pažljivo izaberemo ime. Na primjer, u prethodnom primjeru, konfuziju bismo smanjili ukoliko bismo metodu koja vraća sljedeći datum umjesto prosto “`Sljedeci`” nazvali “`VratisSljedeci`” ili “`DajSljedeci`”. Na taj način, samo ime metode sugerira da ona ne mijenja objekat na koji djeluje, već samo *vraća* novi objekat dobijen nakon obavljenе transformacije.

Ponekad je potrebno učiniti da se neka metoda neke druge klase učini prijateljem neke klase, tj. da joj se omogući pristup njenim privatnim atributima i metodama. To se može učiniti isto kao pri proglašenju obične funkcije prijateljskom, samo se ispred imena metode mora navesti i ime pripadne klase praćeno operatorom “`::`”. Ukoliko je potrebno da *sve metode* neke druge klase (npr. klase “`B`”) imaju pristup privatnim atributima i metodama neke klase (npr. klase “`A`”), tada je najbolje u deklaraciju klase “`A`” staviti deklaraciju poput

```
friend class B;
```

čime se čitava klasa “`B`” proglašava prijateljem klase “`A`” (preciznije, sve metode klase “`B`” postaju prijatelji klase “`A`”). Takvo “*prijateljstvo*” nije ni simetrično ni tranzitivno. Drugim riječima, prethodna deklaracija ne proglašava automatski klasu “`A`” prijateljem klase “`B`”. Također, ako je klasa “`B`” prijatelj klase “`A`”, a klasa “`C`” prijatelj klase “`B`”, to ne čini automatski klasu “`C`” prijateljem klase “`A`”.

Svrha ovog poglavlja bila je da uvede čitatelja odnosno čitateljku u osnovne smjernice koje vode ka objektno zasnovanom odnosno objektno orijentiranom pristupu programiranju. Većina novih programera pri prvom susretu sa ovim konceptima izražava sumnju u njihovu korist i postavlja pitanje da li je trud koji je potrebno uložiti u objektno orijentirano programiranje (s obzirom da ono prisiljava programera na veću disciplinu i razmišljanje unaprijed) isplativ. Da je odgovor na ovo pitanje itekako potvrđan, uvjerićemo se u narednim poglavljima, u kojima će se jasno vidjeti sve prednosti objektno zasnovanog i objektno

orijentiranog programiranja kroz razne tehnike koje bi bile jako teško ostvarljive držeći se samo filozofije proceduralnog programiranja.

31. Konstruktori i destruktori

U prethodnom poglavlju smo vidjeli da nam koncept skrivanja podataka i uvođenje funkcija članica za strogo kontrolirani pristup klase omogućava da spriječimo da se atributima nekog objekta dodijele takve vrijednosti koje bi sam objekat učinili besmislenim. Međutim, i dalje veliki problem može predstavljati činjenica da, poput svih ostalih promjenljivih, promjenljive tipa klase također imaju nedefiniran sadržaj sve dok im se eksplisitno ne dodijeli vrijednost. Ovaj problem se doduše može riješiti tako što se promjenljiva inicijalizira odmah pri deklaraciji (mada je dosta sporno kako uopće inicijalizirati objekat neke klase koja se sastoji od mnoštva atributa, pogotovo ukoliko su oni privatni), odnosno da joj se neposredno nakon inicijalizacije dodijeli vrijednost, ili pozove neka metoda (nazvana, recimo, “*Postavi*”) koja će promjenljivu postaviti u jasno definirano stanje. Međutim, niko nas *ne prisiljava* da inicijalizaciju, dodjelu ili poziv odgovarajuće metode zaista i moramo izvršiti. Stoga, u programiranju veoma često nastaju greške koje su posljedica neinicijaliziranih vrijednosti promjenljivih. Objektno zasnovano programiranje nudi izlaz iz ove situacije uvođenjem *konstruktora*, koji *primorava* korisnika da mora izvršiti inicijalizaciju objekata (i to na strogo kontroliran način) prilikom njihove deklaracije ili stvaranja dinamičkih objekata (pomoću operatora “**new**”). Konstruktori također omogućavaju da se alternativno izvrši automatska inicijalizacija atributa objekta na neke podrazumijevane vrijednosti, u slučaju da se inicijalizacija ne izvede eksplisitno.

Općenito rečeno, konstruktori definiraju *skupinu akcija koje se automatski izvršavaju nad objektom onog trenutka kada dođe do njegovog stvaranja* (tj. prilikom nailaska na njegovu deklaraciju, odnosno prilikom dinamičkog kreiranja objekta pomoću operatora “**new**”). Po formi, konstruktori podsjećaju na funkcije članice klase, samo što *uvijek imaju isto ime kao i klasa kojoj pripadaju i nemaju povratnog tipa*. Jedna klasa može imati više konstruktora, koji se tada moraju razlikovati po broju i/ili tipu parametara. Također, poput običnih funkcija, konstruktor može imati i *podrazumijevane parametre*.

Upotrebu konstruktora ćemo prvo ilustrirati na jednom jednostavnom primjeru. Prepostavimo da smo razvili klasu “Kompleksni” koja treba da predstavlja kompleksni broj. Značenje atributa i metoda prikazane klase jasan je sam po sebi, tako da nema potrebe da dajemo detaljnija objašnjenja. Pri tome smo predvidjeli i postojanje jedne prijateljske funkcije nazvane “*ZbirKompleksnih*”, koju ćemo implementirati kasnije. Ono na šta ovdje treba обратити pažnju su *tri konstruktora*, od kojih je jedan bez parametara, dok su druga dva sa jednim i dva parametra respektivno:

```
class Kompleksni {  
    double re, im;  
  
public:  
    Kompleksni() { re = im = 0; }  
    Kompleksni(double x) { re = x; im = 0; }
```

```

Kompleksni(double r, double i) { re = r; im = i; }

void Postavi(double r, double i) { re = r; im = i; }

void Ispisi() const { cout << "(" << re << "," << im << ","; }

double Realni() const { return re; }

double Imaginarni() const { return im; }

friend const Kompleksni ZbirKompleksnih(const Kompleksni &a,
                                         const Kompleksni &b);

};

```

Naravno, stvarna klasa za opis kompleksnih brojeva trebala bi imati znatno više metoda nego napisana klasa, ali na ovom mjestu želimo da se fokusiramo na *konstruktore*. Razmotrimo prvo konstruktor bez parametara. Konstruktori bez parametara se automatski izvršavaju nad objektom prilikom nailaska na deklaraciju odgovarajućeg objekta bez eksplicitno navedene inicijalizacije. Na primjer, ukoliko se izvrši sljedeća sekvenca naredbi:

```

Kompleksni a;
cout << a.Realni() << " " << a.Imaginarni();

```

na ekranu će se ispisati dvije nule, a ne neke nedefinirane vrijednosti. Prilikom deklaracije objekta “a” automatski je nad njim primijenjen konstruktor bez parametara koji je izvršio inicijalizaciju atributa “*re*” i “*im*” na nule (u skladu sa naredbom unutar tijela konstruktora). Također, prilikom stvaranja dinamičkog objekta pomoću operatora “**new**”, stvoreni objekat će automatski biti inicijaliziran konstruktorom bez parametara. Stoga će sljedeća sekvenca naredbi također ispisati dvije nule:

```

Kompleksni *pok;
pok = new Kompleksni;
cout << pok->Realni() << " " << pok->Imaginarni();

```

Konstruktori sa parametrima se izvršavaju ukoliko se prilikom deklaracije odgovarajućeg objekta (odnosno pri upotrebi operatora “**new**”) eksplicitno u zagradama navedu parametri koje treba proslijediti u konstruktore. Pri tome, ukoliko ne postoji konstruktor čiji se broj i tip parametara ne slaže sa navedenim parametrima, kompjuter prijavljuje grešku. Na primjer, ukoliko se izvrši sekvenca naredbi

```

Kompleksni a, b(3, 2), c(1), d, e(6, -1);
a.Ispisi(); b.Ispisi(); c.Ispisi(); d.Ispisi(); e.Ispisi();

```

na ekranu će redom biti ispisano “(0,0)”, “(3,2)”, “(1,0)”, “(0,0)” i “(6,-1)”. Primjer je dovoljno jasan, tako da detaljnija objašnjenja vjerovatno nisu potrebna. Sličan efekat mogli bismo postići i upotrebom operatora “**new**” i dinamičke alokacije objekata:

```

Kompleksni *pa = new Kompleksni, *pb = new Kompleksni(3, 2),
*pc = new Kompleksni(1), *pd = new Kompleksni,
*pe = new Kompleksni(6, -1);
pa->Ispisi(); pb->Ispisi(); pc->Ispisi(); pd->Ispisi(); pe->Ispisi();

```

Primijetimo da inicijalizacija objekata pomoću konstruktora sa jednim parametrom po sintaksi jako podsjeća na inicijalizaciju promjenljivih jednostavnih tipova (poput “**int**”, “**double**”, “**char**”, itd.) navođenjem početne vrijednosti u zagrati, kao npr. u sljedećem primjeru:

```

int a(5), b(3);
double c(12.245);
char d('A');

```

Već smo rekli da se takva sintaksa inicijalizacije naziva *konstruktorska sintaksa*, upravo zbog činjenice da daje utisak da jednostavni tipovi poput “**int**” također posjeduju konstruktore (bez obzira što oni nisu klase). Takvi prividni konstruktori nekada se nazivaju *pseudo-konstruktori*.

Upotrebom konstruktora sa *parametrima koji imaju podrazumijevane vrijednosti*, deklaracija klase “**Kompleksni**“ se može znatno skratiti. Na primjer, možemo pisati:

```

class Kompleksni {
    double re, im;
public:
    Kompleksni(double r = 0, double i = 0) { re = r; im = i; }
    void Postavi(double r, double i) { re = r; im = i; }
    void Ispisi() const { cout << "(" << re << "," << im << ", ";}
    double Realni() const { return re; }
    double Imaginarni() const { return im; }
    friend const Kompleksni ZbirKompleksnih(const Kompleksni &a,
                                                const Kompleksni &b);
};

```

Kako u ovom primjeru konstruktor klase “**Kompleksni**” ima dva parametra sa podrazumijevanim vrijednostima, on se ponaša kao konstruktor sa dva, jednim ili bez parametara, u zavisnosti koliko je zaista parametara navedeno prilikom deklaracije objekta, odnosno upotrebe operatora “**new**”. Naravno, ukoliko konstruktori rade konceptualno različite stvari za različit broj parametara, podrazumijevane parametre nije moguće koristiti. Također, ukoliko želimo da se konstruktor može pozvati sa dva parametra ili bez parametara, ali ne i sa

jednim parametrom, taj efekat ne možemo ostvariti pomoću parametara sa podrazumijevanim vrijednostima, nego moramo kreirati dva odvojena konstruktora (jedan sa dva parametra, i jedan bez parametara).

Ukoliko neka klasa ima konstruktoare, ali među njima nema *konstruktora bez parametara*, tada nije moguće kreirati instance te klase bez eksplicitnog navođenja parametara koji će se proslijediti konstruktoru. Slično, nije moguće kreirati dinamičke objekte te klase primjenom “**new**“ operatora bez navođenja neophodnih parametara. Pretpostavimo, na primjer, da smo napisali klasu nazvanu “**Datum**“ (radi jednostavnosti, prikazana klasa će biti mnogo siromašnija od istoimene klase razvijene u prethodnom poglavlju), koja ima samo konstruktor sa tri parametra:

```
class Datum {
    int dan, mjesec, godina;
public:
    Datum(int d, int m, int g) { Postavi(d, m, g); }
    void Postavi(int d, int m, int g);
    void Ispisi() const { cout << d << ". " << m << ". " << g; }
};
```

Pri tome, funkcija “**Postavi**“ će biti implementirana na isti način kao i u prethodnom poglavlju:

```
void Datum::Postavi(int d, int m, int g) {
    int broj_dana[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if(g % 4 == 0 && g % 100 != 0 || g % 400 == 0) broj_dana[1]++;
    if(g < 1 || d < 1 || m < 1 || m > 12 || d > broj_dana[m - 1])
        throw "Neispravan datum!\n";
    dan = d; mjesec = m; godina = g;
}
```

Kako konstruktor klase “**Datum**“ u suštini treba da uradi istu stvar kao i metoda “**Postavi**”, unutar tijela konstruktora samo smo prosto pozvali metodu “**Postavi**” sa istim parametrima (iz konstruktora se, kao iz ma koje funkcije članice, može pozvati bilo koja metoda iste klase bez navođenja objekta nad kojim se metoda primjenjuje). Interesantno je da ma koja metoda može pozvati drugu metodu iz iste klase čak i ako je ta metoda deklarirana *iza nje*, ili pristupiti atributu iz iste klase koji je deklariran *iza nje* (ovo je odstupanje od pravila da se smijemo obraćati samo identifikatorima koji su prethodno definirani, ili bar najavljeni). Naime, klase se prevode u *dva prolaza*. U prvom prolazu gledaju se samo deklaracije odnosno prototipovi metoda, dok se implementacije metoda razmatraju tek u drugom prolazu, kada je već poznato koji sve atributi i metode postoje u toj klasi..

Ovakvom definicijom klase “**Datum**“ onemogućena je prosta deklaracija tipa

```
Datum d;
```

Umjesto toga, neophodno je navesti parametre koji će se koristiti za inicijalizaciju objekta:

```
Datum d(7, 12, 2003);
```

Pri tome je bitno da parametri budu *smisleni*. Ukoliko zadamo besmislen datum, konstruktor će baciti izuzetak (zapravo, baciće ga funkcija “*Postavi*“ koja se poziva iz konstruktora), i objekat “*d*“ neće uopće biti stvoren.

Iz istih razloga, nije moguća ni dinamička dodjela poput

```
Datum *pok = new Datum;
```

ali je sljedeća dodjela posve korektna:

```
Datum *pok = new Datum(7, 12, 2003);
```

Vjerovatno će se neko zapitati zbog čega smo pored konstruktora definirali i metodu “*Postavi*“ koja radi istu stvar kao i konstruktor. Stvar je u sljedećem: konstruktor se poziva prilikom stvaranja objekta, *i samo tada*. Da nismo definirali metodu “*Postavi*”, ne bismo bili u mogućnosti da *naknadno* promijenimo sadržaj objekta (zapravo, uskoro ćemo vidjeti da bismo ipak bili u mogućnosti, ali po cijenu osjetnog gubitka na efikasnosti). Ovako, ukoliko kasnije želimo promijeniti datum pohranjen u (već kreiranom) objektu “*d*”, možemo prosto pisati:

```
d.Postavi(30, 4, 2004);
```

Važno je napomenuti da, osim u rijetkim izuzecima, konstruktor mora biti deklariran u *javnom dijelu klase*. Ukoliko bismo konstruktor stavili u privatni dio klase, onemogućili bismo njegovo pozivanje bilo odakle osim iz dijelova programa koji i inače imaju pristup privatnim dijelovima klase. Stoga bismo primjerke te klase mogli deklarirati samo unutar metoda iste klase (što je besmisleno), ili eventualno unutar neke od funkcija prijatelja klase ili metoda prijateljske klase (ovo ponekad može da ima smisla). Isto vrijedi i za stvaranje dinamičkih primjeraka te klase pomoću operatora “**new**”.

Treba naglasiti da nije preporučljivo deklarirati objekte koji imaju konstruktore *unutar tijela petlje*. Naime, lokalni objekti deklarirani unutar tijela petlje se stvaraju i uništavaju prilikom *svakog prolaska kroz tijelo petlje*. Bez obzira što će gotovo svaki kompjuter za C++ sigurno optimizirati stalno zauzimanje memorije na početku tijela petlje i oslobađanje memorije na kraju tijela petlje, tako da će zauzeti memoriju prije početka petlje a osloboditi je tek po

njenom završetku, konstruktor će se ipak sigurno pozivati *pri svakom prolasku kroz petlju*, što je veliki gubitak vremena, pogotovo ukoliko je konstruktor složen (ukoliko klasa ima i destruktorni, i on bi se pozivao pri svakom prolasku kroz petlju). Na primjer, ukoliko napišemo konstrukciju poput

```
for(int i = 1; i <= 1000; i++) {  
    Datum d(5, 12, 2002);  
    ...  
}
```

konstruktor klase “Datum” će se pozvati 1000 puta! Ovo možemo izbjegići ukoliko lokalnu promjenljivu “*a*” deklariramo kao *staticku* (ona se tada stvara samo pri prvom nailasku na njenu deklaraciju). Međutim, tako deklarirana promjenljiva zauzima memoriju sve do završetka programa. Bolje je promjenljivu “*d*” deklarirati *izvan petlje*, kao u sljedećoj konstrukciji:

```
Datum d(5, 12, 2002);  
for(int i = 1; i <= 1000; i++) {  
    ...  
}
```

Nedostatak ovog rješenja je što život promjenljive “*d*” tada neće biti ograničen samo na tijelo petlje. Ukoliko želimo ograničiti vrijeme život ove promjenljive, možemo upotrijebiti *još jedan blok*, kao u sljedećoj konstrukciji:

```
{  
    Datum d(5, 12, 2002);  
    for(int i = 1; i <= 1000; i++) {  
        ...  
    }  
}
```

Jedini slučaj u kojima je smisleno deklarirati objekat sa konstruktorom unutar tijela petlje je ukoliko neki od parametara koje prosljeđujemo konstruktoru zavise od neke promjenljive koja se mijenja u svakom prolazu kroz petlju (npr. promjenljiva “*i*” u prethodnim primjerima). Tada se, pri svakom prolazu kroz petlju, objekat inicijalizira na drugačiji način, pa je njegova deklaracija unutar tijela petlje svrsihodna.

U prethodnim poglavljima vidjeli smo da se strukture mogu inicijalizirati prilikom deklaracije navođenjem vrijednosti svih polja unutar vitičastih zagrada. Na primjer, ukoliko je tip “Datum”

definiran kao *struktura*, moguća je deklaracija sa inicijalizacijom poput sljedeće (primijetimo da je navedena inicijalizacija besmislena sa aspekta korektnosti datuma):

```
Datum d = {35, 14, 2004};
```

Međutim, klase se ne mogu ovako inicijalizirati. Za njihovu inicijalizaciju koriste se konstruktori. Preciznije, klasa se ne može inicijalizirati listom vrijednosti u vitičastim zgradama ukoliko sadrži *makar jedan privatni atribut* (a svaka prava klasa ih sadrži) i/ili ako klasa ima definirane konstruktore (što je također gotovo pravilo – dobro napisana klasa uvijek treba imati konstruktore). Razlozi za ovo su prilično jasni: inicijalizacija listom vrijednosti u vitičastim zgradama omogućila bi nekontroliran pristup privatnim atributima klase, a konstruktori svakako omogućavaju fleksibilniju inicijalizaciju. Za razliku od inicijalizacije listom vrijednosti koja vrši samo *prepisivanje* vrijednosti u odgovarajuće attribute, konstruktori mogu na sebe preuzeti dodatne akcije (poput provjere ispravnosti parametara) koje će garantirati ispravnu inicijalizaciju objekta, pa čak izvršiti i mnoštvo drugih akcija (vidjećemo uskoro da je kreiranje dinamički alociranih nizova najbolje prepustiti upravo konstruktorima).

Svaki konstruktor klase je moguće pozvati kao *običnu funkciju* (ne kao funkciju članicu, nego baš kao klasičnu funkciju, bez primjene na neki konkretan objekat). U tom slučaju, konstruktor se ponaša kao funkcija koja prvo *kreira neki bezimeni privremeni objekat* odgovarajuće klase, zatim ga *inicijalizira* na uobičajeni način, i na kraju *vraća kao rezultat tako kreiran i inicijaliziran objekat*. Dakle, konstruktor uvijek kao rezultat vraća objekat klase kojoj pripada. Stoga se naknadna promjena sadržaja objekta “d” tipa “Datum” može ostvariti i sljedećom konstrukcijom, bez upotrebe metode “Postavi”:

```
d = Datum(30, 4, 2004);
```

U ovom slučaju, poziv konstruktora sa desne strane kreiraće privremeni objekat tipa “Datum” koji će se inicijalizirati na navedene vrijednosti, a zatim će tako kreirani objekat biti iskopiran u objekat “d”. Naime, primjeri klase se također mogu dodjeljivati jedan drugom pomoću operatora “=” . Pri tome se, kao i kod struktura, svi atributi objekta sa desne strane prosto kopiraju u odgovarajuće attribute objekta sa lijeve strane, mada ćemo kasnije vidjeti da je, u slučaju potrebe, moguće definirati i drugačije ponašanje operatora dodjele “=”.

Nije loše naglasiti da se svi automatski kreirani privremeni objekti automatski uništavaju kada se ustanovi da više nisu potrebni, tako da se ne moramo plašiti da će njihovo stvaranje dovesti do curenja memorije. Drugim riječima, efekat prethodne dodjele je sličan kao da smo napisali:

```
{  
    Datum privremeni(30, 4, 2004);  
    d = privremeni;
```

```
}
```

Mada ovaj primjer nedvosmisleno ilustrira da smo mogli proći i bez metode “*Postavi*”, njena primjena je znatno efikasnija, jer nema potrebe za stvaranjem privremenog objekta i njegovim kopiranjem u objekat koji želimo postaviti.

Da bismo bolje shvatili šta se dešava, vratimo se na klasu “*Kompleksni*” koju smo ranije napisali. Neka je potrebno deklarirati promjenljivu “*a*” tipa “*Kompleksni*” i inicijalizirati je na kompleksni broj “*(1, 2)*”. Najneposredniji način da ovo uradimo je slijedeći:

```
Kompleksni a (1, 2);
```

Međutim, postoje još dva načina da postignemo isti efekat. Naime, moguće je izvršiti inicijalizaciju nekog objekta i pomoću znaka “*=*”, ali isključivo *objektom istog tipa* (pri tome se nad objektom sa lijeve strane znaka “*=*” ne izvršava konstruktor). Na primjer, sasvim su moguće sljedeće konstrukcije:

```
Kompleksni a (1, 2);
```

```
Kompleksni b = a;
```

Posljednju deklaraciju možemo ravnopravno pisati i na sljedeći način:

```
Kompleksni b (a);
```

Stoga je drugi način da izvršimo inicijalizaciju objekta “*a*” sljedeći (ukoliko čitatelju ili čitateljki sintaksa koja se ovdje koristi djeluje odnekud poznata, to je zbog toga što tip “complex” koji smo koristili ranije nije ništa drugo nego klasa definirana u istoimenoj biblioteci):

```
Kompleksni a = Kompleksni(1, 2);
```

Pored toga, moguća je i sljedeća sekvenca, u kojoj se koristi *naknadna dodjela* (ali samo zahvaljujući činjenici da klasa “*Kompleksni*” posjeduje i konstruktor bez parametara, dok bi prethodna naredba bila moguća i bez te pretpostavke):

```
Kompleksni a;  
a = Kompleksni(1, 2);
```

Mada sve tri navedene konstrukcije proizvode isti efekat, između njih je velika razlika u pogledu efikasnosti. Prva konstrukcija *kreira objekat “a” i inicijalizira ga na kompleksni broj*

“(1, 2)”. Druga konstrukcija kreira privremeni objekat koji inicijalizira na kompleksni broj “(1, 2)” a zatim ga kopira u objekat “a”. Treća konstrukcija kreira objekat “a” i inicijalizira ga na kompleksni broj “(0, 0)” posredstvom konstruktora bez parametara, a zatim kreira privremeni objekat, inicijalizira ga na kompleksni broj “(1, 2)” i na kraju kopira u objekat “a”. Očigledno je prva konstrukcija ubjedljivo najefikasnija, dok je posljednja konstrukcija ubjedljivo najneefikasnija.

Treba naglasiti da kompjajler ima *pravo* (ali ne i *obavezu*) da izvrši optimizaciju izvjesnih neefikasnih konstrukcija koje je programer upotrijebio. Tako će većina današnjih kompjajlera u mnogim slučajevima izvršiti optimizaciju druge konstrukcije tako da generira isti izvršni kôd kao i prva konstrukcija. Međutim, kompjajler to može izvršiti samo u slučajevima kada je sigurno da obje konstrukcije zaista na kraju dovode do istog cilja. Na primjer, uskoro ćemo vidjeti da projektant klase može *zabraniti* mogućnost kopiranja primjeraka neke klase iz jednog u drugi. U tom slučaju, kompjajler ne može optimizirati drugu konstrukciju, s obzirom da ona uopće neće biti dozvoljena! Također, uskoro ćemo vidjeti da se definiranjem tzv. *konstruktora kopije* može specificirati kako će se tačno obavljati kopiranje jednog primjerka neke klase u drugi. U tom slučaju, kompjajler *ne smije* optimizirati drugu konstrukciju, s obzirom da, uz prisustvo konstruktora kopije, smisao ove konstrukcije (u kojoj se *koristi kopiranje*) ne mora biti isti kao i smisao prve konstrukcije (u kojoj se *ne koristi kopiranje*).

Konstruktor bez parametara se također može pozvati kao funkcija (uz navođenje para praznih zagrada). Na primjer, ukoliko želimo ranije deklariranoj promjenljivoj “a” dodijeliti kompleksni broj “(0, 0)”, možemo to uratiti i ovako:

```
a = Kompleksni();
```

Ovo je u skladu sa sintaksnim konstrukcijama poput “`int()`” itd. sa kojima smo se ranije susreli. Međutim, treba naglasiti da se par praznih zagrada ne smije staviti ukoliko želimo samo deklarirati promjenljivu koja se inicijalizira konstruktorom bez parametara. Tako, ukoliko želimo samo deklarirati promjenljivu “a” koja se inicijalizira konstruktorom bez parametara, to ne smijemo uraditi ovako:

```
Kompleksni a();
```

Par zagrada ovdje je suvišan. Međutim, nezgodno je što je ovakva pogrešna deklaracija također *sintaksno ispravna*. Naime, lako možemo vidjeti da ona zapravo predstavlja *prototip funkcije “a” bez parametara, i koja vraća objekat tipa “Kompleksni” kao rezultat!*

Činjenica da se konstruktor klase pored toga što se automatski poziva prilikom stvaranja objekta može pozvati i kao obična funkcija ima mnogobrojne primjene. Na primjer, razmotrimo kako bismo napisali funkciju “zbirKompleksnih” koju smo ostali dužni da implementiramo. Jedno očigledno rješenje je sljedeće:

```

const Kompleksni ZbirKompleksnih(const Kompleksni &a,
const Kompleksni &b) {
    Kompleksni c;
    c.Re = a.Re + b.Re; c.Im = a.Im + b.Im;
    return c;
}

```

U ovom primjeru se posve nepotrebno poziva konstruktor bez parametara za objekat “c”, s obzirom da odmah nakon toga ručno mijenjamo njegove attribute. Bolje bi bilo odmah primijeniti konstruktor sa dva parametra da inicijalizira objekat “c” na neophodne vrijednosti:

```

const Kompleksni ZbirKompleksnih(const Kompleksni &a,
const Kompleksni &b) {
    Kompleksni c(a.Re + b.Re, a.Im + b.Im);
    return c;
}

```

Međutim, ukoliko uočimo da se konstruktor može pozvati kao *funkcija*, zaključićemo da nam uopće nije potreban lokalni objekat “c”, nego možemo iskoristiti i privremeni objekat koji kreira konstruktor kada se upotrijebi kao funkcija:

```

const Kompleksni ZbirKompleksnih(const Kompleksni &a,
const Kompleksni &b) {
    return Kompleksni(a.Re + b.Re, a.Im + b.Im);
}

```

U ovom slučaju konstruktor kreira privremeni objekat koji inicijaliziramo na željene vrijednosti i vraćamo ga kao rezultat iz funkcije. Ovakvo rješenje ne samo da je jednostavnije, nego je i efikasnije od prethodnog rješenja. U prvom slučaju, objekat “c” prestaje postojati nakon završetka funkcije, pa je neophodno njegovo kopiranje u privremeni objekat koji prihvata vraćenu vrijednost iz funkcije (šta će se sa njim dalje desiti, zavisi od toga u kakvom je kontekstu funkcija pozvana). Međutim, u drugom slučaju, poziv konstruktora kao funkcije odmah kreira privremeni objekat koji ujedno predstavlja vraćenu vrijednost iz funkcije, tako da ne dolazi do nepotrebognog kopiranja.

Ovim ni izdaleka nisu iscrpljene sve mogućnosti primjene pozivanja konstruktora kao funkcija. Da bismo uvidjeli još neke primjene, razmotrimo prvo jedan primjer kako se napisane funkcije mogu primijeniti. Primjer je jasan sam po sebi, i ispisuje “(8,11)”:

```

Kompleksni a(3, 12), b(5, -1);
Kompleksni c = ZbirKompleksnih(a, b);
c.Ispisi();

```

Međutim, činjenica da se konstruktori mogu pozvati kao funkcije omogućava i ovakve konstrukcije:

```
Kompleksni c = ZbirKompleksnih(Kompleksni(3, 12), Kompleksni(5, -1));  
c.Ispisi();
```

Ova konstrukcija je efektivno ekvivalentna sljedećoj konstrukciji:

```
{  
    Kompleksni privremeni_1(3, 12), privremeni_2(5, -1);  
    Kompleksni c = ZbirKompleksnih(privremeni_1, privremeni_2);  
}  
c.Ispisi();
```

Na osnovu onoga što smo vidjeli u prethodnom poglavlju o mehanizmu vraćanja objekata kao rezultata iz funkcija, jasno je da su moguće i konstrukcije poput sljedeće:

```
ZbirKompleksnih(Kompleksni(3, 12), Kompleksni(5, -1)).Ispisi();
```

Ipak, bitno je naglasiti da je direktni prenos objekta koji je vraćen kao rezultat iz konstruktora u neku drugu funkciju moguć samo u slučaju da se odgovarajući parametar prenosi *po vrijednosti*, ili *po referenci na konstantni objekat* (kao što jeste za slučaj funkcije "ZbirKompleksnih"). Ukoliko bi odgovarajući formalni parametar bio *obična referencia* (na nekonstantni objekat) tada bi odgovarajući stvarni parametar morao biti *promjenljiva* ili barem *l-vrijednost* (npr. element niza ili dereferencirani pokazivač). Reference na nekonstantne objekte ne mogu se vezati na privremene objekte!

Sve što je do sada rečeno, čista je posljedica činjenice da se konstruktori mogu pozivati kao funkcije, i potpuno je u skladu sa očekivanjima. Međutim, *posve neočekivano*, rade i sljedeće konstrukcije:

```
Kompleksni a(3, 12), b = 2, c, d;  
c = ZbirKompleksnih(a, 4);  
d = 7;
```

U ovom primjeru, objektima tipa "Kompleksni" *dodjeljujemo realne brojeve, inicijaliziramo ih realnim brojevima, i šaljemo realne brojeve u funkciju koja očekuje parametre tipa "Kompleksni"*! Pri tome, ukoliko ispišemo sadržaje ovih promjenljivih, vidjećemo da su se u ovom primjeru realni brojevi ponašali kao elementi klase "Kompleksni" sa imaginarnim dijelom jednakim nuli. Neko bi naivno mogao pomisliti da je ovo posve prirodno, s obzirom da se realni brojevi zaista mogu shvatiti kao specijalan slučaj kompleksnih brojeva. Međutim, pravo pitanje je *kako kompjajler ovo može znati*, s obzirom da "Kompleksni" nije ugrađeni tip podataka, već klasa koju smo mi napisali!? Tim prije što sličan pokušaj da objektu tipa "Datum" pokušamo dodijeliti broj neće uspeti...

Odgovor na ovu misteriju leži u *konstruktoru sa jednim parametrom* koji smo definirali unutar klase "Kompleksni". Naime, konstruktori sa jednim parametrom imaju, pored uobičajenog značenja, i jedno specijalno značenje. Kada god pokušamo objektu neke klase dodijeliti nešto što *nije objekat te klase*, kompjajler neće odmah prijaviti grešku, nego će provjeriti da li možda ta klasa sadrži konstruktor sa jednim parametrom koji prihvata kao parametar tip onoga što pokušavamo dodijeliti objektu. Ukoliko takav konstruktor ne postoji, biće prijavljena greška. Međutim, ukoliko takav konstruktor postoji, on će biti iskorišten da stvori privremeni objekat tipa te klase (kao da smo konstruktor sa jednim parametrom pozvali kao funkciju), uzimajući

ono što pokušavamo dodijeliti objektu kao parametar, te će nakon toga stvoreni objekat biti iskorišten umjesto onoga što pokušavamo dodijeliti objektu. Drugim riječima, dodjela “`d = 7`” se zapravo interpretira kao

```
d = Kompleksni(7);
```

Sličan proces se odvija i prilikom prenošenja parametara u funkcije, tako da se poziv

```
c = ZbirKompleksnih(a, 4);
```

zapravo interpretira kao

```
c = ZbirKompleksnih(a, Kompleksni(4));
```

Generalno, kad god se negdje kompjajler očekuje da zatekne primjerak neke klase pojavi neki drugi tip podataka (uključujući eventualno i primjerak neke druge klase), kompjajler će prije nego što prijavi grešku pogledati da li ta klasa posjeduje konstruktor sa jednim parametrom koji prihvata kao parametar upravo taj tip podataka. Ukoliko takav konstruktor postoji, on će automatski biti iskorišten za konstrukciju privremenog objekta koji će biti upotrijebљen umjesto onoga čiji tip nije bio odgovarajući. Tek ukoliko takav konstruktor *ne postoji*, biva prijavljena greška. Pojednostavljenio rečeno, možemo reći da se konstruktor neke klase sa jednim parametrom automatski koristi za *pretvorbu tipa* iz tipa koji odgovara parametru konstruktora u tip koji odgovara klasi. Dakle, konstruktor sa jednim parametrom se *implicitno poziva* da izvrši pretvorbu tipa kada god se na mjestu gdje je očekivan objekat tipa neke klase nađe nešto drugo čiji tip odgovara tipu parametra nekog od konstruktora sa jednim parametrom. Na primjer, ukoliko neka funkcija vraća rezultat tipa “`Kompleksni`”, pri pokušaju da naredbom “`return`” vratimo iz funkcije realan broj biće automatski pozvan konstruktor sa jednim parametrom da izvrši neophodnu pretvorbu (kao da smo npr. umjesto “`return x`” napisali “`return Kompleksni(x)`”).

Implicitno korištenje konstruktora sa jednim parametrom za automatsku konverziju tipova je veoma korisna osobina. Međutim, kao što ćemo vidjeti, postoje izvjesne situacije u kojima ovo implicitno (nevidljivo) pozivanje konstruktora sa jednim parametrom može da *zbunjuje*, pa čak i da *smeta*. Ukoliko iz bilo kojeg razloga želimo da zabranimo implicitno korištenje konstruktora sa jednim parametrom za automatsku pretvorbu tipova, tada ispred deklaracije odgovarajućeg konstruktora treba da navedemo ključnu riječ “`explicit`”. U tom slučaju, takav konstruktor se nikad neće implicitno pozivati radi pretvorbe tipova, već samo u slučaju kada ga eksplicitno pozovemo, bilo kao klasičnu funkciju, bilo prilikom deklaracije promjenljivih sa konstruktorskim sintaksom (tj. sa inicijalizatorom unutar zagrade), bilo pri upotrebi operatora “`new`”. Takav konstruktor nazivamo *eksplicitni konstruktor*. Naglasimo da ključna riječ “`explicit`” ima smisla samo ispred deklaracije konstruktora sa jednim parametrom, ili eventualno konstruktora sa više parametara koji imaju podrazumijevane

vrijednosti, tako da se on može pozvati i kao konstruktor sa jednim parametrom (npr. konstruktora sa tri parametra u kojem dva ili sva tri parametra imaju podrazumijevane vrijednosti).

Radi boljeg razumijevanja izloženih koncepata, rezimirajmo ukratko u čemu je razlika između sljedeće tri konstrukcije:

```
Kompleksni a = 5;  
Kompleksni a(5);  
Kompleksni a = Kompleksni(5);
```

Prva i druga konstrukcija su gotovo potpuno ekvivalentne. I u jednom i u drugom slučaju koristi se konstruktor sa jednim parametrom za inicijalizaciju objekta "a". Zapravo, jedina razlika je u tome što prva konstrukcija *neće biti dozvoljena* u slučaju da je konstruktor klase "Kompleksni" (sa jednim parametrom) deklariran kao *eksplisitni konstruktor*. U trećoj konstrukciji, prvo se poziva konstruktor sa jednim parametrom za konstrukciju privremenog objekta, koji se zatim kopira u objekat "a". Efektivno je ova konstrukcija ekvivalentna sa prve dvije (ukoliko smisao kopiranja nije izmijenjen), ali je manje efikasna od nje. Kompajler ima pravo ali ne i obavezu da (ukoliko smije) ovu konstrukciju optimizira da postane ekvivalentna sa prve dvije.

Konstruktor sa jednim parametrom se također koristi i u slučajevima kada koristimo operator za konverziju tipa. Na primjer, izraz "(Kompleksni)x" u kojem se vrši pretvorba tipa objekta "x" u tip "Kompleksni" interpretira se upravo kao izraz "Kompleksni(x)". Ovo je potpuno u skladu sa već uobičajenim tretmanom po kojem su izrazi poput "(int)x" i "int(x)" ekvivalentni. Međutim, treba naglasiti da je izraz "(Kompleksni)x" legalan samo ukoliko konstruktor klase "Kompleksni" nije eksplisitni konstruktor, dok je izraz "Kompleksni(x)" legalan u svakom slučaju.

Nije potrebno posebno napominjati da je moguće napraviti *nizove čiji su elementi primjeri neke klase*, potpuno analogno nizovima čiji su elementi struktornog tipa (razumije se da ćemo u takvim nizovima metode klase primjenjivati nad individualnim elementima niza, a ne nad čitavim nizom). Međutim, prisustvo konstruktora donekle komplicira deklariranje nizova čiji su elementi primjeri klase. Radi jednostavnosti, takve nizove ponekad ćemo zvati prosto *nizovi klase*, iako je jasno da se od klase (koje su *tipovi* a ne *objekti*) ne mogu praviti nizovi (nizove možemo praviti samo od *primjeraka klase*, koji su objekti). Nizove klase moguće je bez problema deklarirati jedino ukoliko klasa *uopće nema konstruktore* (što je vrlo loša praksa), ili ukoliko ima *konstruktor bez parametara*. U tom slučaju, konstruktor bez parametara će biti iskorišten da inicijalizira *svaki element niza*. Na primjer, deklaracija

```
Kompleksni niz[100];
```

deklariraće niz “niz“ od 100 elemenata tipa “Kompleksni”, od koji će svaki (zahvaljujući konstruktoru bez parametara) biti inicijaliziran na kompleksni broj “ $(0,0)$ ”. Potpuno ista stvar bi se desila i u slučaju kreiranja dinamičkog niza čiji su elementi tipa “Kompleksni“ deklaracijom poput

```
Kompleksni *dinamicki_niz = new Kompleksni[100];
```

Naime, i u tom slučaju, svi elementi stvorenog dinamičkog niza bili bi inicijalizirani konstruktorom bez parametara.

Situacija je znatno složenija u slučaju da klasa *ima konstruktore, ali nema konstruktor bez parametara*. U tom slučaju, neposredne deklaracije nizova čiji su elementi primjeri klase (kako statičkih, tako i dinamičkih) *nisu moguće*. Na primjer, ni jedna od sljedeće dvije deklaracije neće biti prihvaćena:

```
Datum niz_datuma[100];
Datum *dinamicki_niz_datuma = new Datum[100];
```

Razlog nije teško naslutiti: ni u jednoj od ove dvije deklaracije nije jasno kako bi se trebali inicijalizirati elementi kreiranih nizova. U ovakvoj situaciji, moguće je deklarirati jedino *statički inicijalizirani niz klase*, pri čemu se inicijalizacija elemenata niza vrši kao i kod male kojeg drugog niza (navođenjem željenih elemenata niza unutar vitičastih zagradama). Pri tome se moraju navesti *svi elementi niza*. Međutim, kako su u ovom slučaju elementi niza *primjeri klase*, to i elementi u vitičastim zagradama moraju biti također primjeri iste klase. Slijedi da oni moraju biti ili promjenljive iste klase, ili rezultati primjene konstruktora pozvanog kao funkcija, ili neki izrazi koji su tipa te klase (npr. pozivi neke funkcije koja vraća objekat te klase kao rezultat). Na primjer, sljedeća konstrukcija se može upotrijebiti za kreiranje inicijaliziranog niza koji sadrži 5 datuma:

```
Datum niz_datuma[5] = {Datum(31, 12, 2004), Datum(8, 4, 2003),
Datum(14, 7, 1998), Datum(4, 11, 2000), Datum(6, 2, 2005)};
```

U slučaju kada sve elemente želimo inicijalizirati na isti datum, situacija je donekle jednostavnija, s obzirom da možemo uraditi nešto kao u sljedećoj konstrukciji:

```
Datum d(1, 1, 2000);
Datum niz_datuma[10] = {d, d, d, d, d, d, d, d, d, d};
```

Međutim, jasno je da je zbog potrebe da navedemo inicijalizaciju za *svaki* element niza na ovaj način praktično nemoguće definirati iole veći niz. Pored toga, potpuno nam je

onemogućeno kreiranje dinamičkih nizova. Postoje dva izlaza iz ove situacije. Jedno rješenje je ubaciti u definiciju klase konstruktor bez parametara koji inicijalizira atribute klase na neku podrazumijevanu vrijednost. Ovo rješenje je jednostavno, ali se ne smatra baš dobrom, s obzirom da ne postoji uvijek smislene podrazumijevane vrijednosti koje bi trebalo da postavlja konstruktor bez parametara. Na primjer, za klasu "Kompleksni" je prirodno izvršiti inicijalizaciju na kompleksni broj "(0, 0)" u slučaju da se drugačije ne navede, međutim nije posve jasno na koje vrijednosti bi trebalo inicijalizirati elemente klase "Datum" u slučaju da se eksplicitno ne navedu dan, mjesec i godina. Stoga je bolje rješenje umjesto nizova klasa koristiti *nizove pokazivača na klase* (odnosno, nizove čiji su elementi pokazivači na dinamički kreirane primjerke klase). Ovo rješenje biće demonstrirano malo kasnije.

Statički inicijalizirani niz čiji su elementi primjeri neke klase moguće je napraviti i u slučaju kada klasa posjeduje kako konstruktore sa parametrima, tako i konstruktore bez parametara. U tom slučaju, inicijalizaciona lista ne mora sadržavati sve elemente, jer će preostali elementi biti automatski inicijalizirani konstruktorom bez parametara. Na primjer, moguće je deklarirati sljedeći niz:

```
Kompleksni niz[10] = {Kompleksni(4, 2), Kompleksni(3), 5};
```

U ovom primjeru element "niz[0]" biće inicijaliziran na kompleksni broj "(4, 2)", element "niz[1]" na kompleksni broj "(3, 0)", element "niz[2]" na kompleksni broj "(5, 0)", a svi ostali elementi niza na kompleksni broj "(0, 0)". Primijetimo da je inicijalizacija elementa "niz[2]" na kompleksni broj "(5, 0)" navođenjem samo broja "5" moguća zahvaljujući pretvorbi tipova koja se ostvaruje pomoću konstruktora sa jednim parametrom. Naravno, ovakva inicijalizacija ne bi bila moguća da je konstruktor klase "Kompleksni" definiran kao eksplicitni konstruktor pomoću ključne riječi "**explicit**".

Opisani problem sa konstruktorima i nizovima klasa u praksi ne predstavlja osobit problem, s obzirom da se u objektno orijentiranom programiranju kao elementi nizova mnogo češće koriste *pokazivači na klase* nego sami *primjeri klase*. Jedan od razloga za to je što često ne znamo unaprijed na koje vrijednosti treba inicijalizirati elemente klase, tako da pojedine primjerke klase često kreiramo *dinamički*, i to tek onda kada saznamo čime ih treba inicijalizirati. Pored toga, kasnije ćemo vidjeti da su nizovi pokazivača na klase također potrebni da se podrži još jedan od bitnih koncepata objektno orijentiranog programiranja, tzv. *polimorfizam*, tako da je bolje da se što ranije naviknemo na njihovu upotrebu. Prepostavimo, na primjer, da želimo unijeti podatke o deset datuma sa tastature i smjestiti ih u niz.

Očigledno, ne možemo znati na koje vrijednosti treba inicijalizirati neki datum prije nego što podatke o njemu unesemo sa tastature. Ukoliko bismo deklarirali *običan niz datuma*, on bi zbog postojanja konstruktora morao na početku biti inicijaliziran nečim, bilo ručnim navođenjem inicijalizacije za svaki element, bilo dodavanjem konstruktora bez parametara u klasu "Datum" koja bi izvršila neku podrazumijevanu inicijalizaciju. Međutim, u oba slučaja radimo bespotrebnu inicijalizaciju objekata kojima ćemo svakako promijeniti sadržaj čim se podaci o konkretnom datumu unesu sa tastature. Stoga je mnogo bolje deklarirati *niz čiji su elementi pokazivači na tip "Datum"*, a same datume kreirati dinamički onog trenutka kada

podaci o njemu budu poznati. Nakon toga ćemo pokazivač na novostvoreni objekat, smjestiti u niz, a samom objektu ćemo indirektno pristupati preko pokazivača. Ilustrirajmo ovo na konkretnom primjeru. Deklariraćemo niz pokazivača na objekte tipa “`Datum`”, koji ćemo nazvati “`niz_datuma`” (iako se, striktno rečeno, radi o nizu pokazivača na datume):

```
Datum *niz_datuma[10];
```

Bitno je istaći da nizove pokazivača na primjerke neke klase uvijek možemo deklarirati bez problema, bez obzira kakve konstruktore klase sadrži (s obzirom da konstruktori ne inicijaliziraju pokazivače, nego primjerke klase, tako da će inicijalizacija biti odgođena za trenutak kada dinamički kreiramo objekat pomoću operatora “`new`”). Dalje sa ovakvim nizom pokazivača možemo raditi kao i da se radi o običnom nizu klasa, samo ćemo za poziv metoda umjesto operatora “`.`” koristiti operator “`->`”. Na primjer, za ispis trećeg elementa niza ćemo umjesto konstrukcije “`niz_datuma[3].Ispisi()`” koristiti konstrukciju “`niz_datuma[3] ->Ispisi()`” (podsjetimo se da je ova konstrukcija zapravo ekvivalentna konstrukciji “`(*niz_datuma[3]).Ispisi()`”). Prema tome, činjenica da radimo sa pokazivačima na primjerke klase umjesto sa samim primjercima klase gotovo da ne unosi nikakvu dodatnu poteškoću. Treba samo obratiti pažnju na dva sitna detalja. Prvo, operator “`new`” može baciti izuzetak u slučaju da ponestane memorije (mada je veoma mala šansa da će se ovo desiti, s obzirom da jedan datum zauzima veoma malo memorije), tako da treba predvidjeti hvatanje izuzetka. Drugo, sve dinamički stvorene objekte treba na kraju i uništiti. Sve ovo je demonstrirano u sljedećem programu, koji traži da se unese 10 datuma sa tastature, i koji nakon toga ispisuje sve unesene datume:

```
Datum *niz_datuma[10] = {};
cout << "Unesi 10 datuma:\n";
try {
    for(int i = 0; i < 10; i++) {
        int dan, mjesec, godina;
        cout << "Datum " << i + 1 << ":" << endl;
        cout << " Dan: "; cin >> dan;
        cout << " Mjesec: "; cin >> mjesec;
        cout << " Godina: "; cin >> godina;
        try {
            niz_datuma[i] = new Datum(dan, mjesec, godina);
        }
        catch(const char greska[]) {           // Konstruktor klase "Datum"
            cout << greska << endl;           // može baciti izuzetak...
            i--;
        }
    }
    cout << "Unijeli ste datume:\n";
    for(int i = 0; i < 9; i++) {
        niz_datuma[i]->Ispisi();
        cout << endl;
    }
}
catch(...) {
    cout << "Problemi sa memorijom!\n";
```

```

    }
    for(int i = 0; i < 9; i++) delete niz_datuma[i]; // Obriši zauzeto...
}

```

Primijetimo da smo na početku sve elemente niza pokazivača inicijalizirali na nule (navođenjem praznih vitičastih zagrada prilikom deklaracije), sa ciljem da izbjegnemo eventualne probleme koje bi na kraju mogao izazvati operator “**delete**“ u slučaju da neka od alokacija slučajno nije uspjela.

Kao što elementi struktura mogu biti ponovo strukture, tako i elementi klase mogu biti (i često trebaju biti) također primjeri neke klase. Na primjer, neka hipotetička klasa “Student“ vjerovatno bi trebala sadržavati atribut “*datum_rodjenja*“ koji je tipa “*Datum*“. Pogledajmo kako bi mogla izgledati deklaracija neke minimalističke klase “Student”:

```

class Student {
    char ime_i_prezime[50];
    int indeks;
    Datum datum_rodjenja;
public:
    // Ovdje bi trebalo definirati interfejs klase
};

```

Međutim, u ovakvim situacijama se ponovo javlja jedna poteškoća uzrokovana konstruktorima (nemojte pomisliti da konstruktori izazivaju samo probleme – konstruktori su jedna od najkorisnijih alatki objektno zasnovanog programiranja, a sve eventualne poteškoće koje nastaju uslijed njihovog korištenja veoma se lako rješavaju, samo treba objasniti kako). Naime, pošto klasa “*Datum*“ ima konstruktor, ni jedan objekat tipa “*Datum*“ ne može ostati neinicijaliziran. Stoga će svako stvaranje objekta tipa “*Student*“ obavezno dovesti i do inicijalizacije njegovog atributa “*datum_rodjenja*“. Jedino je pitanje *kako*. Da klasa “*Datum*“ ima konstruktor bez parametara, on bi bio automatski iskorišten za inicijalizaciju atributa “*datum_rodjenja*“. Međutim, kako klasa “*Datum*“ ima samo konstruktor sa tri parametra, objekti klase “*Datum*“ *moraju biti inicijalizirani sa tri parametra*. Isto vrijedi i za atribute klase: atribut “*datum_rodjenja*“ mora također biti inicijaliziran sa tri parametra, odmah pri stvaranju nekog objekta tipa “*Student*“. Kako je konstruktor jedino mjesto odakle je moguće izvršiti inicijalizaciju objekta odmah po njegovom stvaranju, slijedi da klasa “*Student*“ svakako mora imati konstruktor (što i nije neki problem, jer bi svaka dobro napisana klasa trebala imati konstruktor). Stoga bi konstruktor klase “*Student*“ morao *nekako* da inicijalizira atribut “*datum_rodjenja*“. Dalje, kada bi se inicijalizacija ovog atributa izvela negdje unutar tijela konstruktora, on bi unutar tijela konstruktora bio privremeno neinicijaliziran, sve do mjesta gdje je izvršena inicijalizacija, što se također ne smije dozvoliti. Slijedi da atribut “*datum_rodjenja*“ mora na neki način biti inicijaliziran *prije nego što se tijelo konstruktora uopće počne izvršavati!*

Da bi se riješio ovaj problem, uvedena je sintaksa koja omogućava konstruktoru neke klase da

pozove konstruktore drugih klasa. Za tu svrhu, nakon zatvorene zagrada u deklaraciji parametara konstruktora stavlja se *dvotačka*, iza koje slijedi takozvana *konstruktorska inicijalizacijska lista*. Ona sadrži popis svih inicijalizacija atributa koje se moraju izvršiti putem konstruktora odmah pri stvaranju objekta. Pri tome se u konstruktorskoj inicijalizacijskoj listi koristi ista sintaksa kao pri inicijalizaciji običnih promjenljivih pomoću konstruktora. Tek nakon konstruktorske inicijalizacijske liste slijedi tijelo konstruktora. Demonstrirajmo ovo na konkretnom primjeru. Uvedimo u klasu student konstruktor sa pet parametara koji redom predstavljaju ime (sa prezimenom), broj indeksa, dan, mjesec i godinu rođenja studenta. Ovaj konstruktor će prvo u konstruktorskoj inicijalizacionoj listi inicijalizirati atribut “`datum_rodjenja`“ u skladu sa parametrima proslijeđenim u konstruktor klase “`Student`”, a zatim će unutar tijela konstruktora inicijalizirati preostala dva atributa “`ime_i_prezime`“ i “`indeks`“:

```
class Student {
    char ime_i_prezime[50];
    int indeks;
    Datum datum_rodjenja;

public:
    Student(const char ime[], int indeks, int dan, int mjesec, int godina) :
        DatumRodjenja(dan, mjesec, godina) {
        strcpy(ime_i_prezime, ime); Student::indeks = indeks;
    }
    // Ovdje bi trebalo definirati ostatak interfejsa klase
};
```

Neko bi se mogao zapitati zašto su se morale uvoditi konstruktorske inicijalizacione liste, odnosno zašto ne bi bilo moguće dopustiti da se pri deklaraciji atributa “`datum_rodjenja`“ odmah napiše nešto poput

```
Datum datum_rodjenja(14, 5, 1978);
```

kao kod deklaracija običnih promjenljivih tipa “`Datum`”. Razlog za to je sljedeći: da je omogućeno takvo pojednostavljenje, svi objekti tipa “`Student`“ bi uvijek imali atribut “`datum_rodjenja`“ inicijaliziran na istu vrijednost. Ovako, konstruktorske inicijalizacione liste omogućavaju da konstruktor objekta “`Student`“ definira kako će biti inicijaliziran atribut “`datum_rodjenja`“.

Konstruktorske inicijalizacione liste moraju se koristiti za inicijalizaciju onih atributa koji moraju biti propisno inicijalizirani. Međutim, one se mogu koristiti i za inicijalizaciju ostalih atributa, slično kao što se i inicijalizacija promjenljivih sa prostim tipovima može obavljati konstruktorskog sintaksom. Tako smo i inicijalizaciju atributa “`indeks`“ mogli izvršiti u konstruktorskoj inicijalizacionoj listi (dok sa atributom “`ime_i_prezime`“ to nismo mogli

uraditi, jer on zahtijeva striktno kopiranje znak po znak funkcijom “`strcpy`”):

```
class Student {  
    char ime_i_prezime[50];  
    const int indeks;  
    Datum datum_rodjenja;  
  
public:  
    Student(const char ime[], int indeks, int dan, int mjesec, int godina) :  
        datum_rodjenja(dan, mjesec, godina), indeks(indeks) {  
        strcpy(ime_i_prezime, ime);  
    }  
    // Ovdje bi trebalo definirati ostatak interfejsa klase  
};
```

U ovom primjeru možemo uočiti dvije neobičnosti. Prvo, atribut “`indeks`” deklariran je sa kvalifikatorom “`const`”. Ovim atribut “`indeks`” postaje tzv. *konstantni* (ili *nepromjenljivi*) *atribut*. Takvim atributima se *ne može dodjeljivati vrijednost* (odnosno, njihova se vrijednost može samo čitati), tako da oni zadržavaju vrijednost kakvu su dobili prilikom inicijalizacije čitavo vrijeme dok postoji objekat kojem pripadaju. Kao posljedica te činjenice, konstantni atributi se mogu inicijalizirati *isključivo putem konstruktorskih inicijalizacionih listi* (s obzirom da im je nemoguće dodjeljivati vrijednost). U navedenom primjeru, deklariranje atributa “`indeks`” kao konstantnog atributa je sasvim opravdano, s obzirom da je broj indeksa konstantno svojstvo nekog studenta, koje ne bi trebalo da se mijenja tokom čitavog života objekta koji opisuje nekog konkretnog studenta. Druga neobičnost je upotreba konstrukcije “`indeks(indeks)`” u konstruktorskoj inicijalizacionoj listi. Iako ova konstrukcija djeluje pomalo čudno s obzirom da se isto ime javlja na dva mjesta, ona samo atribut “`indeks`” inicijalizira na vrijednost (istoimenog) formalnog parametra “`indeks`”. Primijetimo da u ovom slučaju, zbog same prirode sintakse, ne nastaje nejasnoća oko toga šta je atribut, a šta formalni parametar.

Treba primijetiti da čak i u slučaju kada je atribut konstantan, različiti primjeri klase mogu imati različitu vrijednost tog atributa. Međutim, interesantna situacija nastaje kada je neki atribut u isto vrijeme *statički* i *konstantan*. Njegova vrijednost je tada nepromjenljiva, a pored toga, ta vrijednost je *zajednička za sve primjerke te klase*. Takvi atributi se ne mogu inicijalizirati čak ni pomoću konstruktorskih inicijalizacijskih listi, jer bi tada različiti primjeri klase mogli izvršiti njegovu inicijalizaciju na različite vrijednosti. Zbog toga se takvi atributi inicijaliziraju *odmah prilikom deklaracije samog atributa*, na isti način kao što se vrši i deklaracija bilo koje konstante. Ovo je jedini izuzetak u kojem se inicijalizacija nekog atributa navodi odmah prilikom njegove deklaracije.

Postoji još jedan slučaj kada se atribut mora inicijalizirati u konstruktorskoj inicijalizacionoj listi. To je slučaj kada atribut predstavlja *referencu* na neki drugi objekat. Zamislimo, na primjer, da želimo voditi evidenciju o knjigama u studentskoj biblioteci koje su zadužili

pojedini studenti. Za tu svrhu prirodno je definirati klasu “Knjiga” koja će sadržavati osnovne informacije o svakoj knjizi, poput naslova, imena pisca, godine izdanja, žanra, itd. Međutim, pored osnovnih informacija o samoj knjizi, potrebno je imati informaciju *kod kojeg studenta* se nalazi knjiga. Naravno, besmisleno je u klasi “Knjiga” imati atribut “Student” u koji bismo smještali podatke o studentu koji je zadužio knjigu, s obzirom da ti podaci svakako već postoje negdje drugdje (recimo, u objektu koji opisuje tog studenta). Nas samo interesira *ko je zadužio knjigu*. Mogli bismo, na primjer, uvesti atribut koji bi čuvao recimo *broj indeksa* studenta koji je zadužio knjigu, tako da bismo, u slučaju potrebe, pretraživanjem svih studenata (npr. u nekom nizu studenata) mogli pronaći i ostale podatke o tom studentu (podrazumijeva se da ne postoje dva studenta sa istim brojem indeksa). Ovaj pristup ima dva nedostatka. Prvo, traženje studenata po indeksu u velikoj gomili studenata može biti vremenski zahtjevno. Drugo, postoji mogućnost da upišemo broj indeksa nepostojećeg studenta (tj. da upišemo da se knjiga nalazi kod nekog fiktivnog studenta koji uopće ne postoji u spisku studenata). Mnogo bolje rješenje je u klasu “Knjiga” uvesti atribut koji predstavlja *pokazivač na studenta koji je zadužio knjigu*. Preko takvog pokazivača mogli bismo, u slučaju potrebe, veoma lako i efikasno dobaviti sve podatke o odgovarajućem studentu. Stoga bi razumna deklaracija klase “Knjiga” mogla izgledati recimo ovako:

```
class Knjiga {
    char naslov[100], ime_pisca[50], zanr[30];
    const int godina_izdanja;
    Student *kod_koga_je;
public:
    // Ovdje bi trebalo definirati interfejs klase
};
```

Alternativno, umjesto pokazivača na studenta, mogli bismo koristiti *referencu na studenta* (koja je, u suštini, preruseni pokazivač). Na taj način postižemo dvije prednosti. Prvo, ukoliko koristimo referencu, možemo koristiti jednostavniju sintaksu (pristup referenci se automatski prevodi u pristup objektu na koji ona ukazuje, bez potrebe da ručno vršimo dereferenciranje, kao u slučaju pokazivača). Drugo, kako reference moraju biti inicijalizirane (tj. mora se zadati objekat za koji će biti vezane), ne bi se moglo desiti da atribut “*kod_koga_je*” ostane greškom neinicijaliziran. Međutim, upravo zbog činjenice da reference ne smiju ostati neinicijalizirane, njihova inicijalizacija se mora izvršiti u konstruktorskoj inicijalizacionoj listi. U svakom slučaju, bez obzira da li koristimo pokazivač ili referencu na studenta, konstruktor klase “Knjiga” kao jedan od parametara svakako mora imati i studenta koji je zadužio knjigu. Drugim riječima, jedan od formalnih parametara konstruktora trebao bi biti tipa “Student”, ili još bolje (glezano sa aspekta efikasnosti), konstantna referencia na tip “Student”.

Može se postaviti pitanje da li je bolje atribute inicijalizirati u konstruktorskoj inicijalizacionoj listi ili unutar samog tijela konstruktora. Odgovor je da je *neznatno efikasnije* izvršiti inicijalizaciju u konstruktorskoj inicijalizacionoj listi. Naime, u tom slučaju se inicijalizacija izvršava *uporedo sa stvaranjem objekta*, dok se u slučaju kada inicijalizaciju izvodimo u tijelu konstruktora *prvo kreiraju neinicijalizirani atributi*, koji se inicijaliziraju tek kad im se izvrši eksplicitna dodjela. Ovo podsjeća na razliku između deklaracije poput

```
int broj(5);
```

i naknadnog dodjeljivanja poput

```
int broj;  
broj = 5;
```

Naravno, atributi koji ne smiju ostati neinicijalizirani (objekti sa konstruktorima, konstantni atributi, atributi reference) moraju se inicijalizirati u konstruktorskoj inicijalizacionoj listi (tu nemamo izbora). Zbog opisane minorne razlike u efikasnosti, preporučuje se da se sve što je moguće inicijalizirati u konstruktorskoj inicijalizacionoj listi inicijalizira unutar nje. Stoga se često dešava da samo tijelo konstruktora ostane potpuno prazno. Na primjer, konstruktor klase “Kompleksni” mogao se napisati i ovako (sa praznim tijelom):

```
class Kompleksni {  
    double re, im;  
public:  
    Kompleksni(double r = 0, double i = 0) : re(r), im(i) {}  
    // Ovdje slijedi ostatak interfejsa klase...  
};
```

Ipak, bitno je naglasiti da kada koristimo konstruktorske inicijalizacione liste, svi atributi pomenuti u listi se inicijaliziraju *onim redoslijedom kako su deklarirani unutar klase, bez obzira na redoslijed navođenja u listi* (razlog za ovu prividnu nelogičnost vezan je za podršku nasljeđivanju, koja predstavlja još jedno važno svojstvo objektno orientiranog programiranja, o kojem ćemo govoriti kasnije). Tako će se, u ranije navedenom primjeru konstruktora klase “Student”, atribut “indeks” inicijalizirati *prije* atributa “datum_rodjenja”, bez obzira što je naveden *kasnije* u inicijalizacionoj listi! U većini slučajeva redoslijed inicijalizacije nije bitan, međutim u rijetkim situacijama kada je programeru bitan redoslijed kojim se inicijaliziraju pojedini atributi, treba voditi računa o ovoj činjenici. Naročito treba izbjegavati inicijalizaciju nekog atributa izrazom koji sadrži vrijednosti drugih atributa (u takvom slučaju redoslijed inicijalizacije može postati bitan).

Još je bitno naglasiti da ukoliko se konstruktor implementira *izvan deklaracije klase*, tada se konstruktorska inicijalizaciona lista navodi tek prilikom *implementacije* konstruktora, a ne prilikom navođenja njegovog prototipa. Na primjer, ukoliko bismo željeli da konstruktor klase “Student” implementiramo *izvan klase* (što svakako treba raditi kad god konstruktor sadrži mnogo naredbi), unutar deklaracije klase “Student” bi trebalo navesti *samo njegov prototip*:

```
class Student {  
    char ime_i_prezime[50];  
    int indeks;  
    Datum datum_rodjenja;
```

```

public:
    Student(const char ime[], int indeks, int d, int m, int g);
    // Ovdje bi trebalo definirati ostatak interfejsa klase
};

```

Sada bi implementacija konstruktora trebala izgledati ovako:

```

Student::Student(const char ime[], int indeks, int d, int m, int g) :
    datum_rodjenja(d, m, g), indeks(indeks) {
    strcpy(ime_i_prezime, ime);
}

```

Možda je nekome palo na pamet solomonsko rješenje koje bi u mnogim slučajevima eliminiralo potrebu za konstruktorskim inicijalizacijskim listama – uvijek definirati konstruktore bez parametara! Međutim, kao što smo vidjeli, ovakvo rješenje je veoma nefleksibilno, jer ne možemo utjecati na postupak inicijalizacije. Treba shvatiti da konstruktorske inicijalizacijske liste nisu uvedene da bi zakomplificirali život programerima, nego upravo suprotno – da olakšaju razvoj programa. Stoga, prihvativate sljedeći savjet: *nemojte koristiti kvazi-rješenja koja predstavljaju samo liniju manjeg otpora.* Konstruktore bez parametara definirajte samo u slučaju kada neka klasa *zaista treba da ima konstruktor bez parametara*, a ne samo zato što ste lijeni da kasnije poduzmete sve što treba poduzeti zbog činjenice što klasa nema konstruktor bez parametara (još gore rješenje je da uopće ne definirate konstruktore). Možda Vam se čini da je tako lakše. Međutim, ukoliko tako postupite, na duži rok će se pokazati upravo suprotno: *biće vam mnogo teže*, pogotovo kad program ne bude radio onako kao što očekujete...

Konstruktori predstavljaju idealno rješenje za situacije u kojima se javlja potreba za dinamičkom alokacijom memorije (i ujedno predstavljaju idealno mjesto gdje treba izvršiti dinamičku alokaciju). Da bismo ovo uvidjeli, razmotrimo jedan konkretni primjer. Pretpostavimo da želimo napraviti klasu (nazovimo je “*VektorN*”) koja će predstavljati n -dimenzionalni vektor (tj. vektor koji se opisuje sa n koordinata), pri čemu je dimenzija n nije fiksna, nego ju je moguće zadavati. Najprirodnije rješenje je uvesti atribute “*br_elemenata*” i “*elementi*”, koji respektivno predstavljaju dimenziju vektora i dinamički alociran niz komponenti vektora. Radi jednostavnosti, ovu klasu ćemo napisati samo sa minimalnim interfejsom, koji čini konstruktor bez parametara koji inicijalizira dimenziju vektora na 0, a pokazivač na elemente na *nul-pokazivač*, zatim metoda “*Stvori*” koja vrši dinamičku alokaciju, metoda “*Unisti*” koja briše alocirani prostor, metoda “*Ispisi*” koja u vitičastim zagradama ispisuje komponente vektora međusobno razdvojene zarezom, kao i dvije metode nazvane “*Element*” (jedna je konstantna, a druga nije) koje omogućavaju pristup komponentama vektora:

```

class VektorN {
    int br_elemenata;
    double*elementi;
public:
    VektorN() : br_elemenata(0), elementi(0) {}
    void Stvori(int n) { br_elemenata = n; elementi = new double[n]; }
    void Unisti() { delete[] elementi; }
    void Ispisi() const;
    double Element(int n) const;
    double &Element(int n);
};

```

Slijede i implementacija metode “*Ispisi*” i “*Element*” koje nisu implementirane unutar deklaracije klase:

```

void VektorN::Ispisi() const {

```

```

        cout << "{";
    for(int i = 0; i < br_elemenata - 1; i++)
        cout << elementi[i] << ",";
    cout << elementi[br_elemenata - 1] << "}";
}

double VektorN::Element(int n) const {
    if(n < 1 || n > br_elemenata) throw "Pogrešan indeks!\n";
    return Elementi[n - 1];
}

double &VektorN::Element(int n) {
    if(n < 1 || n > br_elemenata) throw "Pogrešan indeks!\n";
    return Elementi[n - 1];
}

```

Implementacije dvije metode nazvane “Element“ su posebno interesantne. Prvo, primijetimo da obje verzije imaju posve isto tijelo. U obje metode se prvo provjerava da li je indeks u dozvoljenom opsegu, i ukoliko nije, baca se izuzetak. Dalje, postignuto je da se indeksi numeriraju od *jedinice*, kao što je uobičajeno u matematici. Međutim, prva verzija metode “Element“ vraća *vrijednost* odgovarajućeg elementa niza “elementi”, dok druga verzija vraća *referencu* na odgovarajući element niza. Stoga se prva verzija (konstantna) ne može koristiti sa lijeve strane znaka jednakosti (tj. kao l-vrijednost), dok druga verzija može. Sjetimo se da u slučaju da postoje dvije metode istog imena od kojih je jedna konstantna a druga nije, tada se konstantna verzija poziva samo ukoliko se primijeni nad konstantnim objektom. U svim ostalim slučajevima, poziva se nekonstantna verzija metode. Na ovaj način je sprječeno da se poziv metode “Element“ nade sa lijeve strane jednakosti ukoliko je pozvana nad konstantnim objektom, i na taj način omogući promjena niza na koji pokazuje pokazivač “elementi”. Slijedi jednostavan primjer koji demonstrira upotrebu napisane klase:

```

VektorN v1, v2;
try {
    v1.Stvorи(5); v2.Stvorи(3);
    v1.Element(1) = 3; v1.Element(2) = 5; v1.Element(3) = -2;
    v1.Element(4) = 0; v1.Element(5) = 1;
    v2.Element(1) = 3; v2.Element(2) = 0; v2.Element(2) = 2;
    v1.Ispisi(); v2.Ispisi();
}
catch(...) {
    cout << "Problemi sa memorijom!\n";
}
v1.Unisti(); v2.Unisti();

```

Pojasnimo još malo zbog čega smo uveli dvije verzije metode “Element“. Da smo definirali samo jednu verziju ove metode, koja nije deklarirana kao konstantna, ne bismo mogli pristupati elementima konstantnih vektora. Da smo definirali samo konstantnu metodu “Element“ koja vraća vrijednost odgovarajućeg elementa niza (a ne referencu na njega), mogli bismo samo čitati, ali ne i mijenjati elemente niza. Da smo definirali samo konstantnu metodu “Element“ koja vraća referencu na odgovarajući element niza, mogli bismo promijeniti elemente niza, čak i ukoliko je objekat tipa “VektorN” prethodno deklariran kao konstantan. Na prvi pogled ovo izgleda nemoguće, s obzirom da je metoda “Element“ deklarirana kao konstantna. Međutim, metoda “Element“ zaista ne mijenja niti jedan od atributa klase “VektorN”. Ona samo vraća referencu na element dinamički kreiranog niza, a sama izmjena obavlja se potpuno izvan same metode. Opisano rješenje sa dvije verzije metode “Element“ je zaista jedino koje omogućava konzistentno ponašanje u svim situacijama!

Alternativno smo umjesto napisanih metoda nazvanih "Element" mogli napisati dvije posebne metode "PostaviElement" i "CitajElement", od kojih bi metoda "CitajElement" bila deklarirana kao konstantna. Tada bismo za postavljanje elementa umjesto konstrukcija poput "v1.Element(1) = 3" koristili konstrukciju poput "v1.PostaviElement(1, 3)", dok bismo za čitanje elementa koristili metodu "CitajElement", kao na primjer u konstrukciji poput "cout << v1.CitajElement(1)". Međutim, gore napisano rješenje je univerzalnije. U svakom slučaju, ovo je samo privremeno rješenje dok ne naučimo kako da klasu "VektorN" proširimo tako da može direktno podržavati indeksiranje, tj. da možemo direktno pisati "v1[1] = 3" i "cout << v1[1]" (uskoro ćemo vidjeti da je i ovo moguće).

Bitno je primijetiti da neće nastati nikakvi problemi zbog pozivanja metode "Unisti" u slučaju da kreiranje nekog objekta nije uspjelo, zbog činjenice da je konstruktor inicijalizirao pokazivače na nul-pokazivač, a znamo da operator "**delete**" ne radi ništa u slučaju kada se primijeni na nul-pokazivač. U ovom slučaju, konstruktor nas automatski rješava mnogih briga (sjetimo se primjera u ranijim poglavljima u kojima smo se o ovom problemu morali sami brinuti).

Mada prikazano rješenje radi lijepo, u njemu se javlja jedna nedoslijednost. Primijetimo da se prvo pri deklaraciji objekta tipa "VektorN" on inicijalizira na prazan vektor, a zatim mu dimenzionalnost određuje *naknadno*. Mnogo je pametnije umjesto konstruktora bez parametara uvesti konstruktor sa jednim parametrom koji predstavlja dimenziju vektora, i koji će odmah pri deklaraciji vektora izvršiti dinamičku alokaciju memorije. Na taj način ćemo biti sigurni da vektor ima ispravnu dimenzionalnost odmah nakon što je stvoren, i nećemo imati problema ukoliko slučajno zaboravimo pozvati metodu "Stvori". Ova metoda zapravo više neće ni postojati, a njenu ulogu preuzeće *konstruktor sa jednim parametrom*. Tako više nećemo moći ni deklarirati vektore čija dimenzija nije specificirana. Umjesto toga, željenu dimenzionalnost vektora zadavaćemo prilikom njegove deklaracije. Slijedi nova deklaracija klase "VektorN" u kojoj je dinamička alokacija memorije povjerena konstruktoru:

```
class VektorN {
    int br_elemenata;
    double *elementi;
public:
    explicit VektorN(int n) : br_elemenata(n), elementi(new double[n]) {}
    void Unisti() { delete[] elementi; }
    void Ispisi() const;
    double Element(int n) const;
    double &Element(int n);
};
```

Razloge zbog čega smo ispred konstruktora ubacili ključnu riječ "**explicit**" uvidićemo uskoro. Također, interesantno je uočiti da je čak i sama dinamička alokacija izvršena unutar konstruktorske inicijalizacione liste, tako da je samo tijelo konstruktora ostalo prazno.

Ovako napisani konstruktor samo vrši dinamičku alokaciju niza za smještanje elemenata vektora, ali njegovi elementi ostaju nedefinirani (isto je vrijedilo i za ranije napisanu metodu "Stvori"). Međutim, nikakav problem nije prepraviti konstruktor da inicijalizira i elemente kreiranog niza na neku željenu inicijalnu vrijednost (recimo, na nulu):

```
explicit VektorN(int n) : br_elemenata(n), elementi(new double[n]) {
    for(int i = 0; i < n; i++) elementi[i] = 0;
}
```

Na taj način, pomoću konstruktora možemo precizno definirati kakav će objekat biti odmah po njegovom stvaranju. Slijedi testni primjer kojim ćemo testirati napisanu klasu:

```

try {
    VektorN v1(5), v2(3);
    v1.Element(1) = 3; v1.Element(2) = 5; v1.Element(3) = -2;
    v1.Element(4) = 0; v1.Element(5) = 1;
    v2.Element(1) = 3; v2.Element(2) = 0; v2.Element(3) = 2;
    v1.Ispisi(); v2.Ispisi();
    v1.Unisti(); v2.Unisti();
}
catch(...) {
    cout << "Problemi sa memorijom!\n";
}

```

Primijetimo da je sintaksa kojom se dimenzioniraju objekti tipa “VektorN” analogna sintaksi kojom se dimenzioniraju promjenljive tipa “vector” iz istoimene biblioteke. Ovo nije slučajno: dimenzioniranje objekata tipa “vector” upravo obavlja konstruktor klase “vector”!

U ovom primjeru smo uništavanje vektora “v1” i “v2” morali prebaciti unutar “**try**“ bloka, jer su ovi vektori definirani *lokalno* u “**try**“ bloku (što smo morali učiniti da bismo mogli uhvatiti izuzetak koji bi eventualno mogao biti bačen iz konstruktora), pa kao takvi ne postoje izvan ovog bloka. Međutim, u ovom slučaju može nastati problem ukoliko stvaranje objekta “v1” uspije, a stvaranje objekta “v2“ ne uspije. Izuzetak koji će pri tome biti bačen biće uhvaćen u “**catch**“ bloku, ali nakon toga više nemamo mogućnost da obrišemo objekat “v1“ (koji više nije u vidokrugu)!

Jedno moguće rješenje opisanog problema moglo bi biti bi korištenje ugniježdenih “**try**“ – “**catch**“ konstrukcija, ali takvo rješenje je zaista veoma ružno. Da bi se izbjegli ovakvi problemi, a također i problemi koji mogu nastati zbog činjenice da je veoma lako *zaboraviti* eksplisitno pozvati metodu za uništavanje objekta (u našem primjeru metodu “*Unisti*“), što bi svakako dovelo do curenja memorije, u jezik C++ su pored konstruktora uvedeni i *destruktori*. Destruktor je, poput konstruktora, neka vrsta funkcije članice klase koja se *automatski poziva* onog trenutka kada objekat te klase prestaje postojati (tipično na kraju bloka unutar kojeg je objekat deklariran, ili po izlasku iz funkcije u kojoj je objekat deklariran pomoću naredbi “**return**“ ili “**throw**“). Zadatak destruktora je da oslobodi dodatne resurse koje je objekat zauzeo prilikom svog stvaranja, odnosno prilikom poziva konstruktora (ti resursi su obično *dinamički alocirana memorija*, ali mogu biti i otvorene datoteke na disku, i druge stvari). Slično konstruktoru, destruktori također nemaju povratni tip, ali za razliku od konstruktora, oni *nikada nemaju parametara*, a ime im je isto kao ime klase u kojoj se nalaze, samo sa prefiksom “~” (tilda) ispred imena. Slijedi primjer klase “VektorN“ u kojoj smo definirali destruktur (njegovim uvođenjem otpala je potreba za metodom “*Unisti*“):

```

class VektorN {
    int br_elemenata;
    double *elementi;
public:
    explicit VektorN(int n) : br_elemenata(n), elementi(new double[n]) {}
    ~VektorN() { delete[] elementi; }
    void Ispisi() const;
    double Element(int n) const;
    double &Element(int n);
};

```

Sada upotreba klase “VektorN“ postaje mnogo jednostavnija, jer se više ne moramo eksplisitno brinuti o uništavanju objekta (preciznije, dinamički alocirane memorije koja je zauzeta prilikom njegovog stvaranja):

```

try {
    VektorN v1(5), v2(3);
    v1.Element(1) = 3; v1.Element(2) = 5; v1.Element(3) = -2;
    v1.Element(4) = 0; v1.Element(5) = 1;
    v2.Element(1) = 3; v2.Element(2) = 0; v2.Element(2) = 2;
    v1.Ispisi(); v2.Ispisi();
}
catch(...) {
    cout << "Problemi sa memorijom!\n";
}

```

U ovom slučaju će se nad objektima “v1” i “v2” automatski pozvati njihovi destruktori nakon kraja bloka unutar kojeg su definirani, tako da se ne može desiti da zaboravimo oslobođiti memoriju. Također, lijepo je stvar što se destruktori pozivaju samo nad objektima koji su *zaista stvoreni* (pod stvorenim objektom smatramo objekat čiji se konstruktor završio regularno, a ne bacanjem izuzetka). Tako, ukoliko na primjer svaranje objekta “v1” uspije, a objekta “v2” ne uspije, doći će do bacanja izuzetka i napuštanja “**try**“ bloka. Tada će automatski biti pozvan destruktur nad objektom “v1”, koji će ga uništiti. Destruktur nad objektom “v2“ koji nije stvoren neće se ni pozvati, a to nam upravo i treba.

Iz izloženog se vidi da su destruktori veoma korisni, i oslobođaju nas mnogih problema, a njihova deklaracija je sasvim jednostavna. Zbog toga se može smatrati pravilom da svaka klasa koja ima konstruktor u kojem se vrši dodatno zauzimanje računarskih resursa (tipičan primjer je dinamička alokacija memorije) obavezno mora imati destruktur, koji će oslobođiti sve resurse koji su dodatno zauzeti prilikom izvršavanja konstruktora. Ipak, postoji i jedan potencijalni problem o kojem se dosta rijetko govori (ali nažalost sam problem nije rijedak) koji može nastupiti kod klase koje posjeduju destruktore. O ovom problemu ćemo govoriti u sljedećem poglavlju, kada budemo objašnjavali razloge za uvođenje tzv. *konstruktora kopije*.

Već je rečeno da se destruktori uopće ne pozivaju ukoliko objekat nije stvoren, tj. ukoliko se konstruktor nije do kraja izvršio. To ima za posljedicu da konstruktor nikada ne smije ostaviti iza sebe *polovično stvoren objekat*, jer tada niko neće obrisati niti će biti u stanju da obriše ono što je iza sebe ostavio konstruktor. Drugim riječima, konstruktor prije nego što baci izuzetak (ukoliko utvrdi da ga mora baciti) mora iza sebe počistiti svo “smeće” koje je iza sebe ostavio. To se tipično dešava ukoliko se u konstruktoru dinamički alocira više nizova: ukoliko se prvo izvrši nekoliko uspješnih alokacija, a zatim jedna neuspješna, konstruktor prije nego što baci izuzetak treba da pobriše sve uspješne alokacije (jer ih u protivnom niko neće obrisati). To se može uraditi tako što se unutar konstruktora ugraditi “**try**” blok koji će uhvatiti eventualno neuspješnu alokaciju, nakon čega se u “**catch**” bloku može “počistiti zaostalo smeće” prije nego što se zaista baci izuzetak iz konstruktora. Ovo ćemo demonstrirati u sljedećem poglavlju, u primjeru koji definira klasu “*Matrica*” sa konstruktorima i destruktorma.

Bitno je napomenuti da je veoma nepreporučljivo bacati izuzetke iz *destruktora* (srećom, ovo je rijetko potrebno). Mada je to u principu dozvoljeno, bacanje izuzetaka iz destruktora može u nekim situacijama izazvati veoma čudne efekte, tako da je bacanje izuzetaka iz destruktora najbolje potpuno izbjegći. Također, destruktore nikada ne treba eksplicitno pozivati kao obične funkcije članice, mada sintaksa to dozvoljava. Ukoliko to učinite, samo tražite sebi probleme. Destruktore treba pustiti da se automatski pozivaju tamo gdje je to potrebno, a inače ih treba ostaviti na miru. Ukoliko Vam je potrebno da iz neke druge funkcije izvršite posve iste naredbe koje su sadržane u destruktoru, nemojte pozivati destruktur kao funkciju, nego definirajte pomoćnu funkciju (tipično u privatnoj sekcijsi klase) unutar koje ćete smjestiti te naredbe, a zatim tu pomoćnu funkciju pozovite iz destruktora, i sa bilo kojeg drugog mesta gdje su Vam te naredbe potrebne.

Destruktori se također automatski pozivaju i pri upotrebi operatora “**delete**” i “**delete[]**” u slučaju da smo sa “**new**” stvorili objekat koji sadrži konstruktoare i destruktore. Na primjer, ukoliko smo *dinamički*

stvorili neki objekat tipa “VektorN” naredbom poput

```
VektorN *pok = new VektorN(5);
```

tada će njegovo brisanje pomoći naredbe

```
delete pok;
```

izazvati dva efekta: prvo će nad dinamički stvorenim objektom “*pok” biti pozvan destruktor (koji će oslobođiti memoriju koja je zauzeta u konstruktoru objekta), pa će tek tada biti oslobođena memorija koji je sam objekat zauzimao. Slično, prilikom upotrebe operatora “**delete[]**” za brisanje nekog dinamički stvorenog niza, prije samog brisanja nad *svakim elementom niza* biće pozvan njegov destruktor (ukoliko takav postoji). Ovim je pokazana jedna (od mnogih) razlika između operatora “**delete**” i “**delete[]**” (u slučaju da smo umjesto “**delete[]**” stavili “**delete**”, destruktor bi bio izvršen samo *jednom*, a ne nad svakim elementom niza), koja ukazuje da ova dva operatora ne treba miješati, i da svaki treba koristiti za ono za šta je namijenjen!

Ostali smo dužni da objasnimo zbog čega smo konstruktor klase “VektorN” definirali kao eksplisitni konstruktor. Da ovo nismo uradili, sasvim bi moguće bilo napisati nešto poput

```
v1 = 7;
```

Na osnovu specijalne uloge koju imaju konstruktori sa *jednim parametrom* (automatska pretvorba tipova), ovakva naredba bila bi interpretirana kao

```
v1 = VektorN(7);
```

čime bi se postigao sasvim neočekivan efekat: stvorio bi se novi, sedmodimenzionalni vektor, koji bi bio dodijeljen objektu “v1” (a usput bi se pojavio i mnogo ozbiljniji problem uzrokovan plitkim kopiranjem, o kojem ćemo govoriti u sljedećem poglavlju). Slična neočekivana situacija nastala bi ukoliko bi nekoj funkciji koja očekuje objekat tipa “VektorN” kao parametar bio proslijeden broj (taj broj bi bio proslijeden konstruktoru sa jednim parametrom, nakon čega bi konstruisani objekat bio proslijeden funkciji, a to sigurno nije željeno ponašanje). Ovako, označavanjem konstruktora sa “**explicit**”, zabranjuje se njegovo korištenje za automatsku pretvorbu tipova iz cjelobrojnog u tip “VektorN”, tako da ovakve besmislene konstrukcije neće biti ni dozvoljene (tj. dovešće do prijave greške od strane kompjajlera).

Većinu stvari o kojima smo do sada govorili ilustrira sljedeći program, koji obavlja istu funkciju kao program za obradu rezultata učenika u razredu koji je bio napisan u poglavlju o strukturama, samo što je napisan u duhu objektno orientiranog programiranja. Program je prilično dugačak i relativno složen, pa će nakon njegovog prikaza uslijediti detaljna analiza njegovih pojedinih dijelova:

```
#include <iostream>
#include <cstring>
#include <iomanip>
#include <algorithm>

using namespace std;

const int BrojPredmeta(12); // Pri testiranju smanjite ovaj broj

class Datum {
    int dan, mjesec, godina;
public:
    Datum(int d, int m, int g);
```

```

void Ispisi() const {
    cout << dan << "." << mjesec << "." << godina;
}
};

class Ucenik {
    char ime[20], prezime[20];
    Datum datum_rodjenja;
    int ocjene[BrojPredmeta];
    double prosjek;
    bool prolaz;
public:
    Ucenik(const char ime[], const char prezime[], int d, int m, int g);
    void UpisiOcjenu(int predmet, int ocjena);
    double Prosjek() const { return prosjek; }
    bool Prolaz() const { return prolaz; }
    void Ispisi() const;
};

class Razred {
    const int kapacitet;
    int broj_evidentiranih;
    Ucenik **ucenici;
    static bool BoljiProsjek(const Ucenik *u1, const Ucenik *u2) {
        return u1->Prosjek() > u2->prosjek();
    }
public:
    explicit Razred(int broj_ucenika) : kapacitet(broj_ucenika),
        broj_evidentiranih(0), ucenici(new Ucenik*[broj_ucenika]) {}
    ~Razred();
    void EvidentirajUcenika(Ucenik *ucenik);
    void UnesiNovogUcenika();
    void IspisiIzvjestaj() const;
    void SortirajUcenike() {
        sort(ucenici, ucenici + broj_evidentiranih, BoljiProsjek);
    }
};

int main() {
    int broj_ucenika;
    cout << "Koliko ima ucenika: ";
    cin >> broj_ucenika;
    try {
        Razred razred(broj_ucenika);
        for(int i = 1; i <= broj_ucenika; i++) {
            cout << "Unesi podatke za " << i << " ucenika:\n";
            razred.UnesiNovogUcenika();
        }
        razred.SortirajUcenike();
        razred.IspisiIzvjestaj();
    }
    catch(...) {
        cout << "Problemi sa memorijom!\n";
    }
}

Datum::Datum(int d, int m, int g) {
    int broj_dana[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
}

```

```

    if(g % 4 == 0 && g % 100 != 0 || g % 400 == 0) broj_dana[1]++;
    if(g < 1 || d < 1 || m < 1 || m > 12 || d > broj_dana[m - 1])
        throw "Neispravan datum!\n";
    dan = d; mjesec = m; godina = g;
}

Ucenik::Ucenik(const char ime[], const char prezime[], int d, int m,
    int g) : datum_rodjenja(d, m, g), prosjek(1), prolaz(false) {
    if(strlen(ime) > 19 || strlen(prezime) > 19)
        throw "Predugacko ime ili prezime!\n";
    strcpy(Ucenik::ime, ime);
    strcpy(Ucenik::prezime, prezime);
    for(int i = 0; i < BrojPredmeta; i++) ocjene[i] = 1;
}

void Ucenik::UpisiOcjenu(int predmet, int ocjena) {
    if(ocjena < 1 || ocjena > 5) throw "Pogrešna ocjena!\n";
    if(predmet < 1 || predmet > BrojPredmeta)
        throw "Pogresan broj predmeta!\n";
    ocjene[predmet - 1] = ocjena;
    prosjek = 1; prolaz = false;
    double suma_ocjena(0);
    for(int i = 0; i < BrojPredmeta; i++) {
        if(ocjene[i] == 1) return;
        suma_ocjena += ocjene[i];
    }
    prosjek = suma_ocjena / BrojPredmeta; prolaz = true;
}

void Ucenik::Ispisi() const {
    cout << "Ucenik " << ime << " " << prezime << " rođen ";
    datum_rodjenja.Ispisi();
    if(Prolaz())
        cout << " ima prosjek " << setprecision(2) << Prosjek() << endl;
    else
        cout << " mora ponavljati razred\n";
}

Razred::~Razred() {
    for(int i = 0; i < broj_evidentiranih; i++) delete ucenici[i];
    delete[] ucenici;
}

void Razred::EvidencirajUcenika(Ucenik *ucenik) {
    if(broj_evidentiranih >= kapacitet) throw "Previše ucenika!\n";
    ucenici[broj_evidentiranih++] = ucenik;
}

void Razred::UnesiNovogUcenika() {
    bool pogresan_unos(true);
    while(pogresan_unos) {
        char ime[20], prezime[20];
        int d, m, g;
        cout << "Ime: "; cin >> setw(20) >> ime;
        cout << "Prezime: "; cin >> setw(20) >> prezime;
        cout << "Dan rođenja: "; cin >> d;
        cout << "Mjesec rođenja: "; cin >> m;
        cout << "Godina rođenja: "; cin >> g;
        try {

```

```

Ucenik *ucenik = new Ucenik(ime, prezime, d, m, g);
for(int predmet = 1; predmet <= BrojPredmeta; predmet++) {
    int ocjena;
    cout << "Ocjena iz " << predmet << ". predmeta: ";
    cin >> ocjena;
    ucenik->UpisiOcjenu(predmet, ocjena);
}
EvidentirajUcenika(ucenik);
pogresan_unos = false;
}
catch(const char greska[]) {
    cout << "Greska: " << greska << endl
        << "Molimo, ponovite unos!\n";
}
}

void Razred::IspisiIzvjestaj() const {
    cout << endl;
    for(int i = 0; i < BrojEvidentiranih; i++)
        Ucenici[i]->Ispisi();
}

```

U ovom programu su definirane tri klase: "Datum", "Ucenik" i "Razred". O klasi "Datum" nema se ništa više reći nego što je do sada već rečeno. Klasa "Ucenik" deklarira atribute "ime", "prezime", "datum_rodjenja", "ocjene", "projek" i "prolaz", pri čemu atribut "ocjene" predstavlja klasični niz čiji je broj elemenata određen globalnom konstantom "BrojPredmeta". U interfejsu klase "Ucenik" nalazi se konstruktor koji inicijalizira atribute "ime", "prezime" i "datum_rodjenja" u skladu sa proslijedenim parametrima (za postavljanje datuma rođenja koriste se 3 parametra), sve ocjene inicijalizira na 1 (tako da se smatra da učenik nije zadovoljio predmet sve dok ne dobije pozitivnu ocjenu iz njega), dok atribute "projek" i "prolaz" inicijalizira respektivno na vrijednosti "1" i "false" (atribut "prolaz" sadrži logičku vrijednost "true" ukoliko je učenik prošao a logičku vrijednost "false" ako nije). Dalje su deklarirane i implementirane trivijalne metode "Projek" i "Prolaz" koje vraćaju respektivno vrijednosti atributa "projek" i "prolaz", kao i metoda "Ispis" koja ispisuje podatke o učeniku. Pored toga, imamo i metodu "UpisiOcjenu" sa dva parametra koji predstavljaju broj predmeta i ocjenu. Ova metoda vrši upis odgovarajuće ocjene, vodeći računa da se ne zada pogrešan broj predmeta ili ocjena izvan opsega od 1 do 5 (u suprotnom se baca izuzetak), a zatim ažurira atribute "projek" i "prolaz" u skladu sa novonastalom situacijom.

Interesantno je napomenuti da su se atribute "projek" i "prolaz" mogli izostaviti, pri čemu bi se tada metode "Projek" i "Prolaz" trebale izmijeniti tako da prilikom poziva *računaju* projek i indikator prolaznosti, umjesto da prosto vrate vrijednosti odgovarajućih atributa. Naravno, metoda "UpisiOcjenu" tada ne bi trebala računati projek i indikator prolaznosti. Korisnik tako izmijenjene klase "Ucenik" ne bi primijetio nikakvu izmjenu. Ovaj primjer ilustrira način na koji interfejs klase sakriva njenu internu implementaciju od korisnika klase. Ipak, implementacija za koju smo se mi odlučili nešto je efikasnija, jer se projek i indikator prolaznosti računaju samo prilikom upisa nove ocjene, a uz izmijenjenu implementaciju, oni bi se iznova računali pri svakom pozivu metode "Projek" odnosno "Prolaz". Ovo poboljšanje naročito dolazi do izražaja prilikom sortiranja spiska učenika, u kojem se veoma često poziva metoda "Projek".

Klasa "Razred" je nešto složenija, i nju ćemo razmotriti malo detaljnije. Njeni atribute su dva cjelobrojna polja "kapacitet" i "broj_evidentiranih", kao i dvojni pokazivač "ucenici". Konstantni atribut "kapacitet" predstavlja kapacitet razreda, odnosno maksimalni broj učenika koji

razred može primiti, dok atribut "broj_evidentiranih" predstavlja broj učenika koji su zaista evidentirani u razredu. Preko pokazivača "ucenici" pristupa se dinamički alociranom nizu koji predstavlja niz pokazivača na učenike (zbog toga je pokazivač "ucenici" dvojni pokazivač). Niz pokazivača na učenike se koristi umjesto niza učenika da se izbjegnu problemi o kojima smo govorili u vezi sa nizovima klase i konstruktorima. Konstruktor klase "Razred" postavlja atribut "kapacitet" na vrijednost zadalu parametrom, atribut "broj_evidentiranih" postavlja na nulu, i vrši dinamičku alokaciju niza preko pokazivača "ucenici" u skladu sa željenim kapacitetom razreda.

Interfejs klase "Razred" dalje sadrži metode "EvidentirajUcenika", "UnesiNovogUcenika", "IspisiIzvjestaj" i "SortirajUcenike". Metoda "EvidentirajUcenika" prima kao parametar pokazivač na objekat tipa "Ucenik", smješta ga u dinamički niz evidentiranih učenika (preko pokazivača "ucenici") i povećava broj evidentiranih učenika za 1. Pri tome se baca izuzetak ukoliko je razred eventualno već popunjeno. Metoda "UnesiNovogUcenika" traži da se sa tastature unesu osnovni podaci o jednom učeniku (ime, prezime i datum rođenja), kreira dinamički novi objekat tipa "Ucenik" koji inicijalizira unesenim podacima, traži da se unesu ocjene za učenika iz svih predmeta, upisuje unesene ocjene u kreirani objekat (pozivom metode "UpisiUcjenu") i, konačno, evidentira učenika (tj. upisuje ga u dinamički kreirani niz učenika) pozivom metode "EvidentirajUcenika". Pri tome se hvataju svi izuzeci koji bi eventualno mogli biti bačeni (npr. zbog pogrešno unesenog datuma), a unos se ponavlja sve dok se ne unesu ispravni podaci. Metoda "IspisiIzvjestaj" ispisuje izvještaj o svim upisanim učenicima u razredu prostim pozivom metode "Ispisi" nad svakim od upisanih učenika. Primijetimo da se metode "UpisiOcjenu" i "Ispisi" pozivaju *indirektno* (tj. pomoću operatora "->") jer se primjenjuju na *pokazivač* a ne na sam objekat.

Metoda "SortirajUcenike" je krajnje jednostavna, s obzirom da se njena implementacija sastoji samo od poziva funkcije "sort" iz biblioteke "algorithm". Međutim, za tu potrebu je potrebno definirati funkciju koja definira kriterij po kojem se vrši sortiranje. Za tu svrhu, u privatnoj sekciji klase "Razred" definirana je statička funkcija članica "BoljiProsjek" koja definira traženi kriterij. Treba naglasiti da je ova funkcija članica morala biti deklarirana kao statička funkcija članica. U suprotnom, ona ne bi mogla biti prosto proslijedena funkciji "sort" kao parametar, jer se nestatičke funkcije članice moraju pozivati nad nekim konkretnim objektom, a nad kojim, to funkcija "sort" zaista ne može saznati. Generalno, bilo kojoj funkciji koja kao formalni parametar ima pokazivač na funkciju možemo kao stvarni parametar proslijediti *običnu funkciju* ili *statičku funkciju članicu* (pod uvjetom da su im broj i tip parametara kao i tip povratne vrijednosti odgovarajući), ali ne i *nestatičku funkciju članicu*. Ovo je jedan od primjera u kojima se ne mogu koristiti nestatičke funkcije članice (doduše postoje i tzv. *pokazivači na nestatičke funkcije članice* i posebni operatori ".*" i "->*" koji se sa njima koriste, ali ta tematika je isuvrše napredna i o njoj nećemo govoriti). Naravno, kao alternativu, funkciju kriterija smo mogli definirati i kao običnu funkciju, ali je ovakvo rješenje bolje, s obzirom da se funkcija "BoljiProsjek" ne treba koristiti nigdje osim unutar metode "SortirajUcenike" kao parametar funkcije "sort".

Destruktor klase "Razred" je posebno interesantan. On svakako briše dinamički alociran niz učenika (preciznije, niz pokazivača na učenike) kojem se pristupa preko pokazivaca "ucenici" i koji je kreiran unutar konstruktora, ali prije toga briše i sve dinamički kreirane učenike na koje pokazuju elementi ovog niza, bez obzira što oni nisu kreirani unutar konstruktora, nego na nekom drugom mjestu (preciznije, u metodi "UnesiNovogUcenika"). Bez obzira na mjesto njihovog kreiranja, nakon poziva metode "EvidentirajUcenika" klasa "Razred" u izvjesnom smislu "zna" za njih, i u stanju je da ih i obriše. Ovaj primjer ilustrira da se destrukturu može povjeriti i "generalno čišćenje" svih resursa koji su na bilo koji način vezani za objekat koji se uništava, a ne nužno samo resursa zauzetih unutar konstruktora klase. Kažemo da je klasa "Razred" *vlasnik* (engl. *owner*) svih svojih učenika, odnosno klasa "Razred" *posjeduje* objekte tipa "Ucenik". Bilo koja klasa koja je vlasnik objekata neke druge klase, odgovorna je i za njihovo uništavanje. O tome da li nekoj klasi treba prepustiti vlasništvo nad drugim objektima ili

objekte treba pustiti da se brinu sami o sebi, postoje brojne diskusije. Generalnog odgovora na ovo pitanje nema, i preporučena strategija zavisi od slučaja do slučaja. Uglavnom, za koju god strategiju se odlučimo, ukoliko ne pazimo dobro šta radimo, i ukoliko nismo dosljedni u primjeni izabrane strategije, postoji velika mogućnost da stvorimo viseće pokazivače (ovo se obično dešava ukoliko u jednom dijelu programa dođe do uništavanja nekog objekta za koji se u drugom dijelu programa podrazumijeva da će i dalje postojati). Može se reći da je u objektno orijentiranom programiranju problem vlasništva veoma teško pitanje (kao, uostalom, i u stvarnom životu).

Ostaje još da se osvrnemo na glavni program (funkciju “main“). Nakon što smo praktično sve poslove povjerili klasama, glavni program postaje trivijalan. U njemu se, nakon što se sa tastature unese željeni broj učenika u razredu, prvo deklarira jedna konkretna instanca klase “Razred“ sa traženim kapacitetom, a zatim se nad ovom instancom u petlji poziva metoda “UnesiNovogUcenika“ sve dok se ne unesu svi učenici. Nakon toga se pozivom metode “SortirajUcenike“ svi učenici sortiraju u opadajući redoslijed po prosjeku, i na kraju se pozivom metode “IspisiIzvjestaj“ ispisuje traženi izvještaj. Sve ovo je uklopljeno u “**try**“ blok koji hvata eventualne izuzetke koji mogu biti bačeni sa raznih mesta ukoliko neka od dinamičkih alokacija memorije ne uspije.

Čitatelju odnosno čitateljki će sigurno upasti u oči da je ovako napisan program skoro dvostruko duži od sličnog programa koji je koristio samo strukture i bio napisan u čisto proceduralnom duhu. Zbog toga se postavlja pitanje da li se uloženi trud isplati. Odgovor je svakako *potvrđan*, s obzirom da je posljednji program mnogo pogodniji za eventualna proširenja koja se mogu lako realizirati proširujući razvijene klase novim metodama, pri čemu od velike pomoći mogu biti metode koje su do tada razvijene. Pored toga, razvijeni program ima strogi sistem zaštite od unosa besmislenih podataka, što nije bio slučaj sa izvornim programom.

32. Konstruktor kopije i prekopljeni operator dodjele

U prethodnom poglavlju smo se upoznali sa konstruktorima i destruktorma, kao i prednostima koje njihova upotreba donosi. Dok bi konstruktore trebala da posjeduje praktično svaka klasa, destruktori su potrebni samo ukoliko klasa koristi dodatne resurse u odnosu na resurse koji predstavljaju njeni atributi sami po sebi (npr. ukoliko kreira dinamički alocirane resurse). Zadatak destruktora je tada da oslobodi sve dodatne resurse koje je neki primjerak klase zauzeo, prije nego što taj primjerak prestane postojati. Međutim, činjenica da se destruktori uvijek automatski pozivaju nad objektom neposredno prije nego što objekat prestane postojati može dovesti do nepredvidljivog i veoma opasnog ponašanja u slučajevima kada postoji više identičnih primjeraka iste klase. Problemi koji pri tome nastaju rješavaju se uz pomoć *konstruktora kopije* (engl. *copy constructor*) i *prekopljenog operatora dodjele* (engl. *overloaded assignment operator*). Konstruktor kopije i prekopljeni operator dodjele su toliko važni za ispravno funkcioniranje klase tako da se kao pravilo može uzeti da *svaka klasa koja posjeduje destruktur, obavezno mora posjedovati i konstruktor kopije, i prekopljeni operator dodjele*. Stoga iznenađuje da mnoge knjige o jeziku C++ konstruktore kopije spominju više uzgredno (neke površne knjige ih čak ne spominju nikako). Prekopljenom operatuorom dodjele se također ne pridaje dovoljna pažnja: on se obično opisuje u okviru općenite priče o preklapanju operatora, i to više kao kuriozitet nego kao nešto što je vitalno za ispravno funkcioniranje klase.

Konstruktor kopije je specijalan slučaj konstruktora sa jednim parametrom, čiji je formalni parametar *referenca na konstantni objekat klase kojoj pripada*. Njegova uloga je omogućavanje korisniku da upravlja postupkom koji se odvija prilikom kopiranja jednog objekta u drugi. Da bismo uvidjeli potrebu za konstruktorom kopije, razmotrimo šta se tipično dešava kada se jedan objekat kopira u drugi (npr. objekat A u objekat B). Do ovakvog kopiranja može doći u četiri situacije: kada se neki objekat *inicijalizira* drugim objektom iste klase, kada se vrši *dodjeljivanje* nekog objekta drugom objektu iste klase, kada se neki objekat *prenosi po vrijednosti* u neku funkciju (tada se stvarni parametar kopira u odgovarajući formalni parametar), i kada se neki objekat *vraća kao rezultat iz funkcije*. Podrazumijevano ponašanje prilikom kopiranja je da se svi atributi objekta A prosto kopiraju u odgovarajuće attribute objekta B. Međutim, kada god objekti sadrže attribute koji su *pokazivači*, ovo podrazumijevano kopiranje može dovesti do kreiranja tzv. *plitke kopije* koju smo već spominjali, u kojoj nakon kopiranja oba objekta sadrže pokazivače koje pokazuju na isti objekat u memoriji. U kombinaciji sa destruktorma, plitke kopije mogu biti fatalne. Naime, pretpostavimo da objekti A i B sadrže pokazivače na isti dinamički niz u memoriji, i da zbog nekog razloga objekat A prestane postojati. Tom prilikom će se pozvati njegov destruktur, koji vjerovatno uništava taj dinamički niz. Međutim, objekat B (koji i dalje živi) sada sadrži pokazivač na uništeni niz, i dalje posljedice su nepredvidljive!

Razmotrimo jedan konkretan, naizgled bezazlen primjer koji ilustrira ovo o čemu smo govorili. Pretpostavimo da želimo klasu "VektorN" koju smo napisali u prethodnom poglavlju proširiti funkcijom "ZbirVektora" koja vraća kao rezultat zbir dva n -dimenzionalna vektora koji su joj proslijedeni kao parametri. Da bismo ovoj funkciji omogućili pristup privatnim članovima klase "VektorN", deklariraćemo je u interfejsu klase kao funkciju prijatelja klase (parametre "v1" i "v2" *namjerno* prenosimo po vrijednosti, da bismo ukazali na problem o kojem želimo govoriti):

```
class VektorN {
    int br_elemenata;
    double *elementi;
public:
    explicit VektorN(int n) : br_elemenata(n), Elementi(new double[n]) {}
    ~VektorN() { delete[] elementi; }
    void Ispisi() const;
```

```

double Element(int n) const;
double &Element(int n);
friend const VektorN ZbirVektora(VektorN v1, VektorN v2);
};

```

Implementacije metoda “Ispisi” i “Element” su iste kao u prethodnom poglavlju, a implementacija funkcije “ZbirVektora” mogla bi izgledati ovako (primijetimo da je ona morala biti deklarirana kao prijatelj klase, jer u suprotnom ona ne bi mogla nikako saznati dimenziju vektora):

```

const VektorN ZbirVektora(VektorN v1, VektorN v2) {
    if(v1.br_elemlenata != v2.br_elemlenata)
        throw "Vektori koji se sabiraju moraju biti iste dimenzije!\n";
    VektorN v3(v1.br_elemlenata);
    for(int i = 0; i < v1.br_elemlenata; i++)
        v3.elementi[i] = v1.elementi[i] + v2.elementi[i];
    return v3;
}

```

Na prvi pogled je sve u redu, i uz pretpostavku da su “a”, “b” i “c” tri vektora iste dimenzije, moguće je napisati naredbu poput

```
VektorN c = ZbirVektora(a, b);
```

Na žalost, nije sve baš tako lijepo kao što izgleda. Napisana klasa sadrži ozbiljan propust, a gore prikazana naredba može dovesti do teških posljedica koje se mogu očitovati tek nakon izvjesnog vremena. Naime, nakon gornje naredbe sva tri vektora “a”, “b” i “c” sadržavaće viseće pokazivače! Da bismo vidjeli zašto, razmotrimo šta se zaista dešava pri gornjem pozivu. Prvo dolazi do kopiranja stvarnih parametara “a” i “b” u formalne parametre “v1” i “v2”. Pri tome nastaju *plitke kopije*, u kojima objekti “v1” i “a” odnosno “v2” i “b” sadrže pokazivače na *iste dijelove memorije*. Nakon toga se formira lokalni vektor “v3” koji se popunjava zbirom vektora “v1” i “v2”. Ovaj vektor se vraća kao rezultat funkcije pri čemu dolazi do njegovog kopiranja u vektor “c” (stoga će pokazivači u vektorima “v3” i “c” pokazivati na isti dio memorije). Međutim, po završetku funkcije, uništavaju se sva tri lokalna vektora “v1”, “v2” i “v3” (formalni parametri su također lokalni objekti). Prilikom ovog uništavanja dolazi do pozivanja destruktora klase “VektorN” nad svakim od ova tri objekta, koji će uništiti dinamički alocirane nizove na koje pokazuju pokazivači unutar vektora “v1”, “v2” i “v3” (što i jeste osnovni zadatak destruktora). Međutim, vektori “a”, “b” i “c” sadrže pokazivače koji pokazuju na iste dijelove memorije, tako da će nakon izvršenja ove naredbe sva tri vektora sadržavati pokazivače koji pokazuju na upravo oslobođene dijelove memorije. Stoga, njihovo dalje korištenje može imati kobne posljedice!

Šta da se radi? Kopiranje vektora “a” i “b” u “v1” i “v2” bismo mogli lako izbjegći ukoliko bismo parametre prenosili po referenci na konstantne objekte (tj. ako bismo deklarirali da su “v1” i “v2” reference na konstantne objekte tipa “VektorN”). Kao što znamo, ovo je svakako i preporučeni način prenosa primjeraka klase u funkcije. Međutim, kopiranje rezultata koji se nalazi u vektoru “v3” nije moguće izbjegći čak ni vraćanjem reference na njega kao rezultata (ne smijemo vratiti referencu na objekat koji prestaje postojati, jer tako dobijamo viseću referencu). Očigledno, svi problemi nastaju zbog plitkog kopiranja. Pravo rješenje je promijeniti mehanizam kopiranja, tako da kopija objekta dobija ne samo kopiju pokazivača nego i *kopiju odgovarajućeg dinamičkog niza pridruženog pokazivaču*. Na taj način će destrukturator kopiranog objekta uništiti “svoj” a ne i “tudi” dinamički niz. Sljedeća slika ilustrira razliku između plitke i potpune (duboke) kopije:

Plitka kopija:

a

Duboka kopija:

a

v1

v1

Neko se može zapitati zbog čega jezik C++ prilikom kopiranja objekata automatski ne vrši duboku nego plitku kopiju. Odgovor je veoma jednostavan: kompjajler *ne može znati* na šta pokazuju pokazivači koji se nalaze unutar objekta, jer to može zavisiti od toga šta je programer radio sa tim pokazivačima u čitavom ostatku programa (razumije se da kompjajler svakako može znati *koju adresu sadrži pokazivač*, ali ne može znati *šta logički predstavlja objekat koji se tamo nalazi*). Zbog toga su uvedeni konstruktori kopije koji omogućavaju projektantu klase da *sâm definira postupak kako se objekti te klase trebaju kopirati*, odnosno da po potrebi implementira duboko kopiranje objekata na način kako mu to odgovara. Drugim riječima, ukoliko klasa ima konstruktor kopije, tada se objekti te klase ne kopiraju uobičajenim postupkom (samo kopiranjem atributa) nego na način kako je to definirano u konstruktoru kopije. Zapravo, svaka klasa uvijek posjeduje konstruktor kopije, čak i ukoliko ga nismo sami napisali. U tom slučaju, kompjajler automatski generira *podrazumijevani konstruktor kopije*, koji prosto obavlja kopiranje svih atributa klase jedan po jedan (što, kao što smo vidjeli, tipično dovodi do plitkih kopija kada god neki od atributa klase predstavlja pokazivač na dinamički alocirane resurse).

Na osnovu prethodnog izlaganja, slijedi da je rješenje opisanog problema sa klasom "VektorN" definiranje vlastitog konstruktora kopije (umjesto automatski generiranog podrazumijevanog konstruktora kopije), koji će kreirati potpunu (duboku) kopiju objekta. Kao što je već rečeno, konstruktor kopije ima jedan parametar koji predstavlja konstantnu referencu na objekat koji se kopira (koji je naravno ponovo tipa "VektorN"). Nova deklaracija klase "VektorN" izgledaće ovako:

```
class VektorN {
    int br_elemenata;
    double *elementi;
public:
    explicit VektorN(int n) : br_elemenata(n), elementi(new double[n]) {}
    VektorN(const VektorN &v);
    ~VektorN() { delete[] elementi; }
    void Ispisi() const;
    double Element(int n) const;
    double &Element(int n);
    friend const VektorN ZbirVektora(VektorN v1, VektorN v2);
};
```

S obzirom da konstruktor kopije nije posve kratak, njegovu implementaciju ćemo izvesti izvan deklaracije klase. Razmotrimo šta on zapravo treba da obavi. Atribut "br_elemenata" svakako treba kopirati, ali atribut-pokazivač "elementi" ne treba prosto da se kopira. Umjesto toga, treba stvoriti novi dinamički niz, i kopirati izvorni dinamički niz u novostvoreni niz element po element. Stoga bi implementacija konstruktora kopije za klasu "VektorN" mogla izgledati ovako:

```
VektorN::VektorN(const VektorN &v) {
    br_elemenata = v.br_elemenata;
    elementi = new double[v.br_elemenata];
    for(int i = 0; i < v.br_elemenata; i++)
        elementi[i] = v.elementi[i];
}
```

Napomenimo da smo konstruktor kopije za klasu “VektorN” mogli implementirati i na sljedeći način (ova implementacija je izvedena u skladu sa pravilom da sve što se može inicijalizirati u konstruktorskoj inicijalizacionoj listi treba inicijalizirati u konstruktorskoj inicijalizacionoj listi radi bolje efikasnosti):

```
VektorN::VektorN(const VektorN &v) : br_elemlenata(v.br_elemlenata),  
    elementi(new double[v.br_elemlenata]) {  
    for(int i = 0; i < v.br_elemlenata; i++)  
        elementi[i] = v.elementi[i];  
}
```

Umjesto kopiranja element po element, možemo koristiti i funkciju “copy” iz biblioteke “algorithm”, što može biti efikasnije, kao u sljedećoj implementaciji (za tu svrhu, moramo u program uključiti zaglavlj biblioteke “algorithm”):

```
VektorN::VektorN(const VektorN &v) : br_elemlenata(v.br_elemlenata),  
    elementi(new double[v.br_elemlenata]) {  
    copy(v.elementi, v.elementi + v.br_elemlenata, elementi);  
}
```

Ovako definirani konstruktori kopije obezbjeđuju duboko kopiranje, čime ćemo biti sigurni da će svi različiti objekti imati svoje neovisne primjerke dinamičkih nizova, odnosno da se neće “jedan drugom petljati u posao”. Ipak, konstruktor kopije se poziva samo u tri od četiri situacije u kojima bi mogle nastati plitke kopije. Naime, konstruktor kopije se poziva pri *inicijalizaciji* novostvorenog objekta drugim objektom iste klase, pri *prenosu po vrijednosti* objekata u funkcije, i pri *vraćanju objekata kao rezultata iz funkcije*. Prilikom *dodjeljivanja* nekog objekta drugom objektu koji je ranije stvoren, *konstruktor kopije se ne poziva!* To znači da ukoliko bismo izvršili naredbe poput

```
b = a;  
c = ZbirVektora(a, b);
```

pri čemu sva tri vektora “a”, “b” i “c” postoje *od ranije*, u oba slučaja će biti izvršena plitka kopija, bez obzira što smo definirali konstruktor kopije! Razlog zašto se konstruktor kopije ne poziva i u ovom slučaju je što se ovaj slučaj razlikuje od prva tri po tome što objekat kojem se vrši dodjela već od ranije postoji, pa konstruktor kopije ne može da zna šta treba da radi sa prethodnim sadržajem objekta (u sva tri preostala slučaja radi se o stvaranju novih objekata, pa ovakvih dilema nema). Prepostavimo, na primjer, da su vektori “a” i “b” prethodno deklarirani deklaracijom

```
VektorN a(7), b(5);
```

i da nakon toga izvršimo dodjelu “b = a”. Prije izvršene dodjele, oba vektora “a” i “b” sadrže pokazivače koji pokazuju na dva različita dinamički alocirana niza različite veličine. Nakon obavljenog plitkog kopiranja, oba pokazivača pokazivaće na dinamički niz dodijeljen prvom objektu, dok na dinamički niz koji je bio pridružen objektu “b” više ne pokazuje niko. Dakle, u ovom slučaju dolazi i do curenja memorije, s obzirom da taj dinamički niz više ne može obrisati niko. Ova situacija prikazana je na sljedećoj slici:

Prije kopiranja:

a

Poslije kopiranja:

a

Dalje je važno uočiti da čak ni duboko kopiranje kakvo je implementirano u konstruktoru kopije ne bi riješilo problem. Pri takvom kopiranju bila bi izvršena alokacija novog dinamičkog niza u koji bi bio iskopiran niz označen na slici sa “*Dinamički niz 1*”, ali niz “*Dinamički niz 2*” ne bi bio uništen (niti bi ga iko kasnije mogao uništiti). Očito se prilikom dodjeljivanja nekog objekta nekom drugom objektu koji je već postojao od ranije, trebaju poduzeti drugačije akcije nego što su predviđene konstruktorom kopije. To je razlog zbog kojeg se konstruktor kopije i ne poziva u ovom slučaju.

Opisani problem bi se mogao riješiti kada bi se prilikom dodjele poput “ $b = a$ ” prvo izvršio *destruktur* nad objektom “ b ”, a zatim iskoristio *konstruktor kopije* za kopiranje objekta “ a ” u objekat “ b ”. Međutim, tvorci jezika C++ namjerno nisu željeli automatski podržati ovakvo ponašanje, jer ono uglavnom nije i najbolji način da se ostvari ispravna funkcionalnost. Na primjer, u slučaju da su vektori “ a ” i “ b ” iste dimenzije, najprirodnije ponašanje prilikom dodjele “ $b = a$ ” je prosto samo *iskopirati* sve elemente dinamičkog niza pridruženog objektu “ a ” u dinamički niz dodijeljen objektu “ b ” (koji već postoji odranije). Nikakve dodatne alokacije niti dealokacije memorije nisu potrebne!

Za rješavanje opisanog problema, jezik C++ dopušta da sami definiramo kako će se interpretirati izraz oblika “ $b = a$ ” u slučaju kada objekat “ b ” odranije postoji. Kao što znamo, podrazumijevano ponašanje ovog izraza je prosto kopiranje svih atributa objekta “ a ” u odgovarajuće attribute objekta “ b ”, jedan po jedan (isto kao i kod podrazumijevanog konstruktora kopije). Projektant klase može definirati drugačije ponašanje *preklapanjem operatora dodjele*. Mada je preklapanje operatora dodjele specijalan slučaj općeg preklapanja operatora o kojem ćemo govoriti u sljedećem poglavlju, očekivano ponašanje operatora dodjele je veoma tjesno vezano za ponašanje konstruktora kopije, tako da je o njegovom preklapanju prirodno govoriti zajedno sa opisom konstruktora kopije.

Preklapanje operatora dodjele vrši se definiranjem specijalne funkcije članice sa nazivom “**operator =**” (razmak između ključne riječi “**operator**” i znaka “=” nije obavezan). Mada ćemo u sljedećem poglavlju vidjeti da ova funkcija principijelno može da prima parametar bilo kojeg tipa i da vraća rezultat bilo kojeg tipa, za ostvarenje one funkcionalnosti koju ovdje želimo postići njen formalni parametar treba biti referenca na konstantni objekat pripadne klase (dakle, isto kao i kod konstruktora kopije), dok tip rezultata treba također biti referenca na objekat pripadne klase. Drugim riječima, tražena funkcija članica koja rješava problem dodjele za klasu “**VektorN**” treba imati sljedeći prototip:

```
VektorN &operator =(const VektorN &v);
```

Nakon deklaracije funkcije članice koja realizira preklopjeni operator dodjele, sve dodjele poput “ $a = b$ ” interpretiraće se kao izraz oblika “ $a.\text{operator} = (b)$ ”. Sada bi kompletna deklaracija klase “**VektorN**” mogla izgledati ovako:

```
class VektorN {
    int br_elemenata;
    double *elementi;
public:
    explicit VektorN(int n) : br_elemenata(n), elementi(new double[n]) {}
    VektorN(const VektorN &v);
    VektorN &operator =(const VektorN &v);
    ~VektorN() { delete[] elementi; }
    void Ispisi() const;
```

```

double Element(int n) const;
double &Element(int n);
friend const VektorN ZbirVektora(VektorN v1, VektorN v2);
};

```

Naravno, deklariranu funkciju članicu treba i implementirati. Pri tome, veoma je važno uočiti razliku između neophodnog ponašanja konstruktora kopije i preklopljenog operatora dodjele. Operator dodjele se nikad ne poziva pri *inicijalizaciji* objekata, nego samo kada se dodjela vrši nad objektom *koji od ranije postoji*. Za razliku od konstruktora kopije koji uvijek treba izvršiti dinamičku alokaciju memorije za potrebe objekta u koji se vrši kopiranje (s obzirom da ona nije prethodno alocirana), operator dodjele se primjenjuje nad objektom koji od ranije postoji, tako da je za njegove potrebe već alocirana memorija. Stoga operator dodjele treba da provjeri da li je alocirana količina memorije dovoljna da prihvati podatke koje treba kopirati. Ukoliko jeste, dovoljno je samo izvršiti kopiranje. Međutim ukoliko nije, potrebno je dealocirati memoriju koja je zauzeta i izvršiti ponovnu alokaciju neophodne količine memorije, pa tek tada obavljati kopiranje. Uz ovakve preporuke, moguća implementacija funkcije članice koja realizira preklopljeni operator dodjele mogla bi izgledati recimo ovako:

```

VektorN &VektorN::operator =(const VektorN &v) {
    if(br_elemenata < v.br_elemenata) {
        delete[] elementi;
        elementi = new double[v.br_elemenata];
    }
    br_elemenata = v.br_elemenata;
    copy(v.elementi, v.elementi + v.br_elemenata, elementi);
    return *this;
}

```

U prikazanoj implementaciji, realokacija (tj. dealokacija i ponovna alokacija) memorije izvodi se samo u slučaju kada se vektoru manje dimenzionalnosti dodjeljuje vektor veće dimenzionalnosti. Na primjer, ukoliko je od ranije “*a*” bio sedmodimenzionalni a “*b*” petodimenzionalni vektor, dodjela poput “*b = a*” mora izvršiti realokaciju da bi “*b*” bio spreman da prihvati sve komponente vektora “*a*”. Međutim, u slučaju da je dimenzionalnost vektora “*b*” veća od dimenzionalnosti vektora “*a*”, dinamički niz vezan za vektor “*b*” već sadrži dovoljno prostora da prihvati sve komponente vektora “*a*”, tako da realokacija nije neophodna. U svakom slučaju, ovo je mnogo efikasnije nego kad bi se realokacija vršila uvijek. Naravno, u slučaju da je dimenzionalnost vektora “*a*” *mnogo manja* od dimenzionalnosti vektora “*b*”, također je mudro izvršiti realokaciju, da se bespotrebno ne zauzima mnogo veći blok memorije nego što je zaista potrebno (ova ideja nije ugrađena u gore prikazanu implementaciju). Upravo u tome i jeste poenta: preklapanje operatora dodjele omogućava projektantu klase da samostalno specificira šta će se tačno dešavati prilikom izvršavanja dodjele, i na koji način. Recimo još i to da funkcije članice koje realiziraju preklapanje operatora dodjele gotovo po pravilu vraćaju referencu na objekat nad kojim se sama dodjela vrši (što se postiže dereferenciranjem pokazivača “**this**”). Vidjećemo kasnije da takva konvencija omogućava ulančavanje operatora dodjele.

Neki programeri prilikom realizacije preklopljenog operatora dodjele uvijek brišu memoriju koju je alocirao objekat kojem se vrši dodjela, a nakon toga vrše alokaciju potrebne količine memorije, bez ikakve prethodne provjere (ovakve izvedbe su česte i u mnogim udžbenicima za C++). Ovim se oponaša efekat kao da je nad objektom sa lijeve strane operatora dodjele izvršen destruktor, a zatim izvršen konstruktor kopije. Međutim, ovo nije dobra praksa jer se na taj način bespotrebno vrši dealokacija i ponovna alokacija memorije čak i u slučajevima kad je ona potpuno nepotrebna (recimo, realokacija je bez ikakve sumnje posve neopravdvana u slučaju da su izvorni i odredišni objekat rezervirali potpuno istu količinu memorije). Ipak, radi ilustracije nekih specifičnosti, pokažimo i kako bi izgledala takva implementacija

(koja je neznatno jednostavnija od prethodne implementacije):

```
VektorN &VektorN::operator =(const VektorN &v) {
    if(&v == this) return *this;
    delete[] elementi;
    elementi = new double[v.br_elemenata];
    br_elemenata = v.br_elemenata;
    copy(v.elementi, v.elementi + v.br_elemenata, elementi);
    return *this;
}
```

Ovdje treba posebnu pažnju obratiti na prvi red ove funkcije, koji glasi

```
if(&v == this) return *this;
```

Ovom naredbom se ispituje da li su izvorni i odredišni objekat (vektor) identični, i ukoliko jesu, ne radi se ništa. Svrha ove naredbe je da se izbjegne opasnost od tzv. *destruktivne samododjele* (engl. *destructive self-assignment*). Pod samododjelom se podrazumijeva naredba koja je logički ekvivalentna naredbi “*a = a*”. Naime, u slučaju da dođe do samododjele, u slučaju da nismo preduzeli posebne mjere opreza, objekat sa lijeve strane bio bi uništen, ali bi samim tim bio uništen i objekat sa desne strane, pošto se radi o istom objektu! Razumije se da niko neće eksplicitno pisati naredbe poput “*a = a*”, međutim skrivene situacije koje su logički ekvivalentne ovakvoj naredbi mogu se pojaviti. Na primjer, ukoliko su “*x*” i “*y*” reference koje su vezane na isti objekat “*a*”, tada je dodjela “*x = y*” suštinski ekvivalentna dodjeli “*a = a*”, a situacije da su dvije reference vezane na isti objekat uopće nije rijetka, naročito u programima gdje se mnogo koristi prenos parametara po referenci. Zbog toga, funkcija koja realizira preklopljeni operator dodjele nikada ne smije brisati odredišni objekat bez prethodne garancije da on nije ekvivalentan izvornom objektu. Primijetimo da je u ranijoj implementaciji ova garancija implicitno ostvarena. Naime, mi smo vršili destrukciju (i ponovnu konstrukciju) odredišnog objekta samo ukoliko ustanovimo da je on *manjeg kapaciteta* od kapaciteta izvornog objekta, što u slučaju samododjele sigurno neće biti slučaj.

Rezimirajmo sada kada je neophodno da neka klasa ima konstruktor kopije i preklopljeni operator dodjele. Klase koje ne koriste nikakve druge dodatne resurse osim svojih vlastitih atributa, ne trebaju imati niti destruktor, niti konstruktor kopije, niti preklopljeni operator dodjele. S druge strane, svaka klasa čije metode (a pogotovo konstruktor) vrše dinamičku alokaciju memorije ili drugih računarskih resursa, treba obavezno imati destruktor. Dalje, *svaka klasa koja ima destruktor, po pravilu bi morala imati i konstruktor kopije i preklopljeni operator dodjele*. Načelno, ukoliko smo posve sigurni da objekti neke klase neće biti prenošeni po vrijednosti kao parametri u funkcije, neće biti vraćani kao rezultati iz funkcije, i neće biti korišteni za inicijalizaciju drugih objekata iste klase, konstruktor kopije možemo izbjegići (takav slučaj smo imali kod klase “Razred” u prethodnom poglavlju). Preklopljeni operator dodjele također bismo mogli izbjegći ukoliko smo sigurni da se nikada objekti te klase neće dodjeljivati jedan drugom. Međutim, to nije osobito dobra ideja, s obzirom da ukoliko pišemo klasu za kasnije korištenje koja će se moći upotrebljavati i u drugim programima, moramo predvidjeti i konstruktor kopije i preklopljeni operator dodjele, jer ne možemo unaprijed znati šta će korisnik klase sa njom raditi. Ukoliko baš ne želimo pisati konstruktor kopije i preklopljeni operator dodjele, tada kopiranje objekata te klase i njihovo međusobno dodjeljivanje trebamo potpuno *zabraniti*. To se postiže formalnim deklariranjem prototipova konstruktora kopije i funkcije članice za preklopljeni operator dodjele unutar *privatne sekcije klase*, ali *bez njihovog implementiranja* (u slučaju da ih implementiramo, kopiranje i dodjela će ipak biti mogući, ali samo iz funkcija članica klase i prijateljskih funkcija, s obzirom da su deklarirani u privatnoj sekciji). Tako bi, iz razloga sigurnosti, klasu “Razred” iz prethodnog poglavlja trebalo deklarirati na sljedeći način:

```
class Razred {
```

```

const int kapacitet;
int broj_evidentiranih;
Ucenik **ucenici;
static bool BoljiProsjek(const Ucenik *u1, const Ucenik *u2) {
    return u1->Prosjek() > u2->prosjek();
}
Razred(const Razred &); // Neimplementirano...
Razred &operator =(const Razred &); // Neimplementirano...
public:
    ... // Javna sekcija klase ostaje ista kao i ranije...
};

```

Projektant klase se može odlučiti za zabranu kopiranja i dodjele ukoliko zaključi da bi kopiranje odnosno dodjela mogli biti isuviše neefikasni. Na taj način, projektant klase može spriječiti korisnika klase da prenosi primjerke te klase po vrijednosti u funkcije, i da vrši inicijalizaciju ili dodjelu pomoću primjeraka te klase. Na žalost, time se onemogućava i vraćanje primjeraka te klase kao rezultata iz funkcije. Srećom, postoje brojne situacije u kojima to nije neophodno.

Možemo zaključiti da ukoliko klasa posjeduje destruktur, jedina ispravna rješenja su ili da deklariramo i implementiramo kako konstruktor kopije tako i preklopljeni operator dodjele, ili da zabranimo kopiranje i međusobno dodjeljivanje objekata te klase na upravo opisani način. Uostalom, postoje samo dva razloga zbog kojih bi projektant klase izbjegao da primijeni jedan od dva opisana pristupa: *nezmanje* i *lijenost*. Protiv prvog razloga relativno se lako boriti, jer što se ne zna, uvijek se da naučiti. Protiv drugog razloga (lijenosti) znatno se teže boriti.

Kao nešto složeniju ilustraciju svih do sada izloženih koncepata, daćemo prikaz programa koji definira generičku klasu "Matrica", koja veoma efektno enkapsulira u sebe tehnike za dinamičko upravljanje memorijom. Detaljan opis rada programa slijedi odmah nakon njegovog prikaza:

```

#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;
template <typename Tip>
class Matrica {
    int br_redova, br_kolona;
    Tip **elementi;
    char ime_matrice[10];
    void AlocirajMemoriju(int br_redova, int br_kolona);
    void DealocirajMemoriju(int br_redova);
public:
    Matrica(int br_redova, int br_kolona, const char ime[] = "");
    Matrica(const Matrica &m);
    ~Matrica() { DealokacirajMemoriju(br_redova); }
    Matrica &operator =(const Matrica &m);
    void Unesi();
    void Ispisi(int sirina_ispisa) const;
    friend const Matrica ZbirMatrica(const Matrica &m1,
        const Matrica &m2);
};
template <typename Tip>
void Matrica<Tip>::AlocirajMemoriju(int br_redova, int br_kolona) {
    elementi = new Tip*[br_redova];
    for(int i = 0; i < br_redova; i++) elementi[i] = 0;
}

```

```

    try {
        for(int i = 0; i < br_redova; i++)
            elementi[i] = new Tip[br_kolona];
    }
    catch(...) {
        DealocirajMemoriju(br_redova);
        throw;
    }
}

template <typename Tip>
void Matrica<Tip>::DeallocirajMemoriju(int br_redova) {
    for(int i = 0; i < br_redova; i++) delete[] elementi[i];
    delete[] elementi;
}

template <typename Tip>
Matrica<Tip>::Matrica(int br_redova, int br_kolona, const char ime[]) :
    br_redova(br_redova), br_kolona(br_kolona) {
    if(strlen(ime) > 9) throw "Predugačko ime!\n";
    strcpy(ime_matrice, ime);
    AlocirajMemoriju(br_redova, br_kolona);
}

template <typename Tip>
Matrica<Tip>::Matrica(const Matrica<Tip> &m) : br_redova(m.br_redova),
    br_kolona(m.br_kolona) {
    strcpy(ime_matrice, m.ime_matrice);
    AlocirajMemoriju(br_redova, br_kolona);
    for(int i = 0; i < br_redova; i++)
        for(int j = 0; j < br_kolona; j++)
            elementi[i][j] = m.elementi[i][j];
}

template <typename Tip>
Matrica<Tip> &Matrica<Tip>::operator =(const Matrica<Tip> &m) {
    if(br_redova < m.br_redova || br_kolona < m.br_kolona) {
        DealocirajMemoriju(br_redova);
        AlocirajMemoriju(m.br_redova, m.br_kolona);
    }
    strcpy(ime_matrice, m.ime_matrice);
    br_redova = m.br_redova; br_kolona = m.br_kolona;
    for(int i = 0; i < br_redova; i++)
        for(int j = 0; j < br_kolona; j++)
            elementi[i][j] = m.elementi[i][j];
    return *this;
}

template <typename Tip>
void Matrica<Tip>::Unesi() {
    for(int i = 0; i < br_redova; i++)
        for(int j = 0; j < br_kolona; j++) {
            cout << ime_matrice << "(" << i + 1 << "," << j + 1 << ") = ";
            cin >> elementi[i][j];
        }
}

template <typename Tip>
void Matrica<Tip>::Ispisi(int sirina_ispisa) const {
    for(int i = 0; i < br_redova; i++) {
        for(int j = 0; j < br_kolona; j++)

```

```

        cout << setw(sirina_ispisa) << elementi[i][j];
        cout << endl;
    }
}

template <typename Tip>
const Matrica<Tip> ZbirMatrica(const Matrica<Tip> &m1,
const Matrica<Tip> &m2) {
    if(m1.br_redova != m2.br_redova || m2.br_redova != m2.br_redova)
        throw "Matrice nemaju jednake dimenzije!\n";
    Matrica<Tip> m3(m1.br_redova, m1.br_kolona);
    for(int i = 0; i < m1.br_redova; i++)
        for(int j = 0; j < m1.br_kolona; j++)
            m3.elementi[i][j] = m1.elementi[i][j] + m2.elementi[i][j];
    return m3;
}

int main() {
    int m, n;
    cout << "Unesi broj redova i kolona za matrice:\n";
    cin >> m >> n;
    try {
        Matrica<double> a(m, n, "A"), b(m, n, "B");
        cout << "Unesi matricu A:\n";
        a.Unesi();
        cout << "Unesi matricu B:\n";
        b.Unesi();
        cout << "Zbir ove dvije matrice je:\n";
        ZbirMatrica(a, b).Ispisi(7);
    }
    catch(...) {
        cout << "Nema dovoljno memorije!\n";
    }
}

```

Na prvom mjestu, potrebno je obratiti pažnju na činjenicu da je "Matrica" generička klasa. Mada do sada nismo eksplicitno govorili da klase također mogu biti generičke, to je u skladu sa očekivanjima, s obzirom da strukture mogu biti generičke, a klase su samo poopćenje struktura. Slično kao kod generičkih struktura, samo ime generičke klase (npr. "Matrica") nije tip, nego tip dobijamo tek nakon specifikacije parametara šablona (stoga npr. "Matrica<double>" jeste tip). Unutar same deklaracije klase, parametar šablona *ne treba navoditi*, s obzirom da je već sama deklaracija klase uklopljena u šablon (tako da se parametar šablona podrazumijeva). S druge strane, izvan deklaracije klase, parametar šablona se *ne podrazumijeva* (s obzirom da je područje primjene šablona unutar kojeg je deklarirana generička klasa ograničeno na samu deklaraciju). Stoga se bilo gdje izvan deklaracije same klase gdje se očekuje ime tipa, ime "Matrica" ne smije koristiti samostalno kao ime tipa, već se uvijek mora navesti parametar šablona. Kao parametar šablona možemo iskoristiti neki konkretni tip (ukoliko se želimo vezati za konkretni tip), ali možemo i ponovo iskoristiti neki neodređeni tip, koji je parametar nekog novog šablona. Tako je urađeno i u prikazanom primjeru, u kojem su sve metode klase (osim destruktora) implementirane izvan deklaracije klase. Radi univerzalnosti, svaka od njih je uklopljena u šablon, tako da po formi sve implementacije metoda podsjećaju na implementacije običnih generičkih funkcija.

Pogledajmo sada kako je implementirana ova klasa. Vidimo da pored konstruktora, destruktora i preklopljeno operatora dodjele, interfejs ove klase sadrži metode "Unesi" i "Ispisi", koje su dovoljno jednostavne, tako da se o njima nema mnogo toga reći. Interesantno je jedino istaći da metoda "Ispisi"

zahtijeva kao parametar željenu širinu ispisa koji će zauzeti svaki element matrice. Pored ove dvije metode definirana je i prijateljska funkcija “*ZbirMatrica*” koja vrši sabiranje dvije matrice, koja je također jasna sama po sebi. Razumije se da bi stvarna klasa “*Matrica*” koja bi bila upotrebljiva u više različitih programa morala imati znatno bogatiji interfejs (na primjer, ne postoji ni jedna metoda koja omogućava pristup individualnim elementima matrice). Međutim, ovdje nismo željeli da program ispadne predugačak, jer je osnovni cilj programa da uvidimo kako su implementirani *konstruktori*, *destruktori* i *preklopljeni operator dodjele* generičke klase “*Matrica*”.

Konstruktor klase “*Matrica*” ima tri parametra. Prva dva parametra predstavljaju željene dimenzije matrice, dok treći parametar, koji se može izostaviti, predstavlja ime matrice dato kao nul-terminalirani string, koji se koristi pri ispisu prilikom unosa elemenata matrice (u slučaju izostavljanja podrazumjeva se prazno ime). Ovaj konstruktor, pored inicijalizacije atributa “*br_redova*”, “*br_kolona*” i “*ime_matrice*”, vrši i dinamičku alokaciju matrice kojoj se pristupa preko dvojnog atributa-pokazivača “*elementi*” korištenjem tehnika opisanih u ranijim poglavlјima. Međutim, kako je sama dinamička alokacija dvodimenzionalnih nizova nešto masivniji postupak (pogotovo ukoliko se moramo brinuti o eventualnoj nestašici memorije), a isti postupak će biti potreban u konstruktoru kopije i preklopljenom operatoru dodjele, alokaciju smo povjerili privatnoj metodi “*AlocirajMemoriju*”, koja će se pozivati kako iz običnog konstruktora, tako i iz konstruktora kopije i funkcije članice koja realizira preklopljeni operator dodjele. Na taj način znatno skraćujemo program. Metodu “*AlocirajMemoriju*” smo učinili privatnom, jer korisnik klase nikada neće imati potrebu da ovu metodu poziva eksplicitno. Ovo je lijepa ilustracija kada može biti od koristi da se neka metoda deklarira kao privatna.

Sama metoda “*AlocirajMemoriju*” vrši postupak dinamičke alokacije dvodimenzionalnog niza na sličan način kao u funkciji “*StvoriMatricu*” koju smo demonstrirali u poglavlju koje govori o strukturama. Pri tome je bitno naglasiti da se ova metoda brine da počisti iza sebe sve izvršene alokacije u slučaju da alokacija memorije ne uspije do kraja. Naime, eventualni izuzetak koji baci ova funkcija biće bačen i iz konstruktora, jer unutar konstruktora ne vršimo hvatanje izuzetaka. U slučaju da se izuzetak baci iz konstruktora, smatra se da objekat nije ni kreiran, pa neće biti pozvan ni destruktur. Stoga, ukoliko funkcija “*AlocirajMemoriju*” ne “počisti svoje smeće” iza sebe u slučaju neuspješne alokacije, niko ga drugi neće počistiti (niti će ga moći počistiti), što naravno vodi ka curenju memorije. Pomenuto “čišćenje” povjerenog je funkciji “*DealocirajMemoriju*”, koja se poziva iz konstruktora (u slučaju greške), iz destruktora (destruktur zapravo ne radi ništa drugo osim poziva ove funkcije), i u slučaju potrebe, iz funkcije koja realizira preklopljeni operator dodjele.

Konstruktor kopije klase “*Matrica*” pored kopiranja atributa “*br_redova*”, “*br_kolona*” i “*ime_matrice*” obavlja novu dinamičku alokaciju (pozivom metode “*AlocirajMemoriju*”) nakon čega kopira element po element sve elemente izvorne dinamičke matrice u novokreirani prostor. Ovo kopiranje se također moglo optimizirati korištenjem funkcije “*copy*” iz biblioteke “*algorithm*” na sljedeći način, koji nije baš očigledan na prvi pogled (ovdje se petljom kopiraju individualni redovi matrice, a funkcija “*copy*” se koristi za kopiranje jednog reda):

```
for(int i = 0; i < br_redova; i++)
    copy(m.elementi[i], m.elementi[i] + br_kolona, elementi[i]);
```

Funkcija koja realizira preklopljeni operator dodjele također obavlja duboko kopiranje, vodeći pri tome računa da ne obavlja realokaciju memorije u slučaju kada to nije neophodno (npr. pri dodjeli neke matrice drugoj matrici koja je prethodno bila istog ili većeg formata). Ostaje još glavni program (odnosno funkcija “*main*”) koji je dovoljno jednostavan da ne zahtijeva nikakva posebna objašnjenja.

Treba napomenuti da definiranje konstruktora kopije i preklopljenog operatara dodjele koji realiziraju duboko kopiranje nije niti jedini niti najefikasniji način za rješavanje problema interakcije između

destruktora i plitkih kopija (tj. neželjenog brisanja blokova memorije na koje istovremeno pokazuje više pokazivača, a koji nastaju uslijed plitkih kopija). Naime, nedostatak ovog rješenja je prečesto kopiranje ponekad i prilično velikih blokova memorije. Alternativno rješenje je uvesti neki brojač koji će *brojati koliko primjeraka neke klase sadrže pokazivače koji pokazuju na isti blok memorije*. Konstruktor će ovaj brojač postaviti na 1, a konstruktor kopije će i dalje obavljati kopiranje samo atributa klase (tj. plitko kopiranje), ali će pri tome uvećavati pomenuti brojač za 1. Destruktor će ovaj brojač smanjivati za 1, ali će vršiti brisanje memorije samo u slučaju da brojač dostigne nulu, što znači da više nema ni jednog objekta koji sadrži pokazivač na zauzeti dio memorije. Ova strategija je očigledno mnogo efikasnija od bezuvjetnog kopiranja, a naziva se *brojanje referenciranja* (engl. *reference counting*). Ostaje pitanje *gdje deklarirati ovaj brojač*. On ne može biti obični atribut klase, s obzirom da njega trebaju zajednički dijeliti sve kopije nekog objekta. On također ne može biti statički atribut klase, s obzirom da on treba da bude zajednički samo za one primjerke klase koji su kopija jedan drugog, ali ne i za ostale primjerke klase. Jedino rješenje je da brojač bude *dinamička promjenljiva* (koja se, prema tome, nalazi *izvan same klase*) a da klasa kao atribut sadrži pokazivač na nju, preko kojeg joj se može pristupiti. Njeno kreiranje će izvršiti konstruktor, a uništavanje destruktur, onog trenutka kada više nije potrebna.

Ilustrirajmo opisanu tehniku na primjeru klase “*VektorN*” koju smo prethodno razvili. Za tu svrhu, izmijenićemo njenu deklaraciju tako da izgleda ovako:

```
class VektorN {
    int br_elemenata;
    double *elementi;
    int *pnb;
public:
    explicit VektorN(int n) : br_elemenata(n), elementi(new double[n]),
        pnb(new int(1)) {}
    VektorN(const VektorN &v) : br_elemenata(v.br_elemenata),
        elementi(v.elementi), pnb(v.pnb) { (*pnb)++; }
    VektorN &operator =(const VektorN &v);
    ~VektorN();
    void Ispisi() const;
    double Element(int n) const;
    double &Element(int n);
    friend const VektorN ZbirVektora(VektorN v1, VektorN v2);
};
```

Unutar klase smo uveli novi atribut “*pnb*” (skraćenica od “pokazivač na brojač”), koji predstavlja pokazivač na brojač identičnih kopija. Konstruktor, pored uobičajenih zadataka, vrši kreiranje dinamičke promjenljive koja predstavlja brojač, inicijalizira ga na jedinicu (pomoću konstrukcije “*new int(1)*”), i dodjeljuje njegovu adresu pokazivaču “*pnb*”. Konstruktor kopije prosto kopira sve atribute klase, i povećava brojač kopija za jedinicu. Destruktor je postao nešto složeniji, tako da ćemo ga implementirati izvan klase. U skladu sa provedenim razmatranjem, njegova implementacija bi mogla izgledati ovako:

```
VektorN::~VektorN() {
    if(--(*pnb) == 0) {
        delete[] elementi; delete pnb;
    }
}
```

Ostaje još implementacija prekloppljenog operatora dodjele, koji je nešto složeniji. On treba umanjiti za 1 brojač pridružen objektu sa lijeve strane operatora dodjele, i izvršiti dealokaciju memorije u slučaju da je brojač dostigao nulu. Brojač kopija pridružen objektu sa desne strane treba uvećati za 1, i nakon toga izvršiti plitko kopiranje. Ukoliko prvo izvršimo uvećanje ovog brojača, pa tek onda umanjenje brojača

pridruženog objektu sa lijeve strane (uz eventualnu dealokaciju memorije), izbjegći ćemo i probleme uslijed eventualne samododjele, tako da taj slučaj ne moramo posebno razmatrati:

```
VektorN &VektorN::operator =(const VektorN &v) {
    (*v.pnb)++;
    if(--(*pnb) == 0) delete[] elementi;
    br_elemenata = v.br_elemenata; elementi = v.elementi; pnb = v.pnb;
    return *this;
}
```

Radi boljeg razumijevanja, istu tehniku ćemo ilustrirati i na primjeru generičke klase "Matrica". Nakon dodavanja atributa "pnb" u njenu privatnu sekciju (na identičan način kao i u slučaju klase "VektorN"), nove implementacije konstruktora, konstruktora kopije, destruktora i preklopjenog operatora dodjele mogće bi izgledati recimo ovako:

```
template <typename Tip>
Matrica<Tip>::Matrica(int br_redova, int br_kolona, const char ime[]) :
    br_redova(br_redova), br_kolona(br_kolona), pnb(new int(1)) {
    if(strlen(ime) > 9) throw "Predugačko ime!\n";
    strcpy(ime_matrice, ime);
    AlocirajMemoriju(br_redova, br_kolona);
}

template <typename Tip>
Matrica<Tip>::Matrica(const Matrica<Tip> &m) : br_redova(m.br_redova),
    br_kolona(m.br_kolona), elementi(m.elementi), pnb(m.pnb) {
    strcpy(ime_matrice, m.ime_matrice);
    (*pnb)++;
}

template <typename Tip>
Matrica<Tip>::~Matrica() {
    if(--(*pnb) == 0) {
        DealocirajMemoriju(br_redova);
        delete pnb;
    }
}

template <typename Tip>
Matrica<Tip> &Matrica<Tip>::operator =(const Matrica<Tip> &m) {
    (*m.pnb)++;
    if(--(*pnb) == 0) DealocirajMemoriju(br_redova);
    strcpy(ime_matrice, m.ime_matrice);
    br_redova = m.br_redova; br_kolona = m.br_kolona;
    elementi = m.elementi; pnb = m.pnb;
    return *this;
}
```

Opisana tehnika brojanja referenciranja je prilično jednostavna i vrlo efikasna. Međutim, u praksi je situacija nešto složenija. Naime, kako se i dalje koriste plitke kopije, bilo koja modifikacija dinamičkih elemenata jednog objekta mijenja i dinamičke elemente drugog objekta (jer se, zapravo, radi o istim elementima u memoriji). Ovo nije prevelik problem ukoliko smo svjesni da radimo sa plitkim kopijama, ali kao što smo već ranije istakli, može djelovati kontraintuitivno. Interesantno je da postoji i veoma efikasan, ali nešto složeniji način rješavanja ovog problema. Osnovna ideja sastoji se u tome da se duboko kopiranje vrši samo u slučajevima kada je to zaista neophodno. Na primjer, plitko kopiranje je sasvim dobro sve dok se ne pojavi potreba za *izmjenom* elemenata dinamičkog niza pridruženog klasi. Stoga je moguće duboko kopiranje obavljati samo u metodama koje mijenjaju elemente odgovarajućeg dinamičkog

niza, i to samo u slučajevima kada brojač kopija ima vrijednost veću od 1 (što je siguran znak da još neki pokazivač pokazuje na isti dinamički niz). Opisana tehnika naziva se *kopiranje po potrebi* (engl. *copy when necessary*) ili *kopiranje pri upisu* (engl. *copy on write*). Ona se relativno jednostavno realizira u slučaju kada postoje jasno razdvojene metode koje samo čitaju i metode koje modifificiraju pripadni dinamički niz. S druge strane, efikasna realizacija ove tehnike može postati veoma komplikirana u slučajevima kada postoje metode koje se, zavisno od situacije, mogu koristiti kako za čitanje, tako i za modifificiranje elemenata dinamičkog niza (poput metode "Element" iz klase "VektorN"). Stoga, u detalje ove tehnike nećemo ulaziti. Cilj je bio samo da se čitatelj odnosno čitateljica upoznaju sa osnovnim idejama kako se rad sa klasama koje zauzimaju mnogo memorijskih resursa može učiniti efikasnijim. Ako zanemarimo aspekt efikasnosti, možemo reći da postupak kreiranja dubokih kopija koji smo detaljno objasnili u potpunosti zadovoljava sve druge aspekte za ispravno korištenje klasa.

Interesantno je napomenuti da su tipovi podataka "vector" i "string" sa kojima smo se do sada u više navrata susretali, implementirani upravo kao klase koje interno koriste dinamičku alokaciju memorije, i koje posjeduju propisno izvedene konstruktore kopije i prekopljene operatore dodjele. Zahvaljujući tome, njih je moguće bezbjedno prenositi kao parametre po vrijednosti, vraćati kao rezultate iz funkcije, i vršiti međusobno dodjeljivanje. Treba znati da objekti tipa "vector" i "string" uopće u sebi ne sadrže svoje "elemente". Objekti ovih tipova unutar sebe samo sadrže pokazivače na blokove memorije u kojem se nalaze njihovi "elementi", i još pokoji atribut neophodan za njihov ispravan rad. Međutim, zahvaljujući konstruktorima kopije i prekopljenim operatorima dodjele, korisnik ne može primijetiti ovu činjenicu, s obzirom da pripadni elementi "prate u stopu" svako kretanje samog objekta, odnosno ponašaju se kao "prikolica" trajno zakaćena na objekat. Jedini način kojim se korisnik može uvjeriti u ovu činjenicu je primjena operatara "**sizeof**" kojeg je nemoguće "prevariti". Naime, operator "**sizeof**" primijenjen na objekat tipa "vector" ili "string" (kao i na bilo koji drugi objekat) daće kao rezultat ukupan broj bajtova koji zauzimaju *atributi klase*, bez obzira na to što je eventualno "prikaćeno" na objekte. Stoga će rezultat primjene "**sizeof**" operatora na objekte tipa "vector" ili "string" biti uvijek isti, bez obzira na broj "elemenata" koje sadrži vektor ili dinamički string. Ovo usput objašnjava zbog čega operator "**sizeof**" primijenjen na vektore daje neočekivane rezultate, što smo već ranije naglasili.

Već smo rekli da se u slučaju da ne definiramo vlastiti konstruktor kopije, automatski generira podrazumijevani konstruktor kopije, koji prosto kopira sve atribute jednog objekta u drugi. Pri tome, ukoliko je neki atribut tipa klase koja posjeduje konstruktor kopije, njen konstruktor kopije će biti iskorišten za kopiranje odgovarajućeg atributa. Slično vrijedi i za operator dodjele. Slijedi da nije potrebno definirati vlastiti konstruktor kopije niti prekopljeni operator dodjele za bezbjedno kopiranje objekata koji sadrže npr. atribute tipa "vector" ili "string" (pa ni atribute tipa "VektorN" i "Matrica", koje smo sami razvili), jer će se za ispravno kopiranje objekata pobrinuti konstruktori kopije (odnosno prekopljeni operatori dodjele) odgovarajućih atributa. Naravno, konstruktor kopije će biti potreban u slučaju da pored takvih atributa, klasa posjeduje i dodatne pokazivače koji pokazuju na druge dinamički alocirane resurse.

Opisana činjenica pruža mogućnost da se u velikom broju praktičnih slučajeva u potpunosti izbjegne potreba za definiranjem konstruktora kopije i prekopljenog operatora dodjele. Naime, umjesto korištenja dinamičke alokacije memorije, možemo koristiti tipove "vector" i "string" koji pružaju sve pogodnosti koje pruža i dinamička alokacija memorije (što nije nikakvo iznenadenje, s obzirom da je njihova implementacija zasnovana upravo na dinamičkoj alokaciji memorije). S obzirom da se ovi tipovi kopiraju bez problema (zahvaljujući *njihovim* konstruktorima kopije), korisnik se ne mora brinuti o ispravnom kopiranju. Na primjer, pogledajmo kako bismo mogli realizirati klasu "VektorN" koristeći tip "vector" umjesto dinamičke alokacije memorije:

```
class VektorN {
    int br_elmenata;
```

```

vector<double> elementi;
public:
    explicit VektorN(int n) : br_elemenata(n), elementi(n) {}
    void Ispisi() const;
    double Element(int n) const;
    double &Element(int n);
    friend const VektorN ZbirVektora(const VektorN &v1,
                                      const VektorN &v2);
};

```

Konstruktor kopije i preklopjeni operator dodjele više nisu potrebni. Obratimo pažnju kako je konstruktor klase “VektorN” iskorišten za inicijalizaciju broja elemenata atributa “elementi” tipa “vector”. Naravno, sada nam je potpuno jasno da atribut “elementi” nismo mogli deklarirati deklaracijom poput

```
vector<double> elementi(n);
```

pa čak ni sličnom konstrukcijom kod koje bi u zagradi bio *broj*. Naime, sada znamo da je parametar u zagradi zapravo *poziv konstruktora* (a ne sastavni dio deklaracije) i on se unutar deklaracije klase mora obaviti onako kako se vrši poziv konstruktora bilo kojeg drugog atributa koji je tipa klase, o čemu smo detaljno govorili u prethodnom poglavlju. Implementacije metoda “Ispisi” i “Element”, kao i prijateljske funkcije “ZbirVektora” mogle bi ostati iste kao i do sada.

Ukoliko se sada pitate zbog čega smo se uopće patili sa dinamičkom alokacijom memorije, destruktorma, konstruktorima kopije i preklapanjem operatora dodjele kada problem možemo jednostavnije riješiti prostom upotrebom tipova poput “vector” ili “string”, odgovor je jednostavan: u suprotnom ne bismo mogli shvatiti kako ovi tipovi podataka zapravo rade, i ne bismo bili u stanju kreirati vlastite tipove podataka koji se ponašaju poput njih. Pored toga, korištenje tipa “vector” samo sa ciljem izbjegavanja dinamičke alokacije memorije i pratećih rezultata (konstruktora kopije, itd.) jeste najlakši, ali ne i najefikasniji način za rješavanje problema. Naime, deklariranjem nekog atributa tipa “vector”, u klasu koju razvijamo ugrađujemo sva svojstva klase “vector”, uključujući i svojstva koja vjerovatno nećemo uopće koristiti (isto vrijedi i za upotrebu atributa tipa “string”). Na taj način, klasa koja razvijamo postaje opterećena suvišnim detaljima, što dovodi do gubitka efikasnosti i bespotrebnog trošenja računarskih resursa.

Na kraju ovog poglavlja, napomenimo da smo u ranijim poglavljima u više navrata intuitivno uveli pojam POD (Plain Old Data) tipova podataka, kao tipova koji se mogu bezbjedno kopirati bajt po bajt. Sada možemo i precizno definirati šta su POD tipovi podataka. To su takvi tipovi podataka koji unutar sebe sadrže sve *informacije o sebi*, odnosno koji nemaju “prikolice” prikačene na njih. Na primjer, svi tipovi podataka koji interna sadrže pokazivače na dinamički alocirane resurse nisu POD tipovi. Također, tipovi podataka koji sadrže neki atribut čiji tip nije POD tip, ni sami ne mogu biti POD tipovi. Na primjer, klasa koja sadrži atribut tipa “string” ne može biti POD tip. Konkretno, POD tipovi su oni i samo oni tipovi za čije bezbjedno kopiranje nije potreban konstruktor kopije, i koji pored toga ne sadrže atrubute koji posjeduju vlastiti konstruktor kopije.

33. Preklapanje operatora

Složeni tipovi podataka koje smo uveli u prethodnim poglavljima, kao što su strukture a pogotovo klase, omogućavaju dizajniranje takvih tipova podataka koji na jednostavan i prirodan način modeliraju stvarne objekte sa kojima se susrećemo prilikom rješavanja realnih problema. Međutim, jedini operatori koji su na početku definirani za ovakve tipove podataka su operator dodjele “=” i operator uzimanja adrese “&”. Sve ostale operacije nad složenim objektima do sada smo izvodili bilo pozivom funkcija i prenošenjem objekata kao parametara u funkcije, bilo pozivom funkcija članica nad objektima. Tako smo, na primjer za ispis nekog objekta “v” tipa “Vektor” koristili poziv poput “v.Ispisi()”, dok smo za sabiranje dva vektora “v1” i “v2” i dodjelu rezultata trećem vektoru “v3” koristili konstrukciju poput “v3 = ZbirVektora(v1, v2)”. Rad sa tipom “Vektor” bio bi znatno olakšan kada bismo za manipulacije sa objektima ovog tipa mogli koristiti *istu sintaksu* kakvu koristimo pri radu sa prostim ugrađenim tipovima podataka, tj. kada bismo za ispis vektora mogli koristi konstrukciju “cout << v”, a za sabiranje konstrukciju “v3 = v1 + v2”. Upravo ovaku mogućnost nudi nam *preklapanje* odnosno *preopterećivanje operatora* (engl. *operator overloading*). Preciznije, preklapanje (preopterećivanje) operatora nam omogućava da nekim od operatora koji su definirani za proste tipove podataka damo smisao i za složene tipove podataka koje smo sami definirali. U prethodnom poglavlju smo se upoznali sa preklapanjem operatora dodjele, koje predstavlja specijalni slučaj općeg postupka preklapanja operatora, koji ćemo detaljno razmotriti u ovom poglavlju.

Preklapanje operatora ostvaruje se uz pomoć *operatorskih funkcija*. Operatorske funkcije izgledaju kao i klasične funkcije, samo što umjesto imena imaju ključnu riječ “**operator**” iza koje slijedi oznaka nekog od postojećih operatora. Operatorske funkcije mogu biti i funkcije članice klase, ali takvu ćemo mogućnost razmotriti nešto kasnije. Za sada ćemo prepostaviti da su operatorske funkcije izvedene kao obične funkcije. Operatorske funkcije mogu imati dva parametra, ukoliko je navedeni operator *binarni operator*, ili jedan parametar ukoliko je navedeni operator *unarni operator*. Međutim, tip barem jednog od parametara operatorske funkcije mora biti *korisnički definirani tip podataka*, u koji spadaju strukture, klase, i pobrojani tipovi (odnosno tipovi definirani deklaracijom “**enum**”). Na primjer, neka je data sljedeća deklaracija klase “Vektor”, u kojoj su, radi jednostavnosti, definirani samo konstruktor i trivijalne metode za pristup atributima klase:

```
class Vektor {
    double x, y, z;
public:
    Vektor(double x, double y, double z) : x(x), y(y), z(z) {}
    double Vrati_x() const { return x; }
    double Vrati_y() const { return y; }
    double Vrati_z() const { return z; }
};
```

Za ovaku klasu možemo definirati sljedeću operatorsku funkciju koja obavlja sabiranje dva vektora (razmak iza riječi “**operator**” može se izostaviti):

```

const Vektor operator +(const Vektor &v1, const Vektor &v2) {
    return Vektor(v1.Vrati_x() + v2.Vrati_x(),
                  v1.Vrati_y() + v2.Vrati_y(), v1.Vrati_z() + v2.Vrati_z());
}

```

Operatorsku funkciju principijelno je moguće pozvati kao i svaku drugu funkciju. Na primjer, ukoliko su “a”, “b” i “c” objekti tipa “Vektor”, sljedeća konstrukcija je sasvim ispravna:

```
c = operator +(a, b);
```

Međutim, razlog zbog čega se uopće uvode operatorske funkcije je u tome što je sada moguće prosto pisati

```
c = a + b;
```

Razmotrimo ovu mogućnost malo općenitije. Neka je “ \oplus ” proizvoljan binarni operator podržan u jeziku C++ i neka su “x” i “y” neki objekti od kojih je barem jedan tipa strukture, klase ili pobrojanog tipa. Tada se izraz

$x \oplus y$

prvo pokušava interpretirati kao izraz

```
operator  $\oplus$ (x, y)
```

Drugim riječima, pri nailasku na neki izraz sa korisnički definiranim tipovima podataka koji sadrži binarne operatore, prilikom interpretacije tog izraza prvo se pokušavaju pozvati odgovarajuće operatorske funkcije, koje odgovaraju upotrijebljenim operatorima (kako po oznaci operatora, tako i po tipovima argumenata). Ukoliko takve operatorske funkcije postoje, prosto se vrši njihov poziv, čiji rezultat daje interpretaciju izraza. Međutim, ukoliko takve operatorske funkcije ne postoje, vrši se pokušaj pretvorbe operanada u neke druge tipove za koje je odgovarajući operator definiran, što uključuje podrazumijevane pretvorbe u standardne ugrađene tipove (poput pretvorbe pobrojanih tipova u cjelobrojne, o čemu smo ranije detaljno govorili), kao i korisnički definirane pretvorbe, koje se mogu ostvariti konstruktorima sa jednim parametrom (o čemu smo također govorili), kao i operatorskim funkcijama za pretvorbu (o čemu ćemo govoriti kasnije). Ukoliko se i nakon obavljenih pretvorbi ne može naći odgovarajuća interpretacija navedenog izraza, prijavljuje se greška. Greška se također prijavljuje ukoliko je pretvorbe moguće izvršiti na više različitih međusobno nesaglasnih načina.

Slično vrijedi i za slučaj unarnih operatora. Ukoliko je “ \blacklozenge ” neki unarni operator, tada se izraz

$\blacklozenge x$

prvo pokušava interpretirati kao izraz

operator $\blacklozenge (x)$

Tek ukoliko odgovarajuća operatorska funkcija ne postoji, pokušava se pretvorba u neki drugi tip za koji je taj operator definiran, osim ukoliko se radi o operatuoru uzimanja adrese “&” (koji je podrazumijevano definiran i za korisnički definirane tipove). Ukoliko takve pretvorbe nisu podržane, prijavljuje se greška.

Kao primjer operatorske funkcije za preklapanje unarnih operatora, možemo definirati operatorsku funkciju za klasu “Vektor”, koja omogućava da se unarna varijanta operatara “-” može primijeniti na objekte tipa “Vektor” (sa značenjem obrtanja smjera vektora, kao što je uobičajeno u matematici):

```
const Vektor operator -(const Vektor &v) {
    return Vektor(-v.Vrati_x(), -v.Vrati_y(), -v.Vrati_z());
```

Izvjesni operatori (kao što je upravo pomenuti operator “-”) postoje i u unarnoj i u binarnoj varijanti, tako da možemo imati za isti operator operatorsku funkciju sa jednim parametrom (koja odgovara unarnoj varijanti) i sa dva parametra (koja odgovara binarnoj varijanti). Tako, na primjer, za klasu “Vektor” možemo definirati i binarni operator “-” za oduzimanje na sljedeći način:

```
const Vektor operator -(const Vektor &v1, const Vektor &v2) {
    return Vektor(v1.Vrati_x() - v2.Vrati_x(),
                  v1.Vrati_y() - v2.Vrati_y(), v1.Vrati_z() - v2.Vrati_z());
```

Međutim, istu operatorsku funkciju mogli smo napisati i mnogo jednostavnije na sljedeći način (zbog svoje kratkoće, funkcija je pogodna za realizaciju kao umetnuta funkcija, što smo i učinili):

```
inline const Vektor operator -(const Vektor &v1, const Vektor &v2) {
    return v1 + -v2;
```

```
}
```

Ovdje je iskorištena činjenica da smo prethodno definirali operator “unarni minus” za tip “Vektor” (tako da je kompjuter “naučio” šta znači konstrukcija “ $-v2$ ”), kao i da smo definirali binarni operator “+” za isti tip (tako da kompjuter zna kako se sabiraju dva vektora). Ovdje ne treba naivno pomisliti da smo umjesto “ $v1 + -v2$ ” mogli napisati “ $v1 - v2$ ”, s obzirom da se matematski gledano radi o dva ista izraza. Naime, značenje izraza “ $v1 + -v2$ ” je od ranije dobro definirano, s obzirom da smo prethodno definirali značenje binarnog operatorka “+” i unarnog minusa za objekte tipa “Vektor”. S druge strane, značenje izraza “ $v1 - v2$ ” tek trebamo definirati, i to upravo pisanjem operatorske funkcije za binarni operator “-”. Upotreba izraza “ $v1 - v2$ ” unutar ove operatorske funkcije bila bi zapravo interpretiran kao da ova operatorska funkcija treba pozvati samu sebe (tj. kao rekursija, i to bez izlaza)! Pored toga, programer ima puno pravo da napiše operatorsku funkciju za binarni operator “-” kako god želi (pa čak i tako da uopće ne obavlja oduzimanje nego nešto drugo), tako da izrazi “ $v1 + -v2$ ” i “ $v1 - v2$ ” uopće ne moraju imati isto značenje (mada to nije dobra praksa). U svakom slučaju, prvi od ovih izraza se interpretira kao

```
operator +(v1, operator -(v2))
```

a drugi kao

```
operator -(v1, v2)
```

Ipak, zbog razloga efikasnosti, bolje je operatorsku funkciju za binarni operator “-” implementirati neposredno, ne oslanjajući se na operatorske funkcije za sabiranje i unarni minus (nije teško vidjeti da će ukupan broj izvršenih operacija biti manji u tom slučaju).

Operatorske funkcije se također mogu deklarirati kao funkcije prijateljki klase. U praksi se gotovo uvijek tako radi, s obzirom da u većini slučajeva operatorske funkcije trebaju pristupati internim atributima klase. Na primjer, sasvim je razumno u klasi “Vektor” operatorske funkcije koje smo razmatrali deklarirati kao prijateljske funkcije:

```
class Vektor {
    double x, y, z;
public:
    Vektor(double x, double y, double z) : x(x), y(y), z(z) {}
    double Vrati_x() const { return x; }
    double Vrati_y() const { return y; }
    double Vrati_z() const { return z; }
    friend const Vektor operator +(const Vektor &v1, const Vektor &v2);
    friend const Vektor operator -(const Vektor &v1, const Vektor &v2);
    friend const Vektor operator -(const Vektor &v1, const Vektor &v2);
};
```

Uz ovakvu deklaraciju, definicije pomenutih operatorskih funkcija moglo bi se pojednostaviti:

```
const Vektor operator +(const Vektor &v1, const Vektor &v2) {
    return Vektor(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z);
}

const Vektor operator -(const Vektor &v) {
    return Vektor(-v.x, -v.y, -v.z);
}

const Vektor operator -(const Vektor &v1, const Vektor &v2) {
    return Vektor(v1.x - v2.x, v1.y - v2.y, v1.z - v2.z);
}
```

Možemo primijetiti da u slučaju da smo operatorsku funkciju za binarni operator “-” definirali skraćenim postupkom (preko sabiranja i unarnog minusa), ne bi bilo potrebe da je deklariramo kao funkciju prijatelja klase. S druge strane, od takve deklaracije ne bi bilo ni štete. Zbog toga se smatra dobrom praksom sve operatorske funkcije koje manipuliraju sa objektima neke klase uvijek deklarirati kao prijatelje te klase, jer se tada posmatranjem interfejsa klase jasno vidi koji su sve operatori definirani za datu klasu. U nastavku ćemo podrazumijevati da su sve operatorske funkcije koje budemo definirali deklarirane kao prijatelji klase sa kojom manipuliraju.

Sasvim je moguće imati više operatorskih funkcija za isti operator, pod uvjetom da se one razlikuju po *tipu svojih argumenata*, tako da kompjajler može nedvosmisleno odrediti koju operatorsku funkciju treba pozvati u konkretnoj situaciji. Na primjer, kako je dozvoljeno pomnožiti realni broj sa vektorom, prirodno bi bilo definirati sljedeću operatorsku funkciju:

```
const Vektor operator *(double d, const Vektor &v) {
    return Vektor(d * v.x, d * v.y, d * v.z);
}
```

Nakon ovakve definicije, izrazi poput “ $3 * v$ ” ili “ $c * v$ ” gdje je “ v ” vektor a “ c ” realan broj postaju posve smisleni. S druge strane, izraz poput “ $v * 3$ ” je i dalje nedefiniran, i doveće do prijave greške. Naime, prethodnom definicijom kompjajler je “naučio” kako se množi broj sa vektorom, ali ne i vektor sa brojem! Neko bi mogao prigovoriti da bi kompjajler prosto mogao primijeniti komutativni zakon na operator “*”. Međutim, usvojeno je da se prilikom preklapanja operatora ne prave nikakve pretpostavke o eventualnoj komutativnosti operatora, jer bi u suprotnom bilo nemoguće definirati operatore koji krše ove zakone (npr. ne bi bilo moguće definirati množenje matrica, koje nije komutativno). Zbog toga, ukoliko želimo dati

smisao izrazima poput “ $v * 3$ ”, moramo definirati još jednu operatorsku funkciju za binarni operator “ $*$ ”, koja bi glasila ovako

```
const Vektor operator *(const Vektor &v, double d) {
    return Vektor(d * v.x, d * v.y, d * v.z);
}
```

Ova funkcija se od prethodne razlikuje samo po tipu parametara, a implementacija joj je potpuno ista. Da uštedimo na pisanju, mogli smo pisati i ovako:

```
inline const Vektor operator *(const Vektor &v, double d) {
    return d * v;
}
```

Na ovaj način smo eksplisitno rekli da je “ $v * d$ ” gdje je “ v ” vektor a “ d ” broj isto što i “ $d * v$ ”, a postoji operatorska funkcija koja objašnjava kakav je smisao izraza “ $d * v$ ”. Interesantno je da ovakvom definicijom ništa ne gubimo na efikasnosti, s obzirom da smo ovu operatorsku funkciju izveli kao umetnutu funkciju, tako da se izraz oblika “ $v * d$ ” prosto zamjenjuje izrazom “ $d * v$ ” (odnosno, pripadna operatorska funkcija se ne poziva, već se njeno tijelo prosto umeće na mjesto poziva). Da ovu operatorsku funkciju nismo izveli kao umetnutu, imali bismo izvjestan gubitak na efikasnosti, s obzirom da bi izračunavanje izraza “ $v * d$ ” prvo dovelo do poziva jedne operatorske funkcije, koja bi dalje pozvala drugu operatorsku funkciju (dakle, imali bismo dva poziva umjesto jednog).

Prethodne definicije još uvijek ne daju smisla izrazima poput “ $a * b$ ” gdje su i “ a ” i “ b ” objekti tipa “*Vektor*”. Za tu svrhu potrebno je definirati još jednu operatorsku funkciju za binarni operator “ $*$ ”, čija će oba parametra biti tipa “*Vektor*”. Međutim, u matematici se produkt dva vektora može interpretirati na dva načina: kao *skalarni produkt* (čiji je rezultat *broj*), i kao *vektorski produkt* (čiji je rezultat *vektor*). Ne možemo napraviti obje interpretacije i pridružiti ih operatoru “ $*$ ”, jer se u tom slučaju neće znati na koju se interpretaciju izraz “ $a * b$ ” odnosi (u matematici je taj problem riješen uvođenjem različitih oznaka operatora za ove dvije interpretacije, tako da operator “ \cdot ” označava skalarni, a operator “ \times ” vektorski produkt). Slijedi da se moramo odlučiti za jednu od interpretacija. Ukoliko se dogovorimo da izraz “ $a * b$ ” interpretiramo kao skalarni produkt, možemo napisati sljedeću operatorsku funkciju, koja definira tu interpretaciju:

```
double operator *(const Vektor &v1, const Vektor &v2) {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
```

Ovo ne znači da se moramo odreći mogućnosti da definiramo i operator za vektorski produkt dva vektora, samo to više ne može biti operator “ \times ”. Na žalost, ne postoji mogućnost dodavanja novih operatora, već samo proširivanje značenja već postojećih operatora. Na primjer, nije moguće definirati operator “@” i dati mu neko značenje, s obzirom da takav operator ne postoji u jeziku C++. Zbog toga nije moguće definirati ni operator “ \times ” sa ciljem da nam “ $a \times b$ ” predstavlja vektorski produkt dva vektora. Stoga, jedino što možemo učiniti je da nekom od postojećih operatora proširimo ulogu da obavlja računanje vektorskog produkta. Na primjer, mogli bismo definirati da operator “/” primijenjen na vektore predstavlja vektorsko množenje, tako da u slučaju kada su “ a ” i “ b ” vektori, “ a / b ” predstavlja njihov vektorski produkt. Na žalost, takvo rješenje bi moglo biti zbumujuće, s obzirom da za slučaj kada su “ a ” i “ b ” brojevi, izraz “ a / b ” označava *dijeljenje*. Možda je bolje odlučiti se za operator “%”, tako da definiramo da nam “ $a \% b$ ” označava vektorski produkt (ako ništa drugo, onda barem zbog činjenice da znak “%” više vizuelno podsjeća na znak “ \times ” od znaka “/”):

```
const Vektor operator %(const Vektor &v1, const Vektor &v2) {
    return Vektor(v1.y * v2.z - v1.z * v2.y, v1.z * v2.x - v1.x * v2.z,
        v1.x * v2.y - v1.y * v2.z);
}
```

Pored toga što nije moguće uvoditi nove operatore, nije moguće ni promijeniti *prioritet operatora*. Na primjer, da smo definirali da operator “&” predstavlja vektorsko množenje, njegov prioritet bi ostao niži od prioriteta operatora “+”, tako da bi se izraz “ $v1 + v2 \& v3$ ” interpretirao kao “ $(v1 + v2) \& v3$ ” a ne kao “ $v1 + (v2 \& v3)$ ”. Ovo je još jedan razlog zbog čega je izbor operatora “%” relativno dobar izbor za vektorski produkt dva vektora. Naime, prioritet ovog operatora je u istom rangu kao i prioritet klasičnog operatora za množenje “ \times ”.

Na ovom mjestu ćemo još jednom naglasiti da operatorske funkcije moraju imati barem jedan argument koji je korisnički definiranog tipa (strukture, klase ili pobrojanog tipa). Ovo je urađeno da se spriječi mogućnost *promjene* značenja pojedinih operatora za proste ugrađene tipove. Na primjer, nije moguće napisati operatorsku funkciju poput

```
int operator +(int x, int y) {
    return x * y;
}
```

kojom bismo postigli da vrijednost izraza “ $2 + 3$ ” bude “ 6 ” a ne “ 5 ”. Mada bi, u izvjesnim situacijama, promjena značenja pojedinih operatora za proste ugrađene tipove mogla biti od koristi, tvorci jezika C++ su zaključili da bi šteta od moguće zloupotrebe ovakvih konstrukcija mogla biti znatno veća od eventualne koristi, pa su odlučili da zabrane ovaku mogućnost.

Gotovo svi operatori ugrađeni u jezik C++ mogu se preklopiti (tj. dodefinirati). Jedini

operatori koje nije moguće preklopiti su operatori “`:`”, “`.`”, “`.*`”, “`?:`” i operatori “`sizeof`” i “`typeid`”, s obzirom da ovi operatori imaju u jeziku C++ toliko specifične primjene da bi mogućnost njihovog preklapanja dovela do velike zbrke. Tako je moguće preklopiti sljedeće binarne operatore:

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code><<</code>	<code>>></code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>
<code>==</code>	<code>!=</code>	<code>&</code>	<code>^</code>	<code> </code>	<code>&&</code>	<code> </code>	<code>=</code>	<code>+=</code>	<code>-=</code>	<code>*=</code>
<code>/=</code>	<code>%=</code>	<code>&=</code>	<code>^=</code>	<code> =</code>	<code><<=</code>	<code>>>=</code>	<code>->*</code>	<code>,</code>		

zatim sljedeće unarne operatore:

<code>+</code>	<code>-</code>	<code>!</code>	<code>~</code>	<code>&</code>	<code>*</code>	<code>++</code>	<code>--</code>
----------------	----------------	----------------	----------------	--------------------	----------------	-----------------	-----------------

kao i sljedeće specijalne operatore koje je teško svrstati među gore prikazane operatore:

<code>[]</code>	<code>()</code>	<code>-></code>	<code>new</code>	<code>new[]</code>	<code>delete</code>	<code>delete[]</code>
-----------------	-----------------	--------------------	------------------	--------------------	---------------------	-----------------------

Svi navedeni binarni operatori osim operatora dodjele “`=`”, kao i svi navedeni unarni operatori, mogu se preklopiti na već opisani način (pri čemu preklapanje operatora “`++`” i “`--`” zahtijeva i neke dodatne specifičnosti koje ćemo kasnije objasniti, zbog činjenice da ovi operatori imaju kako prefiksni, tako i postfiksni oblik). O preklapanju operatora dodjele smo već govorili u prethodnom poglavlju (mada ćemo kasnije dati još nekoliko napomena vezanih za njegovo preklapanje), dok ćemo o preklapanju specijalnih operatora govoriti nešto kasnije. Prije toga ćemo reći nekoliko riječi o tome kako bi trebalo preklapati pojedine operatore. Kažemo “trebalo”, s obzirom da programer ima punu slobodu da značenje pojedinih operatora za korisnički definirane tipove podataka definira *kako god želi*. Tako je sasvim moguće definirati da operator “`+`” obavlja oduzimanje dva vektora, a da unarni operator “`-`” računa dužinu vektora. Vjerovatno ne treba posebno napominjati da takvo definiranje nije nimalo mudro. Međutim, ne postoji formalni mehanizam da se to spriječi, isto kao što je nemoguće spriječiti nekoga da promjenljivu koja čuva poluprečnik kruga nazove “`brzina`”, niti da funkciju koja nalazi rješenja kvadratne jednačine nazove “`ProduktMatrica`”. Stoga ćemo u nastavku reći par riječi kako bi trebalo definirati izvjesne operatore sa ciljem da se održi konzistencija sa načinom djelovanja pojedinih operatora na različite proste ugrađene tipove.

Prvo pravilo “bontona” prilikom preklapanja operatora kaže da je neki operator potrebno definirati za neki tip samo ukoliko je *intuitivno jasno* šta bi taj operator trebao da znači za taj tip. Na primjer, sasvim je jasno šta bi operator “`+`” trebao da znači za dva objekta koji predstavljaju neke matematske strukture za koje je pojam sabiranja definiran. Također, ukoliko pravimo klasu koja omogućava rad sa nizovima znakova, ima smisla definirati operator “`+`” koji bi mogao da predstavlja nadovezivanje nizova znakova jedan na drugi (kao što je izvedeno u standardnoj klasi “`string`” iz istoimene biblioteke). S druge strane, teško je dati smisleno značenje izrazu “`a + b`” za slučaj kada su “`a`” i “`b`” objekti klase “`Student`”.

Drugo pravilo preporučuje da kada god definiramo neki od binarnih aritmetičkih operatora kao

što su “+”, “-”, “*” itd. trebamo definirati i odgovarajuće operatore sa pridruživanjem poput “+=”, “-=”, “*=” itd. Naime, pri radu sa prostim ugrađenim tipovima programeri su navikli da umjesto “ $a = a + b$ ” pišu “ $a += b$ ”. Stoga bi lijepo bilo da isto vrijedi i za rad sa složenim tipovima za koje je prethodno definiran operator “+”. Međutim, ovo se neće podrazumijevati samo po sebi, jer je “+=” posve drugačiji operator od operatora “+”, i činjenica da je definiran operator “+” uopće ne povlači da je definiran i operator “+=”. Pored toga, izraz “ $a += b$ ” se interpretira kao

```
operator +=(a, b)
```

iz čega slijedi da se on principijelno može definirati tako da ima potpuno drugačije značenje od izraza “ $a = a + b$ ”, koji se interpretira kao

```
a = operator +(a, b)
```

koji se, u slučaju da je operator “=” također preklopljen, dalje interpretira kao

```
a.operator =(operator +(a, b))
```

Ipak, takva definicija bi bila protivna svim mogućim kodeksima programerskog bontona. Stoga bi operator “+=” trebalo definirati tako da izrazi “ $a = a + b$ ” i “ $a += b$ ” imaju isto značenje. Slijedi jedan jednostavan način kako se ovo može izvesti za tip “Vektor”:

```
inline void operator +=(Vektor &v1, const Vektor &v2) {
    v1 = v1 + v2;
}
```

Ovdje je veoma bitno uočiti da je prvi formalni parametar u ovoj operatorskoj funkciji *referenca na nekonstantni objekat* (odnosno, imamo klasični prenos parametra po referenci). Naime, izraz “ $a += b$ ” koji se interpretira kao “`operator +=(a, b)`” treba da promijeni vrijednost objekta “ a ”, a to je moguće ostvariti jedino prenosom po referenci (i to na nekonstantni objekat). Također, treba primijetiti da smo se u izrazu “ $v1 = v1 + v2$ ” oslonili na činjenicu da smo definirali značenje operatorka “+” za vektore. To i nije loše, jer na taj način čvrsto držimo konzistenciju između značenja operatorka “+” i “+=”. Međutim, ukoliko se zbog efikasnosti ne želimo oslanjati na definiciju operatorka “+” za klasu “Vektor”, operatorsku funkciju za operatorka “+=” možemo napisati i na sljedeći način, koji je posve neovisan od definicije drugih operatorka:

```
void operator +=(Vektor &v1, Vektor v2) {
    v1.x += v2.x; v1.y += v2.y; v1.z += v2.z;
}
```

Obje navedene definicije operatorske funkcije za operator “`+=`” rade posve dobro. Ipak, one nisu u potpunosti saglasne sa onim kako operator “`+=`” djeluje na proste ugrađene tipove. Naime, za slučaj kada su “`a`”, “`b`” i “`c`” objekti nekog prostog ugrađenog tipa (npr. cjelobrojne promjenljive) sasvim je legalno pisati “`a = b += c`”. Sa ovako napisanom operatorskom funkcijom i tipom “`Vektor`” to nije moguće, jer smo joj povratni tip definirali da bude “`void`” (tj. da ne vraća ništa), tako da izraz “`b += c`” ne nosi nikakvu vrijednost koja bi mogla biti dodijeljena promjenljivoj “`a`”. Ovaj nedostatak je lako otkloniti tako što ćemo modificirati operatorsku funkciju da vraća kao rezultat referencu na izmijenjeni vektor (vraćanjem reference izbjegavamo kopiranje objekta koji već postoji):

```
inline Vektor &operator +=(Vektor &v1, const Vektor &v2) {
    return v1 = v1 + v2;
}
```

Vraćanje reference omogućava da se izraz u kojem se javlja operator “`+=`” može naći sa lijeve strane znaka jednakosti. Mada je teško vidjeti neku korist od takve mogućnosti, ona postoji za proste ugrađene tipove, i stoga nije na odmet podržati je i za korisnički definirane tipove. Ukoliko želimo, takvu mogućnost možemo spriječiti vraćanjem *reference na konstantan objekat*.

Kao što je već rečeno, programer ima slobodu da definira operatore kako god želi. Tako je posve moguće definirati operator poput “`+=`” čak i u slučaju da za isti tip nije definiran operator “`+`”. Na primjer, uzimimo da je “`razred`” objekat tipa “`Razred`”, a “`ucenik`” objekat tipa “`Ucenik`”, pri čemu su klase “`Razred`” i “`Ucenik`” deklarirane kao u programu iz poglavlja koje govori o konstruktorima i destruktorma. Neko bi mogao doći na ideju da definira operator “`+=`” koji bi djelovao na ove klase, pri čemu bi izraz poput “`razred += ucenik`” imao značenje upisivanja učenika “`ucenik`” u razred “`razred`”. Međutim, ovakvo rješenje se ne smatra dobrim, s obzirom da njegovo značenje nije posve očigledno na prvi pogled, a pored toga i nema nikakve veze sa sabiranjem. Sigurno je izraz poput “`razred.DodajUcenika(ucenik)`” mnogo jasniji i prirodniji. Odavde slijedi zaključak da operatore treba definirati samo u slučajevima kada se potreba za njihovim definiranjem prirodno nameće.

Sljedeće pravilo preporučuje da ukoliko smo se uopće odlučili da definiramo operatore za neku klasu, treba definirati i smisao operatora “`==`” i “`!=`”. Ovi operatori bi, naravno, trebali testirati jednakost odnosno različitost dva objekta (mada, naravno, nema načina da programeru naredimo da poštuje ovaku konvenciju – on ove operatore može definirati kako god hoće). Za klasu “`Vektor`”, odgovarajuće operatorske funkcije bi mogle izgledati ovako:

```
bool operator ==(const Vektor &v1, const Vektor &v2) {
    return v1.x == v2.x && v1.y == v2.y && v1.z == v2.z;
```

```

    }

bool operator !=(const Vektor &v1, const Vektor &v2) {
    return v1.x != v2.x || v1.y != v2.y || v1.z != v2.z;
}

```

Operatorsku funkciju za operator “!=” smo mogli napisati i na sljedeći način, oslanjajući se na postojeću definiciju operatorske funkcije za operator “==” i na značenje operatora “!” primjenjenog na cijelobrojne tipove:

```

inline bool operator !=(const Vektor &v1, const Vektor &v2) {
    return !(v1 == v2);
}

```

Na ovaj način smo istakli da želimo da značenje izraza “ $v1 \neq v2$ ” treba da bude isto kao i značenje izraza “ $!(v1 == v2)$ ”, što se ne podrazumijeva samo po sebi sve dok ne definiramo da je tako. Naime, mada su ova dva izraza praktično jednaka za proste ugrađene tipove, principijelno je moguće (mada se ne preporučuje) definirati operatore “==” i “!=” na takav način da ovi izrazi budu različiti.

Za slučaj kada je objekte određenog tipa moguće staviti u određeni poredak, preporučljivo je definirati operatore “<”, “<=”, “>” i “>=”. Na primjer, za objekte tipa “Student”, ima smisla definirati da je jedan student “veći” od drugog ako ima bolji prosjek, itd. Posebno je korisno definirati operator “<” ukoliko nad primjercima klase želimo koristiti funkcije poput funkcije “sort” iz biblioteke “algorithm”, s obzirom da ona kao i većina njih srodnih funkcija koriste upravo operator “<” kao kriterij poređenja u slučaju da korisnik ne navede eksplicitno neku drugu funkciju kriterija kao parametar. Kao i za sve ostale operatore, i ovdje vrijedi pravilo da su svi operatori poretka (“<”, “<=”, “>” i “>=”) posve neovisni jedan od drugog, kao i od operatora “==” i “!=”, pa je svakom od njih moguće dati potpuno nevezana značenja. Međutim, posve je jasno da bi ove operatore trebalo definirati tako da zaista odražavaju smisao poretka za objekte koji se razmatraju. Kako za vektore poredak nije definiran, ove operatore za klasu “Vektor” nećemo definirati.

Interesantno je razmotriti operatore “<<” i “>>”. Mada programer i ove operatore može definirati da obavljaju ma kakvu funkciju, njihovo prirodno značenje je ispis na izlazni tok i čitanje sa ulaznog toka. Stoga, kad god je objekte neke klase moguće na smislen način ispisivati na ekran ili unositi sa tastature, poželjno je podržati da se te radnje obavljaju pomoću operatora “<<” i “>>”. Na primjer, ukoliko je “v” objekat klase “Vektor”, sasvim je prirodno podržati ispis vektora na ekran pomoću konstrukcije “cout << v” umjesto konstrukcije “v.Ispisi()” koju smo do sada koristili. Razmotrimo kako se ovo može uraditi. Prvo primijetimo da se izraz “cout << v” interpretira kao

```
operator <<(cout, v);
```

Ukoliko se sjetimo da objekat “cout” nije ništa drugo nego jedna instanca klase “ostream”, lako ćemo zaključiti da prvi parametar tražene operatorske funkcije treba da bude tipa “ostream”, a drugi tipa “vektor”. Preciznije, prvi formalni parametar mora biti *referenca na nekonstantni objekat* tipa “ostream”. Referenca na nekonstantni objekat je neophodna zbog činjenice da je klasa “ostream” veoma složena klasa koja sadrži attribute poput pozicije tekućeg ispisa koji se svakako *mijenjaju* tokom ispisa. Ukoliko bi se stvarni parametar “cout” prenosio *po referenci na konstantni objekat*, izmjene ne bi bile moguće, i bila bi prijavljena greška. Također, ukoliko bi se stvarni parametar “cout” prenosio *po vrijednosti*, sve izmjene ostvarene tokom ispisa, odrazile bi se samo na *lokalnu kopiju* objekta “cout”, što bi onemogućilo dalji ispravan rad izlaznog toka. Srećom, gotovo sve implementacije klase “ostream” zabranjuju da se objekti tipa “ostream” uopće mogu prenosi po vrijednosti (primjenom tehnika opisanih u prethodnom poglavljiju), tako da će nam kompjajler vrlo vjerovatno prijaviti grešku ukoliko uopće pokušamo tako nešto.

Razmotrimo šta bi trebala da vraća kao rezultat operatorska funkcija za preklapanje operatora “<<” sa ciljem podrške ispisa na izlazni tok. Ukoliko bismo kao povratni tip prosto stavili “void”, onemogućili bismo ulančavanje operatora “<<” kao u konstrukciji poput “cout << v << endl”. Kako se ova konstrukcija zapravo interpretira kao “(cout << v) << endl”, odnosno, uz preklopjeni operator “<<”, kao “**operator** <<(**operator** <<(cout, v), endl)”, jasno je da tražena operatorska funkcija treba da vrati *sam objekat izlaznog toka kao rezultat* (odnosno, referencu na njega, čime se izbjegava kopiranje masivnog objekta, koje u ovom slučaju nije ni dozvoljeno). Stoga bi implementacija tražene operatorske funkcije mogla izgledati recimo ovako:

```
ostream &operator <<(ostream &cout, const Vektor &v) {
    cout << "(" << v.x << "," << v.y << "," << v.z << ")";
    return cout;
}
```

Uzmemli u obzir da operator “<<” primijenjen na proste ugrađene tipove također vraća referencu na izlazni tok kao rezultat, prethodnu funkciju možemo još kraće napisati ovako:

```
ostream &operator <<(ostream &cout, const Vektor &v) {
    return cout << "(" << v.x << "," << v.y << "," << v.z << ")";
}
```

Naravno, nema nikakvih razloga da se formalni parametar mora također zvati “cout” (mada nema nikakvih razloga i da se ne zove tako), stoga smo istu funkciju mogli napisati i ovako:

```

void operator <<(ostream &izlaz, const Vektor &v) {
    return izlaz << "(" << v.x << "," << v.y << "," << v.z << ")";
}

```

Ovim primjerom smo samo htjeli istaći da je “`cout`” u suštini obična promjenljiva (iako veoma specifičnog tipa). Također, treba istaći da podrška za ispis “običnih” tipova podataka konstrukcijama poput “`cout << 3`”, “`cout << "Pozdrav"`” i slično nije ostvarena nikakvom posebnom magijom, nego su u biblioteci “`iostream`” jednostavno definirane odgovarajuće operatorske funkcije za operator “`<<`” koje obavljaju njihov ispis. Prvi parametar ovakvih operatorskih funkcija je naravno referenca na objekat tipa “`ostream`”, dok su drugi parametri prostih ugrađenih tipova poput “`int`”, “`char *`” itd. Druga stvar je kako je realizirana implementacija tih funkcija (za tu svrhu korištene su funkcije mnogo nižeg nivoa za direktni pristup izlaznim uređajima, u šta nećemo ulaziti).

Na potpuno analogan način možemo definirati i operator “`>>`” za čitanje sa ulaznog toka, samo trebamo voditi računa da je objekat “`cin`” instanca klase “`istream`”. Tako bi operatorska funkcija koja bi omogućila čitanje vektora sa tastature konstrukcijom poput “`cin >> v`”, pri čemu bi se vektor unosio kao obična trojka brojeva razdvojenih prazninama, mogla izgledati ovako:

```

istream &operator >>(istream &cout, Vektor &v) {
    return cin >> v.x >> v.y >> v.z;
}

```

Bitno je napomenuti da se ovdje drugi parametar također mora prenijeti po referenci na nekonstantni objekat, jer konstrukcija poput “`cin >> v`”, koja se interpretira kao “`operator >>(cin, v)`”, mora biti u stanju da promijeni sadržaj objekta “`v`”.

Od binarnih operatora moguće je preklopiti još i operatore “`&`”, “`^`”, “`|`”, “`&&`”, “`||`”, “`->*`” i “`,`”. Međutim, ovi operatori imaju prilično specifična značenja za standardne ugrađene tipove podataka, i dosta je teško zamisliti šta bi oni trebali da rade ukoliko bi se primijenili na korisnički definirane tipove podataka (mada nesumnjivo postoje slučajevi kada i njihovo preklapanje može biti korisno). Stoga se preklapanje ovih operatora ne preporučuje, osim u slučajevima kada za to postoje jaki razlozi. Isto tako, ne preporučuje se bez velike potrebe preklapati unarne operatore “`*`” i “`&`”, pogotovo ovog drugog (potreba za njihovim preklapanjem može nastati ukoliko želimo kreirati tipove podataka koji na neki način oponašaju ponašanje *pokazivača* odnosno *referenci*).

Kao ilustraciju do sada izloženih koncepcata, slijedi prikaz i implementacija prilično upotrebljive klase nazvane “`Kompleksni`”, koja omogućava rad sa kompleksnim brojevima. Klasa je po funkcionalnosti veoma slična generičkoj klasi “`complex`” iz istoimene biblioteke

koju smo ranije koristili, samo što je nešto siromašnija sa funkcijama i nije generička. Kako je i funkcije prijatelje klase također moguće implementirati odmah unutar deklaracije klase, bez obzira da li se radi o standardnim i operatorskim funkcijama (što je uputno raditi samo ukoliko im je tijelo kratko), to je u ovom primjeru i urađeno, izuzev operatorske funkcije za operator “>>”, čije je tijelo nešto duže:

```

class Kompleksni {
    double re, im;
public:
    Kompleksni(double re = 0, double im = 0) : re(re), im(im) {}
    friend const Kompleksni operator +(const Kompleksni &a) { return a; }
    friend const Kompleksni operator -(const Kompleksni &a) {
        return Kompleksni(-a.re, -a.im);
    }
    friend const Kompleksni operator +(const Kompleksni &a,
        const Kompleksni &b) {
        return Kompleksni(a.re + b.re, a.im + b.im);
    }
    friend const Kompleksni operator -(const Kompleksni &a,
        const Kompleksni &b) {
        return Kompleksni(a.re - b.re, a.im - b.im);
    }
    friend const Kompleksni operator *(const Kompleksni &a,
        const Kompleksni &b) {
        return Kompleksni(a.re * b.re - a.im * b.im,
            a.re * b.im + a.im * b.re);
    }
    friend const Kompleksni operator /(const Kompleksni &a,
        const Kompleksni &b) {
        double pomocna = b.re * b.re + b.im * b.im;
        return Kompleksni((a.re * b.re + a.im * b.im) / pomocna,
            (a.im * b.re - a.re * b.im) / pomocna);
    }
    friend bool operator ==(const Kompleksni &a, const Kompleksni &b) {
        return a.re == b.re || a.im == b.im;
    }
    friend bool operator !=(const Kompleksni &a, const Kompleksni &b) {
        return !(a == b);
    }
    friend Kompleksni &operator +=(Kompleksni &a, const Kompleksni &b) {
        return a = a + b;
    }
    friend Kompleksni &operator -=(Kompleksni &a, const Kompleksni &b) {
        return a = a - b;
    }
    friend Kompleksni &operator *=(Kompleksni &a, const Kompleksni &b) {
        return a = a * b;
    }
    friend Kompleksni &operator /=(Kompleksni &a, const Kompleksni &b) {
        return a = a / b;
    }
    friend ostream &operator <<(ostream &cout, const Kompleksni &a) {
        return cout << "(" << a.re << "," << a.im << ")";
    }
    friend istream &operator >>(istream &cin, Kompleksni &a);
}

```

```

friend double real(const Kompleksni &a) { return a.re; }
friend double imag(const Kompleksni &a) { return a.im; }
friend double abs(const Kompleksni &a) {
    return sqrt(a.re * a.re + a.im * a.im);
}
friend double arg(const Kompleksni &a) { return atan2(a.im, a.re); }

friend const Kompleksni conj(const Kompleksni &a) {
    return Kompleksni(a.re, -a.im);
}
friend const Kompleksni sqrt(const Kompleksni &a) {
    double rho = sqrt(abs(a)), phi = arg(a)/2;
    return Kompleksni(rho * cos(phi), rho * sin(phi));
}
};

istream &operator >>(istream &cin, Kompleksni &a) {
    char znak;
    cin >> ws;                                // "Progutaj" razmake
    if(cin.peek() != '(') {
        cin >> a.re;
        a.im = 0;
    }
    else {
        cin >> znak >> a.re >> znak;
        if(znak != ',') cin.setstate(ios::failbit);
        cin >> a.im >> znak;
        if(znak != ')') cin.setstate(ios::failbit);
    }
    return cin;
}

```

Vidimo da su u ovoj klasi podržana četiri osnovna aritmetička operatorka “+”, “-”, “*” i “/”, zatim odgovarajući pridružujući operatorki “+=”, “-=”, “*=” i “/=”, operatori poređenja “==” i “!=”, kao i operatori za ulaz i izlaz “>>” i “<<”. Pri tome je predviđeno da se kompleksni brojevi ispisuju kao parovi realnih brojeva u zagradama, razdvojeni zarezima (npr. “(2, 3)”), dok se mogu unositi bilo kao običan realni broj (u tom slučaju se podrazumijeva da je imaginarni dio jednak nuli), bilo kao par realnih brojeva unutar zagrada, razdvojen zarezom. Sve napisane operatorske funkcije su posve jednostavne, tako da ih nije potrebno posebno objašnjavati. Jedino je potrebno objasnitи ulogu konstrukcije

“`cin.setstate(ios::failbit)`” u operatorskoj funkciji za operatorku “`>>`”. Metoda “`setstate`” primijenjena na objekat ulaznog toka služi za dovođenje ulaznog toka u željeno stanje, pri čemu se željeno stanje zadaje parametrom. Tako, parametar “`ios::failbit`” označava neispravno stanje (“`ios::failbit`” je konstanta pobrojanog tipa definirana unutar klase “`ios`” u biblioteci “`iostream`”), tako da je svrha poziva “`cin.setstate(ios::failbit)`” upravo da dovedemo ulazni tok u neispravno stanje. Zašto ovo radimo? Primjetimo da se ovaj poziv vrši u slučaju kada na ulazu detektiramo znak koji se ne bi trebao da pojavi na tom mjestu (npr. kada na mjestu gdje očekujemo zatvorenu zgradu zateknemo nešto drugo). Ovim smo postigli da u slučaju da prilikom unosa poput “`cin >> a`” gdje je “`a`” promjenljiva tipa “`Kompleksni`” ulazni tok dospije u neispravno stanje u slučaju da nismo ispravno unijeli kompleksan broj. Na taj način će se operatorka “`>>`” za tip “`Kompleksni`” ponašati na isti način kao i za slučaj prostih ugrađenih tipova. Alternativno rješenje bilo bi bacanje izuzetka u slučaju neispravnog unosa, ali u tom slučaju bismo imali različit tretman grešaka pri unosu za slučaj tipa “`Kompleksni`” i prostih ugrađenih tipova.

Pored ovih operatora, definirano je i šest običnih funkcija (ne funkcija članica) “`real`”, “`imag`”, “`abs`”, “`arg`”, “`conj`” i “`sqrt`” koje respektivno vraćaju realni dio, imaginarni dio, modul, argument, konjugovano kompleksnu vrijednost i kvadratni korijen kompleksnog broja koji im se proslijedi kao argument (činjenica da funkcije “`abs`” i “`sqrt`” već postoje nije problem, jer je dozvoljeno imati funkcije istih imena koje se razlikuju po tipovima argumenata). Ovo sve zajedno čini sasvim solidnu podršku radu sa kompleksnim brojevima. Primijetimo da je unutar funkcije “`arg`” elegantno iskorištena funkcija “`atan2`” iz standardne matematičke biblioteke, koja radi upravo ono što nam ovdje treba. Ne treba zaboraviti da je u program koji implementira ovu klasu obavezno uključiti i matematičku biblioteku “`cmath`”, zbog upotrebe funkcija kao što su “`sqrt`”, “`atan2`”, “`sin`” itd.

Ukoliko bismo testirali gore napisanu klasu `Kompleksni`, mogli bismo uočiti da su automatski podržane i mješovite operacije sa realnim i kompleksnim brojevima, iako takve operatore nismo eksplicitno definirali. Na primjer, izrazi “`3 * a`” ili “`a + 2`” gdje je “`a`” objekat tipa “`Kompleksni`” sasvim su legalni. Ovo je posljedica činjenice da klasa “`Kompleksni`” ima *konstruktor sa jednim parametrom* (zapravo, ima konstruktor sa podrazumijevanim parametrima, koji se po potrebi može protumačiti i kao konstruktor sa jednim parametrom) koji omogućava automatsku pretvorbu tipa “`double`” u tip “`Kompleksni`”. Pošto se izrazi “`3 * a`” i “`a + 2`” zapravo interpretiraju kao “`operator * (3, a)`” odnosno “`operator + (a, 2)`”, konverzija koju obavlja konstruktor dovodi do toga da se ovi izrazi dalje interpretiraju kao “`operator * (Kompleksni(3), a)`” odnosno kao “`operator * (a, Kompleksni(2))`” što ujedno objašnjava zašto su ovi izrazi legalni.

Napomenimo da prilikom interpretacije izraza sa operatorima, komajler prvo pokušava pronaći operatorsku funkciju koja po tipu parametara tačno odgovara operandima upotrijebljenog operatora. Tek ukoliko se takva funkcija ne pronađe, pokušavaju se pretvorbe tipova. U slučaju da je moguće izvršiti više različitih pretvorbi, prvo se probavaju prirodnejne pretvorbe (npr. ugrađene pretvorbe smatraju se prirodnijim od korisnički definiranih pretvorbi, kao što se i pretvorbe između jednog u drugi cijelobrojni tip smatraju prirodnijim od pretvorbe nekog cijelobrojnog u neki realni tip). Na primjer, pretpostavimo da smo definirali i sljedeću operatorsku funkciju:

```
const Kompleksni operator *(double d, const Kompleksni &a) {
    return Kompleksni(d * a.r, d * a.i);
}
```

U tom slučaju, prilikom izvršavanja izraza poput “`2.5 * a`” biće direktno pozvana navedena operatorska funkcija, umjesto da se izvrši pretvorba realnog broja “`2.5`” u kompleksni i pozove operatorsku funkciju za množenje dva kompleksna broja. Ista operatorska funkcija će se pozvati i prilikom izvršavanja izraza “`3 * a`”, iako je “`3`” cijeli, a ne realni broj. Naime, pretvorba cijelog broja “`3`” u realni broj je prirodnija od pretvorbe u kompleksni broj, s obzirom da je to ugrađena, a ne korisnički definirana pretvorba. U navedenim primjerima, rezultat bi bio isti koji god da se funkcija pozove. Međutim, pravila kako se razrješavaju ovakvi pozivi neophodno je znati, s obzirom da programer ima pravo da definira različite akcije u različitim operatorskim funkcijama.

Razmotrimo sada preklapanje operatora “`++`” i “`--`”. Njihova specifičnost u odnosu na ostale unarne operatore je u tome što oni imaju i prefiksni i postfiksni oblik, npr. može se pisati “`++a`” ili “`a++`”. Prefiksni oblik ovih operatora preklapa se na isti način kao i svi ostali unarni operatori, odnosno izraz “`++a`” interpretira se kao “`operator ++(a)`”. Definirajmo, na primjer, operator “`++`” za klasu “`Vektor`” sa značenjem povećavanja svih koordinata vektora za jedinicu (korist od ovakvog operatora je diskutabilna, međutim ovdje ga samo definiramo kao primjer). Kako je za ugrađene tipove definirano da je rezultat operatora “`++`” vrijednost objekta *nakon izmjene*, učinićemo da isto vrijedi i za operator “`++`” koji definiramo za klasu “`Vektor`”. Stoga ovaj operator možemo definirati pomoću sljedeće operatorske funkcije:

```
Vektor &operator ++(Vektor &v) {
    v.x++; v.y++; v.z++;
    return v;
}
```

Čitatelju odnosno čitateljici je vjerovatno sasvim jasno zašto je formalni parametar “`v`” deklariran kao referenca na nekonstantni objekat. Također, kao rezultat je vraćena referenca na modificirani objekat, čime sprečavamo nepotrebno kopiranje. Vraćeni rezultat se mogao deklarirati i kao referenca na *konstantni objekat*, čime bismo spriječili mogućnost da se rezultat operatora “`++`” upotrijebi sa lijeve strane znaka jednakosti. Međutim, kako to nije spriječeno za proste ugrađene tipove podataka (npr. bez obzira na svoju besmislenost, izraz “`++a = 3`” je potpuno legalan u slučaju kada je “`a`” cijelobrojna promjenljiva), nismo to sprečavali ni u upravo definiranoj operatorskoj funkciji (razlog zbog kojeg su ovakve konstrukcije uopće legalne leži u činjenici da legalan i smislen izraz poput “`++(++a)`” odnosno “`++++a`” ne bi bio moguć kada izraz “`++a`” ne bi bio l-vrijednost).

Važno je napomenuti da definiranjem prefiksne verzije operatora “`++`” odnosno “`--`” nije automatski definirana i njihova postfiksna verzija. Na primjer, ukoliko je “`v`” objekat tipa “`Vektor`”, prethodna definicija učinila je izraz “`++v`” legalnim, ali izraz “`v++`” još uvijek nema smisla. Da bismo definirali i postfiksnu verziju operatora “`++`”, treba znati da se izrazi poput “`a++`” gdje je “`a`” neki korisnički definirani tip podataka interpretiraju kao “`operator ++(a, 0)`”. Drugim riječima, postfiksni operatori “`++`” i “`--`” tretiraju se kao *binarni operatori*, ali čiji je drugi operand uvijek cijeli broj “`0`”. Stoga odgovarajuća operatorska funkcija uvijek dobija nulu kao drugi parametar (opravdano, ona teoretski može dobiti i neku drugu vrijednost kao parametar, ali jedino ukoliko operatorsku funkciju eksplicitno pozovemo kao funkciju, recimo konstrukcijom poput “`operator ++(a, 5)`”, što se gotovo nikada ne čini). Zbog toga bismo postfiksnu verziju operatora “`++`” za klasu “`Vektor`” mogli definirati na sljedeći način (jezik C++ uvijek dozvoljava da izostavimo ime formalnog parametra ukoliko nam njegova vrijednost nigdje ne treba, isto kao što je dozvoljeno u prototipovima funkcija, što smo ovdje uradili sa drugim parametrom operatorske funkcije):

```
const Vektor operator ++(Vektor &v, int) {
    Vektor pomocni = v;
```

```

    v.x++; v.y++; v.z++;
    return pomocni;
}

```

Kako za ugrađene proste tipove prefiksne i postfiksne verzije operatora “`++`” i “`--`” imaju isto dejstvo na operand, a razlikuje se samo vraćena vrijednost (postfiksne verzije ovih operatorka vraćaju vrijednost operanda kakva je bila *prije izmjene*), istu funkcionalnost smo simulirali i u prikazanoj izvedbi postfiksne verzije operatorka “`++`” za klasu “`Vektor`”. Primijetimo da u ovom slučaju ne smijemo vratiti referencu kao rezultat, s obzirom da se rezultat nalazi u lokalnom objektu “`pomocni`” koji prestaje postojati po završetku funkcije (ukoliko bismo to uradili, kreirali bismo viseću referencu). Također, kao rezultat smo vratili konstantan objekat, što onemogućava njegovo korištenje sa lijeve strane jednakosti. Ovo smo uradili radi konzistencije, jer isto vrijedi i za proste ugrađene tipove (na primjer, izraz “`a++ = 3`” u slučaju kada je “`a`” cijelobrojna promjenljiva nije legalan, bez obzira što “`++a = 3`” jeste). Moramo priznati da neke konvencije jezika C++ koje vrijede za proste ugrađene tipove zaista djeluju dosta čudne, i teško je na prvi pogled dati odgovor zbog čega su uopće uvedene, ali je veoma dobra praksa očuvati ove konvencije prilikom definiranja operatorskih funkcija za korisnički definirane tipove, što smo i učinili prilikom definiranja operatorskih funkcija za tip “`Vector`”.

Operatorske funkcije se mogu definirati i za pobrojane tipove. Na primjer, neka je data sljedeća deklaracija pobrojanog tipa “`Dani`”:

```

enum Dani {Ponedjeljak, Utorka, Srijeda, Cetvrtak, Petak, Subota,
Nedjelja};

```

Kao što znamo, za ovakav tip operator “`++`” podrazumijevano nije definiran ni u prefiksnoj ni u postfiksnoj verziji, dok se sabiranje sa cijelim brojem podrazumijevano izvodi pretvorbom objekta tipa “`Dani`” u cijeli broj. Ove konvencije možemo promijeniti definiranjem vlastitih operatorskih funkcija za ove operatore nad tipom “`Dani`”, kao u primjeru koji slijedi:

```

Dani &operator ++(Dani &d) {
    if(d == Nedjelja) d = Ponedjeljak;
    else d = Dani(int(d) + 1);
    return d;
}

Dani operator ++(Dani &d, int) {
    Dani pomocni = d;
    ++d;
    return pomocni;
}

```

```

}

Dani operator +(Dani d, int n) {
    return Dani((int(d) + n) % 7);
}

```

U navedenom primjeru, definirali smo operatorske funkcije za prefiksnu i postfiksnu verziju operatora “**++**” primjenjenog nad objektom tipa “*Dani*”, kao i operatorsku funkciju za sabiranje objekta tipa “*Dani*” sa cijelom brojem. Operator “**++**” zamišljen je da djeluje tako što će transformirati objekat na koji je primijenjen da sadrži *sljedeći dan u sedmici* (uvažavajući činjenicu da iza nedjelje slijedi ponedjeljak). Implementacija postfiksne verzije operatora “**++**” oslanja se na prethodno definiranu prefiksnu verziju, u čemu nema ništa loše, osim malog gubitka na efikasnosti. Operator “**+**” zamišljen je da djeluje tako da izraz oblika “*d + n*”, gdje je “*d*” objekat tipa “*Dani*” a “*n*” cijeli broj, predstavlja dan koji se nalazi “*n*” dana ispred dana “*d*” (mogućnost sabiranja dva objekta tipa “*Dani*” nije podržana, s obzirom da je teško zamisliti šta bi takav zbir trebao da predstavlja). Obratite pažnju kako je u implementaciji ove operatorske funkcije vješt iskorišten operator “**%**”. Razumije se da se isti trik mogao iskoristiti i u definiciji operatorske funkcije za operator “**++**”, koja je mogla izgledati i recimo ovako:

```

Dani &operator ++(Dani &d) {
    return d = Dani((int(d) + 1) % 7);
}

```

Ovom prilikom smo, iz edukativnih razloga, namjerno demonstrirali drugačiji način. Također, bitno je uočiti da je eksplisitna konverzija u tip “**int**” u izrazu poput “**int**(*d*) + *n*”, koja bi se u tipičnim situacijama izvršila *automatski*, u ovom slučaju *neophodna*. Naime, izraz poput “*d + n*” u definiciji operatorske funkcije za operator “**+**” bio bi shvaćen *kao rekurzivni poziv!*

Primijetimo također da se u prikazanoj implementaciji operatorske funkcije za operator “**+**”, parametar “*d*” prenosi *po vrijednosti*. S obzirom da su objekti tipa “*Dani*” posve mali (tipično iste veličine kao i objekti tipa “**int**”), njih se *ne isplati* prenositi kao reference na konstantne objekte. Naime, njihovo kopiranje nije ništa zahtjevnije nego kreiranje reference (koja je također tipično iste veličine kao i objekti tipa “**int**”), a upotreba referenci unosi i dodatnu indirekciju. Stoga deklariranjem parametra “*d*” kao reference na konstantni objekat tipa “*Dani*” ništa ne bismo dobili na efikasnosti, već bismo naprotiv imali i neznatan gubitak.

Čitatelju i čitateljici se ostavlja kao korisna vježba da samostalno definiraju i operatorske funkcije za operatore “**--**” i “**-**” primijenjene na objekte tipa “*Dani*”. Treba još napomenuti da je mogućnost preklapanja operatora za pobrojane tipove u standard jezika C++ ušla tek nedavno, tako da su veoma rijetki programi u kojima se susreće preklapanje operatora za pobrojane tipove.

Sve do sada, operatorske funkcije smo definirali kao obične funkcije, koje nisu funkcije članice klase (mada najčešće jesu funkcije prijatelji klase). Međutim, operatorske funkcije se mogu definirati i kao funkcije članice klase. U tom slučaju, interpretacija operatorka se neznatno mijenja. Naime, neka je “ \oplus ” neki binarni operator. Već smo vidjeli da se izraz “ $x \oplus y$ ” interpretira kao “`operator $\oplus(x, y)$` ” u slučaju da je odgovarajuća operatorska funkcija definirana kao obična funkcija. Međutim, ukoliko istu operatorsku funkciju definiramo kao *funkciju članicu*, tada se izraz “ $x \oplus y$ ” interpretira kao “`x.operator $\oplus(y)$` ”, odnosno uzima se da operatorska funkcija *djeluje nad prvim operandom*, a drugi operand *prihvata kao parametar*.

Slično, ukoliko je “ \bullet ” neki unarni operator, tada se izraz “ `$\bullet x$` ” interpretira kao “`operator $\bullet(x)$` ” u slučaju da je odgovarajuća operatorska funkcija definirana kao obična funkcija, a kao “`x.operator $\bullet()$` ” u slučaju da je definirana kao funkcija članica. Za slučaj operatorka “ $++$ ” odnosno “ $--$ ”, za njihove prefiksne verzije vrijedi isto što i za sve ostale unarne operatore, dok se njihove postfiksne verzije u slučaju da su odgovarajuće operatorske funkcije definirane kao funkcije članice interpretiraju kao “`x.operator $++(0)$` ” odnosno “`x.operator $--(0)$` ” (tj. sa jednim fiktivnim cjelobrojnim argumentom).

Pogledajmo kako bi mogle izgledati definicije operatorskih funkcija za binarni operator “ $+$ ” i unarni minus za klasu “*Vektor*” ukoliko bismo se odlučili da ih implementiramo kao funkcije članice klase. Za tu svrhu bismo morali neznatno izmijeniti deklaraciju same klase, da dodamo odgovarajuće prototipove (ili čak i kompletne definicije) za operatorske funkcije članice:

```
class Vektor {
    double x, y, z;
public:
    Vektor(double x, double y, double z) : x(x), y(y), z(z) {}
    double Vrati_x() const { return x; }
    double Vrati_y() const { return y; }
    double Vrati_z() const { return z; }
    const Vektor operator +(const Vektor &v) const;
    const Vektor operator -(const Vektor &v) const;
};
```

Operatorsku funkciju članicu za unarni operator “ $-$ ” smo implementirali odmah unutar deklaracije klase, radi njene kratkoće. Ni definicija operatorske funkcije za binarni operator “ $+$ ” nije mnogo duža, ali smo se ipak odlučili da je implementiramo izvan deklaracije klase, sa ciljem da uočimo kako treba izgledati zaglavje operatorske funkcije članice klase u slučaju kada se ona implementira izvan klase (primjetimo da prva pojava riječi “*Vektor*” označava povratni tip funkcije, dok druga pojava riječi “*Vektor*” zajedno sa operatorm “`::`” označava da se radi o funkciju članici klase “*Vektor*”):

```
const Vektor Vektor::operator +(const Vektor &v) const {
    return Vektor(x + v.x, y + v.y, z + v.z);
}
```

Pri susretu sa ovako definiranom operatorskom funkcijom početnici se u prvi mah zbune na šta se odnose imena atributa poput “x”, “y” i “z” koja su upotrijebljena samostalno, bez operatora “.” koji označava pripadnost konkretnom objektu. Međutim, ne treba zaboraviti da se radi o funkciji članici (tj. metodi), koja se uvijek primjenjuje *nad nekim objektom*. Tada se ova imena odnose na imena onog konkretnog objekta nad kojim je metoda primjenjena. Na primjer, izraz “v1 + v2” gdje su “v1” i “v2” vektori, biće interpretiran kao “v1.operator +(v2)”, tako da će se “x”, “y” i “z” odnositi na atribute objekta “v1” (podsjetimo se da svaka metoda informaciju o objektu nad kojim je primjenjena dobija preko pokazivača “this”). Ovo postaje jasnije ukoliko uočimo da se ista operatorska funkcija članica mogla napisati i ovako:

```
const Vektor Vektor::operator +(const Vektor &v) const {
    return Vektor(this->x + v.x, this->y + v.y, this->z + v.z);
}
```

Još jedan način da napišemo istu funkciju članicu mogao bi izgledati ovako (pri čemu se ne navodi eksplisitno pokazivač “this”, ali se eksplisitno ističe da se “x”, “y” i “z” odnose na objekat klase “Vektor” nad kojim je funkcija članica primjenjena);

```
const Vektor Vektor::operator +(const Vektor &v) const {
    return Vektor(Vektor::x + v.x, Vektor::y + v.y, Vektor::z + v.z);
}
```

Obično je stvar programera da li će neku operatorsku funkciju definirati kao običnu funkciju ili kao funkciju članicu. Međutim, između ova dva načina ipak postoje izvjesne razlike. Da bismo uočili ovu razliku, podsjetimo se da se izraz poput “x ⊕ y”, gdje je “⊕” neki binarni operator, interpretira kao “operator ⊕(x, y)” ili kao “x.operator ⊕(y)” u ovisnosti da li je odgovarajuća operatorska funkcija obična funkcija ili funkcija članica klase. Razmotrimo sada binarni operator “+” koji smo definirali u klasi “Kompleksni”. Vidjeli smo da su zahvaljujući postojanju konstruktora sa jednim parametrom i automatskoj pretvorbi tipova koja se ostvaruje konstruktorom, izrazi poput “a + 3” i “3 + a” gdje je “a” objekat tipa “Kompleksni” potpuno legalni. Prepostavimo sada da smo umjesto kao običnu funkciju, operatorsku funkciju za operator “+” definirali kao funkciju članicu klase “Kompleksni”. Tada bi se dva prethodna izraza interpretirala kao “a.operator +(3)” i “3.operator +(a)”. Prvi izraz bi i dalje bio legalan (jer bi se on, zahvaljujući automatskoj pretvorbi tipova pri prenosu parametara u funkcije, dalje interpretirao kao “a.operator +(Kompleksni(3))”), dok bi drugi izraz bio ilegalan, jer “3” nije objekat nad kojim bi se mogla primijeniti neka metoda. Drugim riječima, izraz “3 + a” postao bi ilegalan mada bi izraz “a + 3” i dalje bio legalan!

Ovaj primjer možemo generalizirati zaključkom da kada se god neki binarni operator definira pomoću operatorske funkcije koja je funkcija članica klase, tada se na prvi operand *ne vrše nikakve automatske pretvorbe tipova*. Na taj način, definiranje operatorske funkcije kao

funkcije članice klase uvodi *izrazitu asimetriju* u način kako se tretiraju lijevi i desni operand (što je vidljivo već i iz činjenice da se operatorska funkcija tad izvodi *nad lijevim operandom*, dok se desni operand *prenosi kao parametar*). Stoga, ukoliko nam ova asimetrija nije poželjna, operatorsku funkciju treba definirati kao *običnu funkciju*, dok u slučaju da nam ova asimetrija odgovara, tada je bolje definirati operatorsku funkciju kao *funkciju članicu*. Da budemo konkretniji, za proste ugrađene tipove, većina binarnih operatora poput “+”, “-”, “*”, “/”, “%”, “<”, “<=”, “>”, “>=”, “==”, “!=”, “&”, “^”, “|”, “&&” i “||” tretira oba svoja operanda *ravnopravno*, tako da je ove operatore bolje preklapati operatorskim funkcijama definiranim kao obične funkcije (tipično prijatelje razmatrane klase). S druge strane, neki od operatora kao što su “=”, “+=”, “-=”, “*=”, “/=”, “%=” , “&=”, “^=”, “|=”, “<<=”, “>>=” su sami po sebi *izrazito asimetrični*, zbog toga što *modificiraju svoj lijevi operand* (koji zbog toga mora biti l-vrijednost) a ne kvare sadržaj svog desnog operanda. Zbog toga je ove operatore mnogo bolje preklapati operatorskim funkcijama koje su članice odgovarajuće klase (napomenimo da za operator “=” uopće nemamo izbora – odgovarajuća operatorska funkcija *mora* biti članica klase). Pogledajmo kako bi mogla izgledati definicija operatorske funkcije za operator “+=” za klasu “*Vektor*” (pri tome pretpostavljamo da smo u deklaraciji klase “*Vektor*” dodali odgovarajući prototip):

```
Vektor &Vektor::operator +=(const Vektor &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

Interesantno je razmotriti posljednju naredbu ove funkcije. Podsjetimo se da bi izraz poput “v1 += v2” trebao da vrati kao rezultat modificiranu vrijednost objekta “v1” (da bi konstrukcije poput “v3 = v1 += v2” bile moguće). Međutim, kako se ovaj izraz interpretira kao “v1.operator +=(v2)”, on bi trebao da vrati kao rezultat modificiranu vrijednost objekta nad kojim je operatorska funkcija pozvana. Ovaj objekat je jedino moguće dohvatiti preko pokazivača “this” (pokazivač “this” pokazuje na njega, tako da je “*this” sam taj objekat). Zapravo, sa ovakvom situacijom smo se susreli i ranije, u jednoj od ranijih definicija klase “*Vektor*”, u kojoj smo imali funkciju članicu “*Saberisa*”, u kojoj smo koristili istu tehniku. Ovdje operatorska funkcija za operator “+=” nije ništa drugo nego prerašena verzija metode “*Saberisa*”. Primjetimo da se zbog činjenice da imamo definiranu operatorsku funkciju za operator “+”, operatorsku funkciju za operator “+=” možemo napisati kraće kao

```
Vektor &Vektor::operator +=(const Vektor &v) {
    *this = *this + v;
    return *this;
}
```

ili još kraće kao

```

Vektor &Vektor::operator +=(const Vektor &v) {
    return *this = *this + v;
}

```

Razumije se da je zbog razloga efikasnosti bolje izvesti operatorsku funkciju za operator “`+=`” neovisno od operatorske funkcije za binarni operator “`+`”. Međutim, ukoliko već želimo iskoristiti zajednički kôd i na jednostavan način održati konzistenciju između definicija operatora “`+`” i “`+ =`”, bolje je operator “`+`” definirati preko operatora “`+ =`” nego obrnuto, ma koliko to čudno izgledalo na prvi pogled. Naime, ako prepostavimo da imamo definiranu operatorsku funkciju za operator “`+ =`” (koja se ne oslanja na definiciju operatora “`+`”), tada operatorsku funkciju za operator “`+`” možemo izvesti ovako:

```

inline const Vektor operator +(Vektor v1, const Vektor &v2) {
    return v1 += v2;
}

```

Ovakva definicija je veoma interesantna zbog činjenice da se ova operatorska funkcija uopće ne treba deklarirati kao funkcija prijatelj klase, s obzirom da nigdje ne pristupa privatnim atributima klase.

Bitno je uočiti da je u prethodnoj definiciji parametar “`v1`” prenesen *po vrijednosti*, tako da se njegova izmjena, do koje dovodi izraz “`v1 += v2`”, neće odraziti na vrijednost odgovarajućeg stvarnog parametra (tako da se u izrazu poput “`a + b`” gdje su “`a`” i “`b`” tipa “`Vektor`” vrijenost vektora “`a`” neće promijeniti, s obzirom da je formalni parametar “`v1`” samo *kopija* stvarnog parametra “`a`”, koja biva uništena odmah po završetku funkcije). Dalje, očigledno je da je vrijednost vraćena iz funkcije zaista traženi zbir “`v1 + v2`”. Konačno, kako je funkcija realizirana kao umetnuta funkcija, kompjuleru su otvorene mogućnosti za brojne optimizacije koje na kraju obično dovode do toga da generirani kôd neće biti ništa lošiji nego u slučaju kada se operatorska funkcija za operator “`+`” realizira neovisno od operatorske funkcije za operator “`+ =`”. U narednom poglavlju ćemo vidjeti da upravo ovakva realizacija može biti veoma pogodna u nekim slučajevima. Ipak, ukoliko je zahtjev za efikasnošću na prvom mjestu, najsigurnije je operatorske funkcije za operatore “`+`” i “`+ =`” realizirati potpuno neovisno jednu od druge. Isto vrijedi i za sve druge parove srodnih operatora.

Operatori “`<<`” i “`>>`” također su asimetrični po prirodi. Međutim, njih ne možemo definirati kao funkcije članice klase (bar ne ukoliko želimo da obavljaju funkciju ispisa na izlazni tok, odnosno unosa sa ulaznog toka). Naime, kako je prvi parametar operatorske funkcije za operator “`<<`” obično referenca na objekat tipa “`ostream`”, a za operator “`>>`” referenca na objekat tipa “`istream`”, slijedi da bi odgovarajuće operatorske funkcije trebale biti funkcije članice klase “`ostream`” odnosno “`istream`”, s obzirom da se operatorske funkcije članice primjenjuju nad svojim lijevim operandom. Međutim, kako nije poželjno (a često nije ni moguće, jer nam njihova implementacija najčešće nije dostupna) mijenjati definicije klase “`ostream`” i “`istream`” i dopunjivati ih novim funkcijama članicama (ove klase su napisane

sa ciljem da se *koriste*, a ne da se *mijenjaju*), kao rješenje nam ostaje korištenje klasičnih funkcija.

Što se unarnih operatora tiče, za njih je u suštini svejedno da li se definiraju preko običnih operatorskih funkcija ili operatorskih funkcija članica klase. Mnogi programeri sve unarne operatore definiraju preko operatorskih funkcija članica, jer takva praksa često vodi ka neznatno kraćoj implementaciji. Operatore “`++`” i “`--`” gotovo svi programeri definiraju preko operatorskih funkcija članica, mada za to ne postoje izraziti razlozi (vjerovatan razlog je što se oni često mogu smatrati kao specijalni slučajevi operatora “`=`” i “`-`”, koji se tipično izvode kao operatorske funkcije članice).

Mogućnost preklapanja nekih specijalnih operatora može također imati veoma interesatne primjene. Posebno je interesantna mogućnost preklapanja operatora “`[]`” i “`()`” koji se normalno koriste za pristup elementima niza, odnosno za pozivanje funkcija. Kako klase nisu niti nizovi, niti funkcije, nad objektima neke klase ne mogu se normalno primjenjivati ovi operatori. Na primjer, ukoliko je “`a`” objekat neke klase, sami po sebi izrazi “`a[5]`” odnosno “`a(3, 2)`” nemaju smisla (druga situacija bi bila da je “`a`” *niz čiji su elementi primjeri neke klase*). Međutim, česta situacija je da imamo klase koje se po svojoj strukturi ponašaju *poput nizova* (tipičan primjer su klase “`VektorN`” i “`Razred`” koje smo ranije razvijali), s obzirom da u sebi čuvaju izvjesne elemente (ovakve klase se nazivaju *kontejnerske klase*). Također, moguće je napraviti klase koje se po svojoj strukturi ponašaju *poput funkcija* (primjer bi mogla biti klasa koja bi opisivala polinom čiji se koeficijenti mogu zadavati). Preklapanje operatora “`[]`” i “`()`” omogućava da se klase koje se ponašaju kao nizovi *koriste kao da se radi o nizovima*, odnosno da se klase koje se ponašaju poput funkcija *koriste kao da se radi o funkcijama*.

Razmotrimo jedan konkretan primjer. U klasi “`VektorN`” koju smo definirali u prethodnom poglavlju imali smo funkciju članicu “`Element`” koju smo koristili za pristup elementima n-dimenzionalnog vektora. Na primjer, ukoliko je “`v`” objekat klase “`VektorN`”, koristili smo rogovatne konstrukcije poput “`v.Element(3) = 10`” ili “`cout << v.Element(2)`”. Bilo bi mnogo prirodnije kada bismo mogli pisati samo “`v[3] = 10`” ili “`cout << v[2]`”. U normalnim okolnostima ovo nije moguće, s obzirom da “`v`” nije niz. Preklapanje operatora “`[]`” će omogućiti da ovo postane moguće. Preklapanje ovog operatora sasvim je jednostavno, a izvodi se definiranjem operatorske funkcije “`operator []`” koja obavezno *mora biti funkcija članica* klase za koju se definira ovaj operator. Pri tome se izraz oblika “`x[y]`” interpretira kao

```
x.operator [](y)
```

Odavde neposredno slijedi da ukoliko želimo podržati da umjesto “`v.Element(i)`” možemo pisati prosto “`v[i]`”, sve što treba uraditi je promijeniti imena funkcija članica “`Element`” u “`operator []`”, dok njihova definicija može ostati ista. Slijedi poboljšana izvedba klase “`VektorN`”, u kojoj smo pored operatora “`[]`” usput definirali i operatore “`+`” i “`<<`” umjesto

funkcije "ZbirVektora" i metode "Ispisi", koje smo koristili u ranijim definicijama (zbog korištenja funkcije "copy", ova izvedba traži uključivanje zaglavlja biblioteke "algorithm" u program):

```

class VektorN {
    int br_elemlenata;
    double *elementi;
public:
    explicit VektorN(int n) : br_elemlenata(n), elementi(new double[n]) {}
    VektorN(const VektorN &v);
    ~VektorN() { delete[] elementi; }
    VektorN &operator =(const VektorN &v);
    friend ostream &operator <<(ostream &cout, const VektorN &v);
    double operator [](int i) const;
    double &operator [](int i);
    friend const VektorN operator +( const VektorN &v1, const VektorN &v2);
};

VektorN::VektorN(const VektorN &v) : br_elemlenata(v.br_elemlenata),
    elementi(new double[v.br_elemlenata]) {
    copy(v.elementi, v.elementi + v.br_elemlenata, elementi);
}

VektorN &VektorN::operator =(const VektorN &v) {
    if(br_elemlenata < v.br_elemlenata) {
        delete[] elementi;
        elementi = new double[v.br_elemlenata];
    }
    br_elemlenata = v.br_elemlenata;
    copy(v.elementi, v.elementi + v.br_elemlenata, elementi);
    return *this;
}

ostream &operator <<(ostream &cout, const VektorN &v) {
    cout << "{";
    for(int i = 0; i < v.br_elemlenata - 1; i++)
        cout << v.elementi[i] << ",";
    return cout << v.elementi[v.br_elemlenata - 1] << "}";
}

double VektorN::operator [](int i) const {
    if(i < 1 || i > br_elemlenata) throw "Pogrešan indeks!\n";
    return elementi[i - 1];
}

double &VektorN::operator [](int i) {
    if(i < 1 || i > br_elemlenata) throw "Pogrešan indeks!\n";
    return elementi[i - 1];
}

const VektorN operator +( const VektorN &v1, const VektorN &v2) {
    if(v1.br_elemlenata != v2.br_elemlenata)
        throw "Vektori koji se sabiraju moraju biti iste dimenzije!\n";
}

```

```

VektorN v3(v1.br_elemenata);
for(int i = 0; i < v1.br_elemenata; i++)
    v3.elementi[i] = v1.elementi[i] + v2.elementi[i];
return v3;
}

```

Jasno je da objekte ovakve klase, zbog činjenice da je definiran operator “[]” možemo koristiti kao nizove i to kao *pametne nizove* kod kojih se provjerava opseg indeksa i čiji indeksi počinju od *jedinice*. Čitatelju odnosno čitateljici neće biti teško zaključiti da je funkcioniranje tipa “vector” definiranog u istoimenoj biblioteci također zasnovano na preklapanju operatora “[]”. Napomenimo da postoje još neki izvedeni tipovi koji su definirani u standardnim bibliotekama jezika C++, kao što su tipovi “valarray” i “deque”, koji također posjeduju preklopljeni operator “[]”, tako da se ponašaju poput nizova. O ovim tipovima, koji se koriste na dosta sličan način kao i tip “vector” mada između njih ima i znatnih razlika, govorićemo u kratkim crtama u kasnijim poglavljima.

Bitno je naglasiti da kada smo za neku klasu definirali operator “[]”, izraz “x[y]” gdje je “x” objekat te klase *ne predstavlja indeksiranje* (jer “x” nije niz), mada izgleda poput indeksiranja, i obično se implementira tako da zaista indeksira neki niz kojem se pristupa preko nekog od internih atributa klase. Međutim, kao i za sve operatore, programer ima pravo da operator “[]” definira kako god želi. Na primjer, sasvim je moguće definirati ovaj operator tako da “indeks” (pod navodnicima, jer se ne radi o pravom indeksu) u zagradama uopće nije cijeli broj, nego npr. realni broj, string ili čak neki drugi objekat. Na primjer, neka smo definirali klasu “Tablica” koja čuva vrijednost neke funkcije zadane tablerarno u tačkama 1, 2, 3 itd. i neka je “T” objekat klase “Tablica”. Prirodno je definirati operator “[]” tako da izraz “T[2]” daje vrijednost funkcije u tački 2. Međutim, isto tako ima smisla definirati ovaj operator i za necjelobrojne “indekse”, tako da npr. “T[2.65]” daje procjenu vrijednosti funkcije u tački 2.65 uz izvjesne prepostavke, npr. da je funkcija približno linearna između zadanih tačaka (ovaj postupak poznat je u numeričkoj matematici kao *linearna interpolacija*). Na ovom mjestu nećemo implementirati takvu klasu, nego smo samo željeli dati ideju za šta bi se “indeksiranje realnim brojem” moglo upotrijebiti. Također, veoma je interesatna mogućnost definiranja klase kod kojih operator “[]” kao “indeks” prihvata string. Ovakve klase služe za implementaciju tzv. *asocijativnih nizova*. Primjer jedne takve klase biće ilustriran u sljedećem poglavlju.

Preklapanje operatora “()” izvodi se veoma slično kao preklapanje operatora “[]”. Njegovo preklapanje izvodi se definiranjem operatorske funkcije članice “operator ()”, a izraz oblika “x(y)” gdje je “x” objekat klase u kojoj je definirana ova operatorska funkcija interpretira se kao

```
x.operator ()(y)
```

Definiranje ovog operatora omogućava da se objekti neke klase (koji naravno nisu funkcije)

koriste *kao da su funkcije*, tj. da se nad njima koristi sintaksa koja izgleda *kao poziv funkcije*. Čemu ovo može da služi? Zamislimo da naša klasa “VektorN” treba da posluži za smještanje koeficijenata nekog polinoma. Kako su polinomi funkcije (u matematičkom smislu), prirodno bi bilo omogućiti da se tako definirani polinomi mogu koristiti kao funkcije. Na primjer, ukoliko je “v” objekat klase “VektorN” (koju bi sada bolje bilo zvati “Polinom”, ali radi jednostavnosti joj nećemo mijenjati ime), prirodno bi bilo omogućiti sintaksu oblika “v(x)” koja bi računala vrijednost polinoma čiji su koeficijenti definirani u objektu “v” u tački “x”, tj. vrijednost izraza $v_1x + v_2x^2 + \dots + v_Nx^N$, pri čemu su $\{v_1, v_2, \dots, v_N\}$ elementi vektora “v” (ovaj polinom je bez slobodnog člana jer smo uveli konvenciju da se indeksi elemenata vektora obilježavaju od jedinice, ali to je po potrebi lako izmjeniti). Ovo ćemo omogućiti upravo preklapanjem operatora “()” za klasu “VektorN”. Slijedi definicija izmijenjene klase “VektorN”, u kojoj je navedena samo implementacija ove operatorske funkcije, dok implementacije ostalih elemenata klase ostaju iste kao i u prethodnoj definiciji:

```

class VektorN {
    int br_elemenata;
    double *elementi;
public:
    explicit VektorN(int n) : br_elemenata(n), elementi(new double[n]) {}
    VektorN(const VektorN &v);
    ~VektorN() { delete[] elementi; }
    VektorN &operator =(const VektorN &v);
    friend ostream &operator <<(ostream &cout, const VektorN &v);
    double operator [](int i) const;
    double &operator [](int i);
    double operator ()(double x) const;
    friend const VektorN operator +( const VektorN &v1, const VektorN &v2);
};

double VektorN::operator ()(double x) const {
    double suma(0);
    for(int i = br_elemenata - 1; i >= 0; i--)
        suma = suma * x + elementi[i];
    return suma * x;
}

```

Ovdje smo vrijednost polinoma računali Hornerovim postupkom, što je mnogo efikasnije u odnosu na računanje zasnovano na upotrebi funkcije “pow”. Slijedi i kratka demonstracija definiranog operatorka:

```

VektorN poli(4);
poli[1] = 3; poli[2] = 4; poli[3] = 2; poli[4] = 8;
cout << poli(2);

```

Ovaj će primjer ispisati broj “166”, jer je $3 \cdot 2 + 4 \cdot 2^2 + 2 \cdot 2^3 + 8 \cdot 2^4 = 166$.

Za razliku od operatorske funkcije za operator “[]” koja isključivo prima jedan i samo jedan parametar, operatorska funkcija za operator “()” može imati *proizvoljan broj parametara*. Tako se izraz oblika `x(p, q, r ...)` u općem slučaju interpretira kao

```
x.operator () (p, q, r ...)
```

Na ovaj način je omogućeno da se objekti neke klase mogu ponašati poput funkcija sa proizvoljnim brojem parametara. Ova mogućnost je naročito korisna ukoliko želimo da definiramo klase koje predstavljaju matrice (što smo već više puta radili). Naime, prirodno je podržati mogućnosti pristupa individualnim elementima matrice, ali uz prirodnije indeksiranje (tako da indeksi redova i kolona idu od jedinice, a ne od nule) i provjeru ispravnosti indeksa. Nažalost, dosta je teško preklopiti operator “[]” da radi sa matricama onako kao što smo navikli pri radu sa dvodimenzionalnim nizovima. Naime, kao što smo već vidjeli kada smo razmatrali dvodimenzionalne nizove, izraz oblika “`x[i] [j]`” interpretira se kao “`(x[i]) [j]`”, tako da se on u slučaju kada je “`x`” objekat neke klase zapravo interpretira kao

```
(x.operator [] (i)) [j]
```

Odavde vidimo da ukoliko želimo preklopiti operator “[]” za klase tipa matrice tako da radi onako kako smo navikli pri radu sa dvodimenzionalnim nizovima, rezultat operatorske funkcije za ovaj operator mora biti nekog tipa za koji je također definiran operator “[]”. S obzirom da niti jedna funkcija ne može kao rezultat vratiti niz, ostaju nam samo dvije mogućnosti. Prva i posve jednostavna mogućnost je da operatorska funkcija za operator “[]” vrati kao rezultat *pokazivač* na traženi red matrice, tako da se drugi par uglastih zagrada primjenjuje na vraćeni pokazivač (što je, kao što već znamo, sasvim legalno). Na žalost, tada se na drugi indeks primjenjuje standardno tumačenje operatora “[]” primijenjenog na pokazivače, tako da nemamo nikakvu mogućnost da utičemo na njegovu interpretaciju (npr. nije moguće ugraditi provjeru ispravnosti drugog indeksa, niti postići da se on kreće u opsegu od jedinice a ne nule). Druga mogućnost je izvesti da operatorska funkcija za operator “[]” vrati kao rezultat *objekat neke klase koja također ima preklopljen operator “[]”*, tako da će izraz “`x[i] [j]`” biti interpretiran kao

```
(x.operator [] (i)).operator [] (j)
```

Ovo rješenje je, nažalost, dosta složeno za realizaciju (jedna mogućnost je da redove matrice ne čuvamo u običnim nizovima, nego u nekoj klasi poput “*VektorN*” koja se ponaša poput niza i ima definiran operator “[]”). Međutim, mnogo jednostavnije, elegantnije i efikasnije rješenje je umjesto operatora “[]” preklopiti operator “()” koristeći operatorsku funkciju sa *dva parametra*, što je veoma lako izvesti. Doduše, tada će se elementima neke matrice (recimo “`a`”) umjesto pomoću konstrukcije “`a[i] [j]`” pristupati pomoću konstrukcije “`a(i, j)`” što baš nije u duhu jezika C++ (već više liči na BASIC ili FORTRAN), ali je ovakva sintaksa

možda čak i ljepša i prirodnija od C++ sintakse “`a[i][j]`”. Sva tri predložena rješenja biće demonstrirana u primjerima u sljedećem poglavlju.

Primjeri klase koje imaju preklopjen operator “`()`”, i koji se zbog toga mogu koristiti poput funkcija, nazivaju se *funktori*. Funktori se ponašaju kao “pametne funkcije”, u smislu da se oni mogu koristiti i pozivati poput običnih funkcija, ali je sa njima moguće izvoditi i druge operacije, zavisno od njihove definicije. Oni se, poput primjeraka svih drugih klasa, mogu kreirati konstruktorima, uništavati destruktorma, mogu se prenositi kao parametri u funkcije, vraćati kao rezultati iz funkcija, itd. Stoga, iako nije moguće napraviti funkciju koja će kao rezultat vratiti *drugu funkciju*, sasvim je moguće kao rezultat iz funkcije vratiti *funktor*, što na kraju ima isti efekat. Recimo, moguće je napraviti funkciju “`Izvod`” koja kao svoj parametar prima funkтор koji predstavlja neku matematičku funkciju, a vraća kao rezultat funkтор koji predstavlja *njen izvod*, tako da ukoliko je “`f`” neki funkтор, izraz “`f(x)`” predstavlja vrijednost funkcije u tački “`x`”, dok izraz “`Izvod(f)(x)`” predstavlja vrijednost njenog izvoda u tački “`x`”. Mada kreiranje ovakvih funkcija nije posve jednostavno i traži dosta trikova, ono je u načelu moguće. Sasvim je jasno da su na taj način otvorene posve nove mogućnosti.

Funktori su naročito korisni za realizaciju objekata koji se ponašaju kao funkcije koje pamte stanje svog izvršavanja. Na primjer, u poglavlju u kojem smo se upoznali sa funkcijama koje vraćaju vrijednost, definirali smo funkciju “`KumulativnaSuma`” sa jednim parametrom koja vraća kao rezultat ukupnu sumu svih dotada zadanih vrijednosti njenih stvarnih argumenata. Tom prilikom, njena definicija izgledala je ovako:

```
int KumulativnaSuma(int n) {
    static int suma(0);
    return suma += n;
}
```

Kao primjer upotrebe, navedimo da će naredba

```
for(int i = 1; i <= 5; i++) cout << KumulativnaSuma(i) << endl;
```

ispisati sekvencu brojeva 1, 3, 6, 10 i 15 ($1+2=3$, $1+2+3=6$, $1+2+3+4=10$, $1+2+3+4+5=15$).

Vidimo da ova funkcija svoje stanje (dotatašnju sumu) čuva u statičkoj promjenljivoj “`suma`” unutar definicije funkcije. Pretpostavimo sada da je, iz nekog razloga, potrebno vratiti akumuliranu sumu na nulu, odnosno “resetirati” funkciju tako da ona “zaboravi” prethodno sabrane argumente i nastavi rad kao da je prvi put pozvana. Ovako, kako je funkcija napisana, to je gotovo nemoguće, jer nemamo nikakvu mogućnost pristupa promjenljivoj “`suma`” izvan same funkcije. Iskreno rečeno, to ipak nije posve nemoguće, jer će konstrukcija

```
KumulativnaSuma (-KumulativnaSuma (0)) ;
```

ostvariti traženi cilj (razmislite zašto). Međutim, ovo je očigledno samo prljav trik, a ne neko univerzalno rješenje. Moguće univerzalno rješenje moglo bi se zasnovati na definiranju promjenljive “*suma*” kao globalne promjenljive, tako da bismo resetiranje funkcije uvijek mogli ostvariti dodjelom poput “*suma = 0*”. Nakon svega što smo naučili o konceptima enkapsulacije i sakrivanja informacija, suvišno je i govoriti kojiko je takvo rješenje loše. Pravo rješenje je definirati odgovarajući *funktor* koji će posjedovati i metodu nazvanu recimo “*Resetiraj*”, koja će vršiti njegovo resetiranje (tj. vraćanje akumulirane sume na nulu). Za tu svrhu, možemo definirati funktorsku klasu poput sljedeće:

```
class Akumulator {  
    int suma;  
  
public:  
    Akumulator() : suma(0) {}  
    void Resetiraj() { suma = 0; }  
    int operator ()(int n) { return suma += n; }  
};
```

Nakon toga, traženi funkтор možemo deklarirati prosto kao instancu ove klase (za njegovo imenovanje iskoristili smo mala slova, da naglasimo da se ipak radi o *objektu*, a ne o *funkciji*):

```
Akumulator kumulativna_suma;
```

Slijedi i jednostavan primjer upotrebe:

```
for(int i = 1; i <= 5; i++) cout << kumulativna_suma(i) << endl;  
kumulativna_suma.Resetiraj();  
for(int i = 1; i <= 3; i++) cout << kumulativna_suma(i) << endl;
```

Ova sekvenca naredbi ispisće slijed brojeva 1, 3, 6, 10, 15, 1, 3, 6 (bez poziva metode “*Resetiraj*” ispisani slijed bio bi 1, 3, 6, 10, 15, 16, 19, 25).

Treba napomenuti da sve funkcije iz standardne biblioteke “*algorithm*”, koje kao parametre prihvataju pokazivače na funkcije, također prihvataju i funktoare odgovarajućeg tipa, što omogućava znatno veću fleksibilnost. Ista biblioteka također sadrži nekoliko jednostavnih funktorskih objekata koji obavljaju neke standardne operacije. Tako je, na primjer, definirana generička funktorska klasa “*greater*”, koja je definirana otprilike ovako:

```

template <typename UporediviTip>
class greater {
    bool operator() (UporediviTip x, UporediviTip y) { return x > y; }
}

```

Na primjer, ukoliko izvršimo deklaraciju

```
greater<int> veci;
```

tada će izraz “`veci(x, y)`” za cijelobrojne argumente “`x`” i “`y`” imati isto značenje kao izraz “`x > y`”. Isto značenje ima i izraz poput “`greater<int>() (x, y)`”. Naime, izraz “`greater<int>()`” predstavlja poziv podrazumijevanog konstruktora za klasu “`greater<int>`” koji kreira bezimeni primjerak te klase, na koji se dalje primjenjuje operatorska funkcija za operator “`()`”. Sve je ovo lijepo, ali čemu služe ovakve komplikacije? One pojednostavljaju korištenje mnogih funkcija iz biblioteke “`algorithm`”. Na primjer, ukoliko želimo sortirati niz “`niz`” od 100 realnih brojeva u *rastući perekopak*, to možemo izvesti prostim pozivom

```
sort(niz, niz + 100, greater<double>());
```

bez ikakve potrebe da samostalno definiramo odgovarajuću funkciju kriterija (napomenimo još jednom da konstrukcija “`greater<double>()`” kreira odgovarajući bezimeni funktor, koji se dalje prosljeđuje kao parametar u funkciju “`sort`”). Pored generičke funktorske klase “`greater`”, postoje i generičke funktorske klase “`less`”, “`equal_to`”, “`not_equal_to`”, “`greater_equal`” i “`less_equal`”, koje respektivno odgovaraju relacionim operatorima “`<`”, “`=`”, “`!=`”, “`>=`” i “`<=`”, zatim generičke funktorske klase “`plus`”, “`minus`”, “`multiplies`”, “`divides`”, “`modulus`”, “`logical_and`” i “`logical_or`”, koje respektivno odgovaraju binarnim operatorima “`+`”, “`-`”, “`*`”, “`/`”, “`%`”, “`&&`” i “`||`”, i konačno, generičke funktorske klase “`negate`” i “`logical_not`”. Kao primjer primjene ovih funktorskih klasa, navedimo da naredba

```
transform(niz1, niz1 + 10, niz2, niz3, multiplies<int>());
```

prepisuje u niz “`niz3`” produkte odgovarajućih elemenata iz nizova “`niz1`” i “`niz2`” (prepostavljujući da su sva tri niza nizovi od 10 cijelih brojeva), bez ikakve potrebe da definiramo odgovarajuću funkciju koju prosljeđujemo kao parametar (kao što smo radili kada smo se prvi put susreli sa funkcijom “`transform`”). Na sličan način se mogu iskoristiti i ostale funktorske klase.

O preklapanju operatora “`=`” već smo govorili, i vidjeli smo da se izraz oblika “`x = y`”, u

slučaju da je definirana odgovarajuća operatorska funkcija, interpretira kao “`x.operator = (y)`”. Tom prilikom smo rekli da takva operatorska funkcija za neku klasu gotovo po pravilu prima parametar koji je tipa reference na konstantni objekat te klase, i kao rezultat vraća referencu na objekat te klase. Međutim, postoje situacije u kojima je korisno definirati i operatorske funkcije za operator “`=`” koje prihvataju parametre drugačijeg tipa. Na primjer, zamislimo sa su objekti “`x`” i “`y`” različitog tipa, i neka je objekat “`x`” tipa “`Tip1`”, a objekat “`y`” tipa “`Tip2`”. Podrazumijevana interpretacija izraza “`x = y`” je da se pokuša pretvorba objekta “`y`” u tip “`Tip1`” (recimo, pomoću neeksplicitnog konstruktora sa jednim parametrom klase “`Tip1`”), a da se zatim izvrši dodjela između objekata jednakog tipa. Međutim, ukoliko postoji operatorska funkcija za operator “`=`” unutar klase “`Tip1`” koja prihvata parametar tipa “`Tip2`”, ona će odmah biti iskorištena za realizaciju dodjele “`x = y`”, bez pokušaja pretvorbe tipa. Na taj način je moguće realizirati dodjelu između objekata različitih tipova čak i u slučajevima kada pretvorba jednog u drugi tip nije uopće podržana. Također, direktna implementacija dodjele između objekata različitog tipa može biti osjetno efikasnija nego indirektna dodjela koja se izvodi prvo pretvorbom objekta jednog tipa u drugi, a zatim dodjelom između objekata jednakog tipa. Ovo je još jedan razlog kada može biti korisno preklopiti operator “`=`” za objekte različitih tipova. Razumije se da, kao i za sve ostale operatore, programer ima pravo definirati operatorsku funkciju za operator “`=`” da radi šta god mu je želja, ali nije nimalo mudra ideja davati ovom operatoru značenje koje u suštini nema nikakve veze sa dodjeljivanjem.

Ovim smo objasnili postupak preklapanja svih operatora koji se mogu preklopiti osim operatora “`->`” i operatora “`new`”, “`new[]`”, “`delete`” i “`delete[]`”. Operator “`->`” preklapa se operatorskom funkcijom članicom “`operator ->`” bez parametara, pri čemu se izraz oblika “`x->y`” interpretira kao

`(x.operator->())->y`

Preklapanje ovog operatora omogućava korisniku da kreira objekte koje se ponašaju *poput pokazivača na primjerke drugih struktura ili klasa*. Takvi objekti nazivaju se *iteratori*, i o njima ćemo govoriti kasnije. S druge strane, preklapanje operatora “`new`”, “`new[]`”, “`delete`” i “`delete[]`” omogućava korisniku da kreira vlastiti mehanizam za dinamičko alociranje memorije (npr. umjesto u radnoj memoriji, moguće je alocirati objekte u datotekama na disku). Kako je za efikasnu primjenu preklapanja ovih operatora potrebno mnogo veće znanje o samom mehanizmu dinamičke alokacije memorije nego što je dosada prezentirano, postupak preklapanja ovih operatora izlazi izvan okvira ovog teksta.

Postoji još jedna vrsta operatorskih funkcija članica, koje se koriste za konverziju (pretvorbu) tipova (slično konstruktorima sa jednim parametrom). Naime pretpostavimo da imamo dva tipa “`Tip1`” i “`Tip2`”, od kojih je “`Tip1`” neka klasa. Ukoliko “`Tip1`” posjeduje konstruktor sa jednim parametrom tipa “`Tip2`”, tada se taj konstruktor koristi za konverziju objekata tipa “`Tip2`” u tip “`Tip1`” (osim ako je konstruktor deklariran sa ključnom riječi “`explicit`”). Međutim, kako definirati konverziju objekata “`Tip1`” u tip “`Tip2`”? Ukoliko je i “`Tip2`” neka klasa, dovoljno je definirati njen konstruktor sa jednim parametrom tipa “`Tip1`”. Problemi

nastaju ukoliko tip “Tip2” nije klasa (nego npr. neki prosti ugradeni tip poput “**double**”). Da bi se mogao definirati i postupak konverzije u *proste ugrađene tipove*, uvode se *operatorske funkcije za konverziju tipa*. One se isto pišu kao funkcije članice neke klase, a omogućavaju pretvorbu objekata te klase u bilo koji drugi tip, koji tipično nije klasa. Operatorske funkcije pišu se tako da se ime tipa u koji se vrši pretvorba piše *iza ključne riječi “operator”*, a povratni tip funkcije se *ne navodi*, nego se podrazumijeva da on mora biti onog tipa kao tip u koji se vrši pretvorba. Pored toga, operatorske funkcije za konverziju tipa *nemaju parametre*. Slijedi primjer kako bismo mogli deklarirati i implementirati operatorsku funkciju za pretvorbu tipa “VektorN” u tip “**double**”, tako da se kao rezultat pretvorbe vektora u realni broj dobija njegova dužina (razumije se da je ovakva pretvorba upitna sa aspekta matematske korektnosti):

```
class VektorN {
    ...
public:
    ...
    operator double() const;
};

Vektor::operator double() const {
    double suma(0);
    for(int i = 0; i < br_elemana; i++)
        suma += elementi[i] * elementi[i];
    return sqrt(suma);
}
```

Ovim postaju moguće sljedeće stvari: eksplicitna konverzija objekata tipa “VektorN” u tip “**double**” pomoću operadora za pretvorbu tipa (type-castinga), zatim neposredna dodjela promjenljive ili izraza tipa “VektorN” promjenljivoj tipa “**double**”, kao i upotreba objekata tipa “VektorN” u izrazima na mjestima gdje bi se normalno očekivao operand tipa “**double**” (pri tome bi u sva tri slučaja vektor prvo bitno bio pretvoren u realni broj). Na primjer, ukoliko je “v” promjenljiva tipa “VektorN” a “d” promjenljiva tipa “**double**”, sljedeće konstrukcije su sasvim legalne:

```
cout << (double)v;
cout << double(v);
d = v;
cout << sin(v);
```

Napomenimo da je funkcionska notacija oblika “**double**(v)”, kao i do sada, moguća samo za slučaj tipova čije se ime sastoji od *jedne riječi*, tako da ne dolazi u obzir za tipove poput “**long int**”, “**char ***” itd.

Operatorske funkcije za konverziju klasnih tipova u proste ugrađene tipove definitivno mogu biti korisne, ali sa njima ne treba pretjerivati, jer ovakve konverzije tipično vode ka gubljenju informacija (npr. zamjenom vektora njegovom dužinom gube se sve informacije o njegovom položaju). Također, previše definiranih konverzija može da zbuni kompjajler. Recimo, ukoliko su podržane konverzije objekata tipa "Tip1" u objekte tipa "Tip2" i "Tip3", nastaje nejasnoća ukoliko se neki objekat tipa "Tip1" (recimo "x") upotrijebi u nekom kontekstu u kojem se ravnopravno mogu upotrijebiti kako objekti tipa "Tip2", tako i objekti tipa "Tip3". U takvom slučaju, kompjajler ne može razriješiti situaciju, i prijavljuje se greška. Do greške neće doći jedino ukoliko eksplicitno ne specificiramo koju konverziju želimo, konstrukcijama poput " $(Tip2)x$ " ili " $(Tip3)x$ ". Stoga, kada god definiramo korisnički definirane pretvorbe tipova, moramo paziti da predvidimo sve situacije koje mogu nastupiti. Odavde neposredno slijedi da od previše definiranih konverzija tipova često može biti više štete nego koristi.

Naročit oprez je potreban kada imamo međusobne konverzije između dva tipa. Prepostavimo, na primjer, da su "Tip1" i "Tip2" dva tipa, i da je definirana konverzija iz tipa "Tip1" u tip "Tip2", kao i konverzija iz tipa "Tip2" u tip "Tip1" (nebitno je da li su ove konverzije ostvarene konstruktorom ili operatorskim funkcijama za konverziju tipa). Prepostavimo dalje da su za objekte tipova "Tip1" i "Tip2" definirani operatori za sabiranje. Postavlja se pitanje kako treba interpretirati izraz oblika " $x + y$ " gdje je " x " objekat tipa "Tip1" a " y " objekat tipa "Tip2" (ili obrnuto). Da li treba objekat " x " pretvoriti u objekat tipa "Tip2" pa obaviti sabiranje objekata tipa "Tip2", ili treba objekat " y " pretvoriti u objekat tipa "Tip1" pa obaviti sabiranje objekata tipa "Tip1"? Kompajler nema načina da razriješi ovu dilemu, tako da će biti prijavljena greška. U ovom slučaju bismo morali eksplicitno naglasiti šta želimo izrazima poput " $x + (Tip1)y$ " ili " $(Tip2)x + y$ ". Na primjer, klasa "Kompleksni" koju smo ranije definirali omogućava automatsku pretvorbu realnih brojeva u kompleksne. Kada bismo definirali i automatsku pretvorbu kompleksnih brojeva u realne (npr. uzimanjem modula kompleksnih brojeva) ukinuli bismo neposrednu mogućnost sabiranja kompleksnih brojeva sa realnim brojevima i obrnuto, jer bi upravo nastala situacija poput maločas opisane.

Treba naglasiti da do opisane dileme ipak neće doći ukoliko postoji i operatorska funkcija za operator "+" koja eksplicitno prihvata parametre tipa "Tip1" i "Tip2" (npr. jedan kompleksni i jedan realni broj). Tada će se, pri izvršavanju izraza " $x + y$ " ova operatorska funkcija pozvati prije nego što se pokuša ijedna pretvorba tipa (s obzirom da se pretvorbe provode samo ukoliko se ne nađe operatorska funkcija čiji tipovi parametara tačno odgovaraju operandima). Međutim, tada treba definirati i operatorsku funkciju koja prihvata redom parametre tipa "Tip2" i "Tip1", ako želimo da i izraz " $y + x$ " radi ispravno! Slijedi da prisustvo velikog broja korisnički definiranih pretvorbi tipova drastično povećava broj drugih operatorskih funkcija koje treba definirati da bismo imali konzistentno ponašanje.

34. Primjeri razvoja korisničkih tipova podataka

Klase kao složeni tipovi podataka koji podržavaju mehanizme enkapsulacije i sakrivanja informacija, predstavljaju idealno sredstvo za razvoj apstraktnih korisničkih tipova podataka, za koje je kada se jednom razviju, bitno samo *kako se koriste*, a ne i kako su implementirani. Upotreba primjeraka pojedinih klasa je dodatno olakšana podrškom koju pruža mogućnost preklapanja operatora. Tako smo do sada u više navrata koristili tipove podataka poput "complex", "string" i "vector", ne ulazeći u njihovu implementaciju. Implementacioni detalji bitni su samo za projektanta klase, a ne i za korisnika klase. Međutim, do sada smo se u više navrata također stavljali upravo u ulogu projektanta klase, i demonstrirali kako se neke od mogućnosti koje pružaju tipovi podataka definirani u standardnim bibliotekama jezika C++ mogu implementirati. Nakon što smo savladali sve neophodne alate za tu svrhu, u ovom poglavlju ćemo rezimirati stečena znanja kroz primjere razvoja nekoliko tipičnih korisničkih tipova podataka.

Kao prvi primjer, definiraćemo klasu "Vektor" koja predstavlja trodimenzionalni vektor, odnosno vektor u prostoru. Ovu klasu smo, u raznim ogoljenim varijantama, više puta definirali kao primjer koncepta o kojima smo govorili. Na ovom mjestu, dajemo kompletну definiciju klase "Vektor", u kojoj su definirani svi elementi i operatori koji bi za objekte ove klase mogli imati smisla (operator "%" predstavlja vektorski proizvod):

```
class Vektor {
    double x, y, z;
public:
    Vektor() : x(0), y(0), z(0) {}
    Vektor(double x, double y, double z) : x(x), y(y), z(z) {}
    Vektor(const double (&niz)[3]) : x(niz[0]), y(niz[1]), z(niz[2]) {}
    operator double() const { return sqrt(x * x + y * y + z * z); }
    double operator[](int i) const;
    double &operator[](int i);
    const Vektor operator +() const { return *this; }
    const Vektor operator -() const { return Vektor(-x, -y, -z); }
    friend const Vektor operator +(const Vektor &v1, const Vektor &v2) {
        return Vektor(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z);
    }
    friend const Vektor operator -(const Vektor &v1, const Vektor &v2) {
        return Vektor(v1.x - v2.x, v1.y - v2.y, v1.z - v2.z);
    }
    friend const Vektor operator *(const Vektor &v, double d) {
        return Vektor(v.x * d, v.y * d, v.z * d);
    }
    friend const Vektor operator *(double d, const Vektor &v) {
        return v * d;
    }
    friend double operator *(const Vektor &v1, const Vektor &v2) {
        return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
    }
    friend const Vektor operator %(const Vektor &v1, const Vektor &v2) {
        return Vektor(v1.y * v2.z - v1.z * v2.y,
```

```

        v1.z * v2.x - v1.x * v2.z, v1.x * v2.y - v1.y * v2.x);
    }

friend bool operator ==(const Vektor &v1, const Vektor &v2) {
    return v1.x == v2.x && v1.y == v2.y && v1.z == v2.z;
}

friend bool operator !=(const Vektor &v1, const Vektor &v2) {
    return !(v1 == v2);
}

Vektor &operator +=(const Vektor &v) { return *this = *this + v; }
Vektor &operator -=(const Vektor &v) { return *this = *this - v; }
Vektor &operator *=(double d) { return *this = *this * d; }
friend ostream &operator <<(ostream &cout, const Vektor &v) {
    return cout << "{" << v.x << "," << v.y << "," << v.z << "}";
}

friend istream &operator >>(istream &cin, Vektor &v) {
    return cin >> v.x >> v.y >> v.z;
}
};

double Vektor::operator[](int i) const {
    if(i == 1) return x;
    else if(i == 2) return y;
    else if(i == 3) return z;
    else throw "Neispravan indeks!\n";
}

double &Vektor::operator[](int i) {
    if(i == 1) return x;
    else if(i == 2) return y;
    else if(i == 3) return z;
    else throw "Neispravan indeks!\n";
}
}

```

U ovoj klasi se uglavnom ne pojavljuje ništa novo o čemu nismo do sada detaljno govorili, tako da nema potrebe da previše detaljno razmatramo ovu klasu. S obzirom da se unutar klase ne koristi dinamička alokacija memorije, nema potrebe za definiranjem destruktora, konstruktora kopije i prekloppljenog operatorka dodjele, već se možemo osloniti na podrazumijevano ponašanje prilikom kopiranja i dodjele. Konstruktor bez parametara omogućava kreiranje vektora čije su koordinate inicijalizirane na nulu, dok konstruktor sa tri parametra omogućava kreiranje vektora čije su koordinate inicijalizirane na zadane vrijednosti. Primijetimo da to nije isto kao da imamo konstruktor sa tri parametra čije su podrazumijevane vrijednosti nule (u tom slučaju, vektor bi se mogao inicijalizirati bez parametara, zatim sa jednim, dva ili tri parametra). Posebno je interesantan konstruktor čiji je parametar referenca na niz od tri elementa, koji omogućava da se prilikom deklaracije objekata tipa "Vektor" oni inicijaliziraju običnim nizom od tri realna broja. Drugim riječima, ovaj konstruktor omogućava konstrukcije oblika:

```

double a[3] = {5, 6, 8};
Vektor v = a;

```

Također, zahvaljujući ovom konstruktoru i automatskoj pretvorbi tipova koju on definira,

svaka funkcija koja kao parametar očekuje objekat tipa “Vektor” odnosno operator koji kao svoj operand očekuje objekat tipa “Vektor”, moći će kao parametar odnosno operand primiti i obični niz od tri realna broja. Može djelovati čudno zbog čega je parametar konstruktora deklariran kao referenca na niz od tri elementa, a ne kao obični niz od tri elementa. Ovdje je iskorišteno svojstvo da se referenca na niz može vezati samo na niz sa istim brojem elemenata kao u deklaraciji reference, tako da konstruktor neće primiti kao stvarni parametar niz sa drugačijim brojem elemenata, što bi bilo dozvoljeno u slučaju da formalni parametar nije referenca (u skladu sa načinom kako se tretiraju parametri nizovnog tipa).

Interesantno je uočiti definiciju preklopjenog operatora “[]” koja omogućava da komponentama nekog vektora (npr. “v”) pristupamo preko indeksa korištenjem izraza poput “v[1]”, “v[2]” i “v[3]”. Ovo je interesantan primjer kako se od korisnika klase može sakriti njena interna implementacija, što i jeste jedan od osnovnih ciljeva objektno orijentiranog programiranja. Naime, definirana klasa uopće ne sadrži atribute nizovnog tipa, a definicija operatora “[]” omogućava njeno korištenje kao da se radi o nizu od tri elementa.

Sljedeća interesantna klasa koju ćemo implementirati je klasa nazvana “Vrijeme”, koja je namijenjena za čuvanje informacija o vremenskim trenucima predstavljenim u formi sati, minuta i sekundi:

```
class Vrijeme {
    int h, m, s;
public:
    Vrijeme() : h(0), m(0), s(0) {}
    Vrijeme(int h, int m, int s);
    operator long int() const { return 3600L * h + 60L * m + s; }
    Vrijeme &operator ++() { return *this += 1; }
    Vrijeme &operator --() { return *this -= 1; }
    Vrijeme operator ++(int) {
        Vrijeme pomocna = *this; *this += 1; return pomocna;
    }
    Vrijeme operator --(int) {
        Vrijeme pomocna = *this; *this -= 1; return pomocna;
    }
    friend const Vrijeme operator +(const Vrijeme &v, int n);
    friend const Vrijeme operator -(const Vrijeme &v, int n) {
        return v - n;
    }
    friend long int operator -(const Vrijeme &v1, const Vrijeme &v2) {
        return (long int)v1 - (long int)v2;
    }
    Vrijeme &operator +=(int n) { return *this = *this + n; }
    Vrijeme &operator -=(int n) { return *this = *this - n; }
    friend bool operator ==(const Vrijeme &v1, const Vrijeme &v2) {
        return (long int)v1 == (long int)v2;
    }
    friend bool operator !=(const Vrijeme &v1, const Vrijeme &v2) {
        return (long int)v1 != (long int)v2;
    }
```

```

friend bool operator <(const Vrijeme &v1, const Vrijeme &v2) {
    return (long int)v1 < (long int)v2;
}
friend bool operator >(const Vrijeme &v1, const Vrijeme &v2) {
    return (long int)v1 > (long int)v2;
}
friend bool operator <=(const Vrijeme &v1, const Vrijeme &v2) {
    return (long int)v1 <= (long int)v2;
}
friend bool operator >=(const Vrijeme &v1, const Vrijeme &v2) {
    return (long int)v1 >= (long int)v2;
}
friend ostream &operator <<(ostream &cout, const Vrijeme &s);
};

Vrijeme::Vrijeme(int h, int m, int s) {
    if(h < 0 || h > 23 || m < 0 || m > 59 || s < 0 || s > 59)
        throw "Neispravno vrijeme!\n";
    Vrijeme::h = h; Vrijeme::m = m; Vrijeme::s = s;
}

ostream &operator <<(ostream &cout, const Vrijeme &s) {
    if(s.h < 10) cout << 0;
    cout << s.h << ":";
    if(s.m < 10) cout << 0;
    cout << s.m << ":";
    if(s.s < 10) cout << 0;
    return cout << s.s;
}
const Vrijeme operator +(const Vrijeme &s, int n) {
    long int p = (long int)s + n;
    return Vrijeme((p / 3600) % 24, (p % 3600) / 60, (p % 3600) % 60);
}

```

Konstruktor klase bez parametara kreira objekat tipa tipa “Vrijeme” inicijaliziran na “00:00:00”, dok konstruktor sa tri parametra omogućava inicijalizaciju na zadani broj sati, minuta i sekundi, pri čemu se provjerava ispravnost postavljenih vrijednosti. Operatorska funkcija za pretvorbu u tip “**long int**” omogućava pretvorbu zapisa o vremenu u veliki cijeli broj, pri čemu je rezultat pretvorbe to isto vrijeme izraženo samo u sekundama. Sufiks “L” iza konstanti “3600” i “60” tretira ove konstante kao cijele brojeve proširenog opsega (tj. tipa “**long int**”), čime se sprečava mogućnost prekoračenja opsega na kompjajlerima kod kojih tip “**int**” ima premali opseg dozvoljenih vrijednosti.

Operatorske funkcije za operatore “**++**” i “**--**” omogućavaju povećanje odnosno smanjenje zapisanog vremena za 1 sekundu, pri čemu se izvedbe ovih operatora oslanjaju na izvedbe operatora “**=**” i “**-=**” koji su definirani nešto kasnije. Podržane su kako prefiksne, tako i postfiksne verzije ovih operatora. Binarni operator “**+**” omogućava da objekat tipa “Vrijeme” saberemo sa cijelim brojem (npr. *n*), pri čemu kao rezultat dobijamo vrijeme koje za *n* sekundi prednjači ispred vremena zapisanog u objektu. Implementacija ove operatorske funkcije prvo pretvara vrijeme u broj sekundi, zatim ga sabira sa *n*, i konačno, zbir ponovo razlaže na sate, minute i sekunde, a rezultirajući objekat tvori pozivom konstruktora. Slično dejstvo ima i binarni operator “**-**”, samo što je njegov rezultat vrijeme koje za *n* sekundi kasni iza vremena

zapisanog u objektu “Vrijeme”. Implementacija ove operatorske funkcije svodi se na sabiranje sa $-n$. Pored toga, podržano je i međusobno oduzimanje dva objekta tipa “Vrijeme”, pri čemu se kao rezultat dobija broj sekundi između dva navedena vremena. Operatori “ $+=$ ” i “ $-=$ ” uvedeni su da omoguće izraze tipa “ $v += n$ ” i “ $v -= n$ ” kao alternativu izrazima “ $v = v + n$ ” i “ $v = v - n$ ”. Definicija relacionih operatora “ $==$ ”, “ $!=$ ”, “ $<$ ”, “ $>$ ”, “ $<=$ ” i “ $>=$ ” omogućava poređenje dva vremena u hronološkom poretku. Njihova implementacija je trivijalna, i svodi se na poređenje zapisa ovih vremena izraženih samo brojem sekundi. Konačno, operator “ $<<$ ” omogućava ispis vremena u formatu $hh:mm:ss$. Njegova implementacija bi se mogla pojednostaviti upotrebom manipulatora “`setw`” i “`setfill`”, što citatelj odnosno čitateljica mogu uratiti kao vježbu.

Demonstrirajmo sada upotrebu napisane klase “Vrijeme” na jednom jednostavnom primjeru. Prepostavimo da smo izvršili sljedeću sekvencu naredbi:

```
Vrijeme s1(9, 25, 38), s2(13, 44, 6), s3;
cout << s1 << " " << s2 << " " << s3 << endl;
if(s1 < s2) cout << "Prvo vrijeme je ispred drugog!\n";
else if(s1 > s2) cout << "Prvo vrijeme je iza drugog!\n";
s1++; s2--;
cout << s1 << " " << s2 << endl;
cout << s1 + 10 << " " << s2 - 10 << " ";
s3 = s1 + 60;
cout << s3 << endl;
s1 += 3600; s2 -= 3600;
cout << s1 << " " << s2 << endl;
s3 = Vrijeme(10, 7, 26);
cout << s3 << " " << (long int)s3 << " ";
cout << s2 - s1 << endl;
```

Ove naredbe će dovesti do sljedećeg prikaza na ekranu:

```
09:25:38 13:44:06 00:00:00
Prvo vrijeme je ispred drugog!
09:25:39 13:44:05
09:25:49 13:43:55 09:26:39
10:25:39 12:44:05
10:07:26 36446 8306
```

U sljedećem primjeru ćemo razviti klasu “String” (sa velikim početnim slovom), koja se ponaša veoma slično poput standardne klase “string” definirane u istoimenoj biblioteci, samo uz izmjenjena imena metoda i neke druge sitnije izmjene. Na taj način ćemo stići uvid u to kako objekti tipa “string” zapravo funkcioniraju. Ova klasa je nešto složenije izvedbe, ali ilustrira većinu tehnika o kojima smo dosada govorili, uključujući i tehnike za rukovanje dinamički alociranom memorijom. Kao i do sada, prvo ćemo prikazati deklaraciju same klase i njenu implementaciju, nakon čega slijedi detaljno objašnjenje implementacije:

```

class String {
    int br_znakova, kapacitet;
    char *sadrzaj;
public:
    explicit String(int n = 0) : br_znakova(0), kapacitet(n),
        sadrzaj(new char[n + 1]) { sadrzaj[0] = 0; }
    String(const char s[]);
    String(const String &s);
    ~String() { delete[] sadrzaj; }
    String &operator =(const String &s);
    friend ostream &operator <<(ostream &cout, const String &s) {
        return cout << s.sadrzaj;
    }
    friend istream &operator >>(istream &cin, String &s);
    friend istream &getline(istream &cin, String &s);
    char operator [](int i) const;
    char &operator [](int i);
    int Duzina() const { return br_znakova; }
    friend const String operator +(const String &s1, const String &s2);
    String &operator +=(const String &s);
    const String operator ()(int i, int j) const;
    operator const char *() { return sadrzaj; }
    friend bool operator ==(const String &s1, const String &s2) {
        return strcmp(s1.sadrzaj, s2.sadrzaj) == 0;
    };
    friend bool operator <(const String &s1, const String &s2) {
        return strcmp(s1.sadrzaj, s2.sadrzaj) < 0;
    }
    friend bool operator >(const String &s1, const String &s2) {
        return strcmp(s1.sadrzaj, s2.sadrzaj) > 0;
    }
    friend bool operator !=(const String &s1, const String &s2) {
        return !(s1 == s2);
    }
    friend bool operator <=(const String &s1, const String &s2) {
        return !(s1 > s2);
    }
    friend bool operator >=(const String &s1, const String &s2) {
        return !(s1 < s2);
    }
};

String::String(const char s[]) {
    kapacitet = br_znakova = strlen(s);
    sadrzaj = new char[br_znakova + 1];
    strcpy(sadrzaj, s);
}

String::String(const String &s) : br_znakova(s.br_znakova),
    kapacitet(s.br_znakova), sadrzaj(new char[s.br_znakova + 1]) {
    strcpy(sadrzaj, s.sadrzaj);
}

String &String::operator =(const String &s) {
    br_znakova = s.br_znakova;
    if(kapacitet < br_znakova) {
        delete[] sadrzaj;
}

```

```

        kapacitet = br_znakova;
        sadrzaj = new char[br_znakova + 1];
    }
    strcpy(sadrzaj, s.sadrzaj);
    return *this;
}

char String::operator [](int i) const {
    if(i < 0 || i >= br_znakova) throw "Neispravan indeks!\n";
    return sadrzaj[i];
}

char &String::operator [](int i) {
    if(i < 0 || i >= br_znakova) throw "Neispravan indeks!\n";
    return sadrzaj[i];
}

const String String::operator ()(int i, int j) const {
    if(i < 0 || i >= br_znakova || j < 0 || j >= br_znakova || i > j)
        throw "Neispravan opseg!\n";
    String s(j - i + 1);
    s.br_znakova = j - i + 1;
    copy(sadrzaj + i, sadrzaj + j + 1, s.sadrzaj);
    s.sadrzaj[s.br_znakova] = 0;
    return s;
}

const String operator +(const String &s1, const String &s2) {
    String s3(s1.br_znakova + s2.br_znakova);
    strcpy(s3.sadrzaj, s1.sadrzaj);
    strcat(s3.sadrzaj, s2.sadrzaj);
    s3.br_znakova = s1.br_znakova + s2.br_znakova;
    return s3;
}

String &String::operator +=(const String &s) {
    if(kapacitet >= br_znakova + s.br_znakova) {
        br_znakova += s.br_znakova;
        strcat(sadrzaj, s.sadrzaj);
    }
    else *this = *this + s;
    return *this;
}

istream &operator >>(istream &cin, String &s) {
    char privremeni[1000];
    cin.width(sizeof privremeni);
    cin >> privremeni;
    s = privremeni;
    return cin;
}

istream &getline(istream &cin, String &s) {
    char privremeni[1000];
    cin.getline(privremeni, sizeof privremeni);
    s = privremeni;
    return cin;
}

```

Analizirajmo sada ovu klasu. Ona sadrži pokazivač "sadrzaj" na prostor za čuvanje znakova

stringa (koji će se alocirati dinamički), zatim i informacije o broju znakova u stringu (atribut “`br_znakova`”) kao i količini stvarno alocirane memorije (atribut “`kapacitet`”) koja može biti i veća od stvarnog broja znakova u stringu (na ovaj način se smanjuje potreba za prečestom realokacijom). Glavni konstruktor kreira prazan string, pri čemu opcionalni parametar konstruktora omogućava da se zada količina memorije koja će inicijalno biti zauzeta za čuvanje elemenata stringa (konstruktor je deklariran kao eksplisitan da spriječimo mogućnost da se stringu dodijeli cijeli broj). Ova mogućnost će nam kasnije poslužiti za lakšu izvedbu nekih drugih operacija (napomenimo da standardna klasa “`string`” iz istoimene biblioteke ne podržava ovu mogućnost). Uveden je još jedan konstruktor koji kao parametar prima obični konstantni niz znakova, koji nakon obavljenе alokacije memorije kopira taj niz znakova u alocirani prostor. Ovaj konstruktor je uveden da omogući inicijalizaciju stringova običnim stringovnim konstantama (tj. nizovima znakova omeđenim znakovima navoda), tako da su moguće inicijalizacije i dodjele tipa

```
String s = "Ja sam string!";
```

Pored toga, ovaj konstruktor omogućava da se praktično u svim situacijama kada se kao parametar ili operand očekuje objekat klase “`String`”, umjesto njega može upotrijebiti obični niz znakova ili stringovna konstanta. Od konstruktora vrijedi još spomenuti i konstruktor kopije, koji obavlja već dobro poznati zadatak, a čija je implementacija veoma slična prethodnom konstruktoru. Gdje je konstruktor kopije, tu je obično i njegova sestra, operatorska funkcija članica za preklapanje operatora dodjele (naravno, tu je i destruktor, koji se brine za oslobođanje zauzete memorije prije nego što objekat prestane postojati). Preklopljeni operator dodjele se brine da ne vrši bespotrebnu realokaciju memorije ukoliko odredišni string već ima dovoljan kapacitet da prihvati znakove stringa koji se dodjeljuje odredišnom stringu. Zahvaljujući operatoru dodjele, sasvim su legalne ovakve konstrukcije, koje omogućavaju da se ne moramo patiti sa funkcijom “`strcpy`” i njoj srodnim funkcijama:

```
String s = "Ja sam string!";
...
s = "A sada sam neki drugi string...";
```

Ipak, treba napomenuti da je dodjela iz prikazanog primjera manje efikasna nego što bi mogla biti, jer je ona, zapravo ekvivalentna sljedećoj konstrukciji:

```
s = String("A sada sam neki drugi string...");
```

Drugim riječima, prvo se koristi konstruktor sa jednim parametrom da pretvori obični niz znakova u bezimeni pomoćni primjerak klase “`String`”, a zatim se tako kreirani objekat tipa “`String`” dodjeljuje objektu “`s`” pomoću preklopljenog operatorka dodjele. Efikasnost ovakvih dodjela mogla bi se povećati kada bismo pored operatorske funkcije za operator dodjele koja kao parametar prima referencu na konstantni objekat tipa “`String`” dodali još jednu operatorsku funkciju za operator dodjele koja kao parametar prima obični konstantni niz

znakova, i koja bi direktno vršila takvu dodjelu. Čitateljima odnosno čitateljicama se ostavlja da predloženo poboljšanje implementiraju kao vježbu.

Klasa “`string`” definira i nekoliko veoma korisnih operatora. Operator “[]” omogućava da pojedinim znakovima unutar klase “`String`” pristupamo na identičan način kao da se radi o običnim nizovima znakova, samo uz provjeru ispravnosti indeksa. Treba napomenuti da, radi efikasnosti, operator “[]” definiran u klasi “`string`” iz istoimene standardne biblioteke *ne vrši provjeru ispravnosti indeksa*, mada postoji metoda nazvana “`at`” koja *vrši takvu provjeru* (uz bacanje izuzetka u slučaju neispravnosti), tako da ukoliko je “`s`” neki objekat tipa “`string`” i ukoliko želimo provjeru ispravnosti indeksa, umjesto “`s[i]`” možemo pisati “`s.at(i)`”.

Trenutnu dužinu (broj znakova) objekta tipa “`string`” možemo saznati pomoću funkcije članice “`Duzina`” (analogne funkcije članice standardne klase “`string`” zovu se “`size`” odnosno “`length`”). Veoma koristan operator je i binarni operator “`+`” koji nadovezuje drugi string na kraj prvog stringa (identično kao u standardnoj klasi “`string`”). Implementacija ovog operatora ne bi trebala da predstavlja neki problem za analizu (za ostvarivanje nadovezivanja iskorištena je funkcija “`strcat`”), a njegovo postojanje omogućava konstrukciju poput:

```
String s1 = "Klasa String ", s2 = "ilustrira principe ", s3;
s3 = s1 + s2 + "implementacije dinamičkih stringova";
cout << s3;
```

Primijetimo da je miješanje objekata tipa “`string`” sa običnim stringovnim konstantama moguće zahvaljujući konstruktorom koji vrši pretvorbu običnih znakovnih nizova u objekte tipa “`String`”. Jasno je da je ovakav rad mnogo elegantniji nego rad koji zahtijeva upotrebu funkcija “`strcpy`” i “`strcat`”. Definiran je i operator “`+=`” da podrži konstrukcije poput

```
String s = "Preopterećivanje ";
s += "operatora";
cout << s;
```

Radi efikasnosti, umjesto trivijalne implementacije tipa “`return *this = *this + s`”, koja bi zahtijevala konstruisanje zbiru stringova a zatim kopiranje konstruisanog zbiru u odredište, prikazana implementacija vrši prosto nadovezivanje pozivom funkcije “`strcat`” ukoliko ustanovi da odredišni string već ima dovoljan kapacitet da prihvati nadovezani dio. U suprotnom, postupamo klasično, pozivajući se na operatore “`=`” i “`+`” koji će obaviti neophodne realokacije memorije.

Operator za ispis “`<<`” (čija je definicija trivijalna) omogućava jednostavno ispisivanje objekata klase “`string`” na ekran, a operator “`>>`” omogućava unos objekata klase “`string`”

sa tastature. Unos se prvo vrši u neki privremeni niz znakova, koji se po obavljenom unosu dodjeljuje samom objektu (za ovu dodjelu zaslужni su konstruktor i operator dodjele). Dužinu niza znakova koji se prihvataju ograničili smo na 1000 znakova, što nije preveliko ograničenje. Ovo ograničenje bismo mogli ukloniti tako što bismo znakove sa tastature čitali *znak po znak* (primjenom metode “*get*”) dok se ne dostigne praznina (razmak ili kraj reda), pri čemu bismo svaki pročitani znak prosto *nadovezivali* na kraj tekućeg stringa, na sličan način kao što je izvedeno u definiciji operatora “*=*”. Radi jednostavnosti, nismo htjeli da ulazimo u ove komplikacije. Čitatelj odnosno čitateljica takvu modifikaciju mogu izvesti kao vježbu.

Pored operatora “*>>*”, koji omogućava samo čitanje riječ po riječ, definirana je i klasična funkcija “*getline*” (ne funkcija članica klase “*String*”) za čitanje čitave rečenice sa tastature u objekte tipa “*String*” (podsetimo se da ista mogućnost postoji i u standardnoj klasi “*string*”). Zbog činjenice da je “*getline*” ovdje izvedena kao klasična funkcija, za čitanje objekta “*s*” tipa “*String*” putem objekta ulaznog toka “*cin*” koristimo sintaksu poput “*getline(cin, s)*”, što smo već koristili sa objektima tipa “*string*”. Sada imamo mogućnost da shvatimo razlog za neophodnost ovakve sintakse. Razumije se da bi bilo ljepešće da možemo koristiti sintaksu poput “*cin.getline(s)*”, jer bi ona bila više u skladu sa načinom kako se inače objekat “*cin*” koristi u drugim kontekstima. Međutim, da bismo podržali ovaku sintaksu, “*getline*” bi morala biti metoda klase “*istream*” (a ne klase “*String*”). Drugim riječima, morali bismo u samu klasu “*istream*” dodati novu metodu “*getline*” koja bi kao parametar prihvatala objekte tipa “*String*”, što ne možemo uraditi, jer nemamo direktni pristup deklaraciji i implementaciji klase “*istream*” (čak i ukoliko imamo, izmjena deklaracija i implementacija klasa koje čine dio standardne biblioteke jezika C++ nije nimalo mudra ideja).

Veoma interesantan operator podržan u klasi “*String*” je operator “*()*” koji omogućava da iz stringa na jednostavan način izdvojimo neki njegov dio (standardna klasa “*string*” ne podržava ovaj operator, ali se veoma sličan efekat može dobiti primjenom metode “*substr*”, o čemu smo ranije govorili). U zagradama je potrebno navesti indeks početnog i krajnjeg znaka koje treba izdvojiti iz stringa, a kao rezultat se dobija upravo dio stringa u navedenom opsegu indeksa. Implementacija ovog operatorka kreira pomoćni string koji se vraća kao rezultat, a u koji se kopiraju znakovi iz navedenog opsega. Uvodjenje ovog operatorka omogućuje konstrukcije poput:

```
String s = "Ovdje fali...";  
cout << s(0, 4) + " nešto" + s(5, 12);
```

Definiranje relacionih operatorka “*=*”, “*!=*”, “*<*”, “*>*”, “*<=*” i “*>=*” za klasu “*String*” omogućava da objekte klase “*String*” možemo upoređivati (po abecednom kriteriju) korištenjem običnih relacionih operatorka, bez potrebe za korištenjem funkcije “*strcmp*” (što je, kao što već znamo, podržano i u standardnoj klasi “*string*”). Ovim postaju moguće i ovakve konstrukcije:

```
String lozinka;
```

```

cout << "Unesi lozinku: ";
cin >> lozinka;
if(lozinka != "Hrkljuš") cout << "Neispravna lozinka!\n";

```

Naravno, same funkcije koje implementiraju ove operatore interno su realizirane upravo pomoću funkcije "strcmp".

Demonstrirana klasa "String" je po funkcionalnosti veoma slična standardnoj klasi "string", ako zanemarimo neke sitnije modifikacije, kao i činjenicu da standardna klasa "string" posjeduje neke dodatne metode koje klasa "String" koju smo razvili ne posjeduje. Ipak, treba napomenuti da su tipične implementacije standardne klase "string" obično efikasnije od naše implementacije klase "String", s obzirom da tipične implementacije klase "string" najčešće koriste ranije opisanu tehniku brojanja referenciranja, sa ciljem smanjenja broja nepotrebnih kopiranja.

Sljedeća klasa koju ćemo demonstrirati je generička klasa nazvana "AdaptivniNiz", koja predstavlja pojednostavljenu verziju standardne generičke klase "vector" definirane u istoimenoj biblioteci, i koja ilustrira princip na kojem se zasniva rad generičke klase "vector". Naravno, prikazane ideje predstavljaju samo jedan od mogućih načina na koji klasa poput klase "vector" može biti implementirana, dok se njena stvarna implementacija može zasnivati i na drugačijim idejama od ovdje prikazanih ideja.

Pored standardnih elemenata koji uključuju konstruktor sa jednim parametrom pomoću kojeg se zadaje željeni kapacitet niza, destruktor, konstruktor kopije, preklopljeni operator dodjele, i preklopljeni operator indeksiranja "[]", prikazana generička klasa "AdaptivniNiz" sadrži i metode "Velicina", "Redimenzioniraj" i "DodajNaKraj" koje obavljaju analogne zadatke kao metode "size", "resize" i "push_back" u standardnoj generičkoj klasi "vector":

```

template <typename Tip>
class AdaptivniNiz {
    int br_elemenata, kapacitet;
    Tip *elementi;
public:
    explicit AdaptivniNiz(int n = 0) : br_elemenata(n), kapacitet(n),
        elementi(new Tip[n]) {}
    ~AdaptivniNiz() { delete[] elementi; }
    AdaptivniNiz(const AdaptivniNiz &niz);
    AdaptivniNiz &operator =(const AdaptivniNiz &niz);
    Tip operator [](int i) const;
    Tip &operator [](int i);
    int Velicina() const { return br_elemenata; }
    void Redimenzioniraj(int n);
    void DodajNaKraj(const Tip &element);
};

template <typename Tip>
AdaptivniNiz<Tip>::AdaptivniNiz(const AdaptivniNiz<Tip> &niz) :
    br_elemenata(niz.br_elemenata), kapacitet(niz.br_elemenata),
    elementi(new Tip[niz.br_elemenata]) {

```

```

        copy(niz.elementi, niz.elementi + br_elemenata, elementi);
    }
template <typename Tip>
AdaptivniNiz<Tip> &AdaptivniNiz<Tip>::operator =
(const AdaptivniNiz<Tip> &niz) {
    br_elemenata = niz.br_elemenata;
    if(kapacitet < br_elemenata) {
        delete[] elementi;
        kapacitet = br_elemenata;
        elementi = new Tip[br_elemenata];
    }
    copy(niz.elementi, niz.elementi + br_elemenata, elementi);
    return *this;
}

template <typename Tip>
Tip AdaptivniNiz<Tip>::operator [](int i) const {
    if(i < 0 || i >= br_elemenata) throw "Neispravan indeks!\n";
    return elementi[i];
}

template <typename Tip>
Tip &AdaptivniNiz<Tip>::operator [](int i) {
    if(i < 0 || i >= br_elemenata) throw "Neispravan indeks!\n";
    return elementi[i];
}

template <typename Tip>
void AdaptivniNiz<Tip>::Redimenzioniraj(int n) {
    if(n < kapacitet) br_elemenata = n;
    else {
        Tip *novi = new Tip[n];
        copy(elementi, elementi + br_elemenata, novi);
        delete[] elementi;
        kapacitet = br_elemenata = n;
        elementi = novi;
    }
}

template <typename Tip>
void AdaptivniNiz::DodajNaKraj(const Tip &element) {
    if(br_elemenata == kapacitet) {
        int pomocna = br_elemenata;
        Redimenzioniraj(br_elemenata * 1.5 + 1);
        br_elemenata = pomocna;
    }
    elementi[br_elemenata++] = element;
}

```

Razmotrimo sada rad ove klase. Slično ranije razmotrenoj klasi “String”, i ova klasa sadrži pokazivač “elementi” na prostor za čuvanje elemenata niza, kao i informacije o logičkom broju elemenata u nizu i stvarnom broju alociranih elemenata niza. Činjenica da se broj stvarno alociranih elemenata niza definiran atributom “kapacitet” može razlikovati od logičkog broja elemenata niza definiranog atributom “br_elemenata” omogućuje veću efikasnost nekih operacija nad nizom. Konstruktor vrši dinamičku alokaciju prostora za čuvanje elemenata niza, pri čemu se i logički i stvarni broj alociranih elemenata postavlja na vrijednost zadatu parametrom (vidjećemo da do razlike između logičkog i stvarnog broja

alociranih elemenata može doći samo nakon dodjele jednog niza drugom nizu različite veličine, ili nakon promjene veličine niza pomoću metoda “`Redimenzioniraj`” ili “`DodajNaKraj`”). Za razliku od standardne klase “`vector`”, konstruktor klase “`AdaptivniNiz`” ne inicijalizira elemente alociranog niza na podrazumijevane vrijednosti, što je u slučaju potrebe veoma lako dodati. Destruktor, konstruktor kopije i preklopljeni operator dodjele implementirani su na manje ili više standardan način, i o njima nema potrebe detaljnije govoriti. Možemo samo primijetiti da se preklopljeni operator dodjele brine da se realokacija ne vrši u slučajevima kada se kraći niz dodjeljuje dužem nizu, koji već sadrži dovoljnu količinu alocirane memorije (nakon takve dodjele, logički broj elemenata i stvarni broj alociranih elemenata u odredišnom nizu mogu postati različiti).

Implementacije operatorskih funkcija za preklapanje operatara indeksiranja “[]” su trivijalne, i o njima se nema ništa posebno reći. One bacaju izuzetak u slučaju da se kao operand upotrijebi broj koji izlazi izvan opsega u kojem se smiju kretati elementi niza. Treba naglasiti da, za razliku od klase koju ovdje razvijamo, preklopljeni operator indeksiranja “[]” kod standardne klase “`vector`” ne vrši provjeru ispravnosti indeksa (da se pri indeksiranju ne bi gubilo na efikasnosti). S druge strane, slično standardnoj klasi “`string`”, standardna klasa “`vector`” također posjeduje metodu “`at`” koja se može koristiti kao zamjena za operatorku “[]”, uz provjeru ispravnosti indeksa.

Razmotrimo sada implementaciju metode “`Redimenzioniraj`”, koja mijenja dimenzionalnost niza na vrijednost zadalu parametrom. U slučaju da je nova željena dimenzija manja od trenutno alociranog broja elemenata niza, nikakva realokacija se ne vrši, već se samo mijenja logički broj elemenata niza. U suprotnom se alocira novi niz spreman da prihvati traženi broj elemenata niza, a zatim se svi elementi postojećeg niza kopiraju u novoalocirani niz. Nakon obavljenog kopiranja, novoalocirani niz postaje aktuelni niz, a postojeći niz se briše. Na taj način je ostvareno povećanje kapaciteta niza bez gubitka postojećeg sadržaja niza. Ukoliko smatramo da je to potrebno, sasvim je lako modificirati metodu “`Redimenzioniraj`” da vrši realokaciju i pri smanjivanju broja elemenata niza, umjesto proste promjene logičkog broja elemenata niza uz zadržavanje iste količine alocirane memorije.

Na kraju, ostaje još da razmotrimo implementaciju metode “`DodajNaKraj`”, koja je zapravo najinteresantnija, i ilustrira zbog čega smo razdvajali logički broj elemenata niza i broj stvarno alociranih elemenata niza. Ova metoda se, u načelu, mogla implementirati i na sljedeći, mnogo jednostavniji način:

```
template <typename Tip>
void AdaptivniNiz::DodajNaKraj(const Tip &element) {
    Redimenzioniraj(br_elemenata + 1);
    elementi[br_elemenata - 1] = element;
}
```

Princip rada ovakve pojednostavljene verzije metode “`DodajNaKraj`” je očigledan. Ona prosto povećava dimenziju niza za jedinicu (pozivom metode “`Redimenzioniraj`”), i upisuje

element zadan kao parametar na kraj niza nakon obavljenog proširivanja (formalni parametar “element” je deklariran kao referenca na konstantni objekat tipa “Tip” umjesto samo kao objekat tipa “Tip”, radi sprečavanja nepotrebnog kopiranja stvarnog argumenta u formalni u slučaju da tip “Tip” predstavlja neki masivni tip, poput tipa “string”). S druge strane, takva pojednostavljena varijanta metode “DodajNaKraj” je vrlo neefikasna, jer se realokacija (odnosno uništavanje starog i kreiranje novog proširenog niza) obavlja praktično pri svakom dodavanju novog elementa u niz. Složenija varijanta metode “DodajNaKraj” koju smo ranije prikazali, također vrši realokaciju u slučaju da je niz popunjen (tj. u slučaju da je logički broj elemenata niza jednak broju stvarno alociranih elemenata niza). Međutim, ovdje se prilikom realokacije broj alociranih elemenata ne povećava za jedinicu, već se povećava za 50% od ukupnog broja do tada alociranih elemenata, dok se logički broj elemenata niza povećava za jedinicu, kao što i treba. Drugim riječima, prilikom dodavanja novog elementa, unaprijed rezerviramo više prostora nego što je potrebno samo za njegovo smještanje. Na ovaj način smo se pripremili za eventualno kasnije dodavanje novih elemenata. Kako se realokacija ne vrši u slučaju da je broj elemenata niza manji od broja stvarno alociranih elemenata, to do sljedeće realokacije neće doći odmah pri dodavanju sljedećeg elementa, već tek kada se dodatno rezervirani prostor popuni. Na taj način se osjetno smanjuje broj realokacija koje se vrše prilikom kontinuiranog dodavanja novih elemenata, po cijenu rezerviranja nešto većeg prostora nego što je u konkretnom trenutku zaista potreban. Praktično sve implementacije standardne klase “vector” koriste trikove slične ovdje prikazanom triku za smanjenje broja realokacija pri dodavanju novih elemenata u vektor, mada se sami detalji implementacije mogu razlikovati od slučaja do slučaja. Na primjer, strategija da se rezervira dodatna količina u iznosu od 50% trenutno zauzetog prostora ovdje je odabrana potpuno proizvoljno. Jasno je da je moguće implementirati i drugačije strategije. Međutim, osnovna ideja u svakom slučaju ostaje ista.

Prethodni primjer generičke klase “AdaptivniNiz” predstavlja klasični primjer *kontejnerske strukture podataka sa direktnim pristupom*, s obzirom da njenim elementima možemo pristupati u proizvoljnem poretku. S druge strane, u praksi se često javlja potreba za *kontejnerskim strukturama podataka sa ograničenim pristupom*, kod kojih je način na koji se vrši pristup elementima pohranjenim u kontejneru strogo ograničen. Jedna od najpoznatijih takvih kontejnerskih struktura je struktura podataka koja se obično naziva *stek* ili *stog* (engl. *stack*). To je kontejnerska struktura podataka u koju se podaci ne mogu smještati na proizvoljnu poziciju već samo na kraj (iza već do tada smještenih podataka), i iz koje se podaci mogu čitati samo obrnutim redoslijedom od redoslijeda smještanja (slično dubokoj a uskoj ladici za papire, iz koje papire možemo vaditi samo obrnutim redoslijedom u odnosu na redoslijed kojim smo ubacivali papire). Pri tome, čitanje ima destruktivan karakter, u smislu da se podatak koji pročitamo sa steka automatski uklanja sa steka. Stoga se stek još naziva i LIFO struktura podataka (od engl. Last In, First Out – zadnji ušao, prvi izašao). Tako, bi interfejs klase koja implementira stek (i koju ćemo nazvati prosto “Stek”) mogla sadržavati tri metode, koje možemo nazvati “*Stavi*”, “*Skinji*” i “*Prazan*” (u engleskoj literaturi ove tri metode se obično nazivaju “Push”, “Pull” ili “Pop” i “Empty”). Metoda “*Stavi*” posjeduje parametar koji predstavlja vrijednost koju stavljamo na stek. Metoda “*Skinji*” nema parametara. Ona vrši uklanjanje posljednjeg elementa stavljenog na stek, i usput vraća kao rezultat vrijednost upravo uklonjenog elementa. Metoda “*Prazan*” također nema parametara. Ona vraća jedinicu kao rezultat u slučaju da je stek prazan (tj. ukoliko ne sadrži niti jedan element), a u suprotnom vraća nulu. Ukoliko prepostavimo da je klasa “Stek” napisana kao

generička klasa, tako da se tip elemenata koje ćemo stavljati na stek može zadavati, tada bi sekvenca naredbi

```
Stek<int> s;
s.Stavi(5);
s.Stavi(2);
s.Stavi(7);
s.Stavi(6);
s.Stavi(3);
while(!s.Prazan()) cout << s.Skini() << endl;
```

trebala redom da ispiše brojeve 3, 6, 7, 2, 5 (svaki u posebnom redu).

U opisanom razmatranju definirali smo stek kao *apstraktnu strukturu podataka*, što zapravo znači da smo definirali kako se ova struktura podataka koristi, bez ulazeњa u to kako je ona implementirana. U nastavku ćemo razmotriti kako bismo mogli *implementirati* takvu strukturu podataka. Ukoliko prepostavimo da je maksimalni broj elemenata koji se mogu staviti na stek ograničen i unaprijed poznat, tada za simulaciju steka možemo koristiti običan niz (ovakva implementacija steka naziva se *statička implementacija*). Naime, elemente steka možemo čuvati u nizu, pri čemu će neki atribut (nazovimo ga recimo “gdje_je_vrh”) čuvati informaciju o tome gdje se nalazi tzv. *vrh steka*, odnosno informaciju o poziciji posljednjeg elementa koji je stavljen na stek. Ovakva implementacija u načelu radi dobro, ali je za smještanje elemenata steka uvijek zauzeta ista (maksimalna) količina memorije, bez obzira na stvarni broj elemenata na steku. Smještanje elementa na stek samo vrši upis u već unaprijed zauzeti prostor. Također, skidanje elementa sa steka samo pomjera informaciju o tome gdje je vrh steka, dok se prostor koji je element zauzimao u memoriji ne oslobađa. Zbog toga se takva implementacija steka naziva statičkom implementacijom. Ona bi, na primjer, mogla izgledati ovako:

```
template <typename Tip>
class Stek {
    static const int Kapacitet = 1000;
    Tip elementi[Kapacitet];
    int gdje_je_vrh;
public:
    Stek() : gdje_je_vrh(0) {}
    bool Prazan() const { return gdje_je_vrh == 0; }
    void Stavi(const Tip &element);
    Tip Skini();
};

template <typename Tip>
void Stek<Tip>::Stavi(const Tip &element) {
    if(gdje_je_vrh == Kapacitet) throw "Popunjen kapacitet steka!\n";
    elementi[gdje_je_vrh++] = element;
}
```

```

template <typename Tip>
Tip Stek<Tip>::Skinj() {
    if(gdje_je_vrh == 0) throw "Stek je prazan!\n";
    return elementi[--gdje_je_vrh];
}

```

Ova implementacija je dovoljno jednostavna da ne zahtijeva detaljnija objašnjenja. Metoda “Skinj” bi alternativno, umjesto objekta tipa “Tip” mogla kao rezultat vratiti konstantnu referencu na objektat tipa “Tip”, čime bismo mogli izbjegći kopiranje povratne vrijednosti. U ovom konkretnom primjeru, vraćanje reference je u principu dozvoljeno, o obziru da objekat na koji se referencia vraća nastavlja postojati i nakon završetka funkcije (on se nalazi u nizu). Međutim, vraćanje reference može u nekim rijetkim situacijama dovesti do problema (naime, tada je za vraćeni rezultat moguće vezati novu referencu, čiji se sadržaj može neočekivano promijeniti ukoliko kasnije dodamo novi element na stek), tako da je ipak sigurnije vratiti kopiju traženog elementa nego referencu na njega.

Ograničenje prikazane implementacije je u tome što je kapacitet steka unaprijed fiksiran, i određen statičkim konstantnim članom klase “Kapacitet” (koji je, u razmotrenom primjeru, postavljen na vrijednost 1000). Nešto fleksibilniju implementaciju bismo dobili ukoliko bismo statički niz “elementi” zamijenili dinamički alociranim nizom (pri čemu bismo njegovu veličinu mogli zadavati putem parametra konstruktora klase). Na taj način bismo kapacitet steka mogli zadati prilikom deklaracije odgovarajuće instance klase “Stek”. Međutim, i ovakva implementacija steka ostaje *statička*, s obzirom da je zauzeće memorije *fiksno* od trenutka kreiranja instance klase “Stek” pa sve do trenutka njenog uništenja. Kasnije ćemo razmotriti *dinamičke implementacije* klase “Stek”, kod kojih se veličina zauzete memorije mijenja kako se mijenja broj elemenata pohranjenih na steku.

Pored steka, u praktičnim primjenama veoma često se koristi i relativno srodnna struktura podataka nazvana *red*, ili ponekad *red čekanja* (engl. *queue*). Za razliku od steka, kod kojeg se podaci skidaju obrnutim redoslijedom od redoslijeda upisivanja, kod reda se podaci skidaju onim redoslijedom kojim su upisivani, slično obradi klijenata na šalterima kod kojeg se (barem teoretski) prvo uslužuje klijent koji je prvi u redu (odатле potiče naziv red čekanja). Stoga red nazivamo i FIFO strukturom podataka (od engl. First In, First Out – prvi ušao, prvi izašao). Ukoliko prepostavimo da hipotetička generička klasa “Red” koja implementira red poznaje metode nazvane “Ubaci”, “Izvadi” i “Prazan” (u engleskoj literaturi tipično “Enqueue”, “Dequeue” i “Empty”), tada bi sekvenca naredbi poput

```

Red<int> r;
r.Ubac(i(5);
r.Ubac(i(2);
r.Ubac(i(7);
r.Ubac(i(6);
r.Ubac(i(3);

while(!s.Prazan()) cout << s.Izvadi() << endl;

```

trebala redom da ispiše brojeve 5, 2, 7, 6, 3 (svaki u posebnom redu). Ovdje smo ujedno poštovali i izvjesne jezičke (terminološke) konvencije: podaci se *stavljaju na stek* i *skidaju sa steka*, a *ubacuju u red* i *vade iz reda*.

Kako je red veoma važna struktura podataka za praktične primjene, razmotrićemo moguće implementacije generičke klase “Red”. Pri tome ćemo se, slično kao u slučaju steka, ograničiti samo na *staticke implementacije*, kod kojih je maksimalan broj elemenata u redu fiksiran. Nažalost, literatura je prepuna loših statičkih implementacija redova, pa je na ovom mjestu neophodno pokazati nekoliko primjera *kako ne treba* i *kako treba* realizirati statičku implementaciju klase “Red”. Prvo ćemo prikazati jednu implementaciju koja u načelu radi ispravno, samo je veoma neefikasna:

```
template <typename Tip>
class Red {
    static const int Kapacitet = 1000;
    Tip elementi[Kapacitet];
    int gdje_je_vrh;
public:
    Red() : gdje_je_vrh(0) {}
    bool Prazan() const { return gdje_je_vrh == 0; }
    void Ubaci(const Tip &element);
    Tip Izvadi();
};

template <typename Tip>
void Red<Tip>::Ubaci(const Tip &element) {
    if(gdje_je_vrh == Kapacitet) throw "Popunjeno kapacitet reda!\n";
    elementi[gdje_je_vrh++] = element;
}

template <typename Tip>
Tip Red<Tip>::Izvadi() {
    if(gdje_je_vrh == 0) throw "Red je prazan!\n";
    gdje_je_vrh--;
    Tip element = elementi[0];
    copy(elementi + 1, elementi + gdje_je_vrh, elementi);
    return element;
}
```

S obzirom da se elementi reda čuvaju u običnom nizovnom atributu nazvanom “elementi”, dodavanje novog elementa na kraj je trivijalno. Problemi nastaju u metodi “Izvadi”, s obzirom da se elementi trebaju vaditi sa *početka niza*. Nažalost, vađenje elementa sa početka niza u ovakvoj implementaciji zahtijeva da se svi elementi niza koji slijede iza njega “pomjere” za jedno mjesto unazad, što je veoma neefikasno, pogotovo ukoliko je u red upisano mnogo elemenata. Može li se ovo pomjeranje izbjegići? Može, ukoliko ne zahtijevamo da elementi reda obavezno budu smješteni u niz počev od indeksa 0. Naime, umjesto da pamtimo samo indeks *posljednjeg* elementa u redu, moguće je pamtititi i indeks *prvog* elementa u redu, koji ćemo prilikom vađenja elemenata sa steka povećavati za 1, čime praktički pomjeramo indeks koji određuje poziciju prvog elementa u redu za 1. Na ovaj način dolazimo do sljedeće implementacije:

```

template <typename Tip>
class Red {
    static const int Kapacitet = 1000;
    Tip elementi[Kapacitet];
    int gdje_je_vrh, gdje_je_dno;
public:
    Red() : gdje_je_vrh(0), gdje_je_dno(0) {}
    bool Prazan() const { return gdje_je_vrh == gdje_je_dno; }
    void Ubaci(const Tip &element);
    Tip Izvadi();
};

template <typename Tip>
void Red<Tip>::Ubaci(const Tip &element) {
    if(gdje_je_vrh == Kapacitet) throw "Popunjeno kapacitet reda!\n";
    elementi[gdje_je_vrh++] = element;
}

template <typename Tip>
Tip Red<Tip>::Izvadi() {
    if(gdje_je_dno == gdje_je_vrh) throw "Red je prazan!\n";
    return elementi[gdje_je_dno++];
}

```

Ova implementacija je znatno efikasnija od prethodne, ali nažalost posjeduje jednu veliku manu. Da bismo ovo uvidjeli, pretpostavimo prvo da smo u red ubacili 950 elemenata (kapacitet reda je 1000). Nakon ovog ubacivanja, indeks “gdje_je_vrh” ima vrijednost 950, a indeks “gdje_je_dno” vrijednost 0. Neka smo nakon toga izvadili svih 950 elemenata iz reda. Po završetku ove operacije, oba indeksa “gdje_je_vrh” i “gdje_je_dno” imaju vrijednost 950. Ukoliko nakon ovoga poželimo da ponovo ubacujemo elemente u red, moći ćemo ih ubaciti samo 50, mada je red bio prazan! Problem je u tome što popunjenošć reda detektiramo tako što ispitujemo da li je indeks “gdje_je_vrh” dostigao kapacitet niza, s obzirom da se ovaj indeks ne smije dalje povećavati. Ovaj problem se može riješiti tako što ćemo zamisliti da je kraj niza “elementi” prosto “slijepljen” sa početkom (poput kružne papirnate trake), tako da iza posljednjeg elementa ponovo slijedi prvi element. Stoga ćemo indekse “gdje_je_vrh” i “gdje_je_dno” realizirati tako da se kreću “kružno” (kako pri umetanju elemenata tako i pri njihovom vađenju), odnosno tako da njihovo povećanje iza zadanog kapaciteta niza kao rezultat daje nulu. Ovo možemo uraditi i pomoći “**if**” naredbe, ali možemo koristiti i operaciju ostatka pri dijeljenju, jer se u principu radi o modularnoj aritmetici (tj. sabiranju po modulu “Kapacitet”). Još ostaje problem utvrđivanja kada je red popunjenošć pri ovakvoj realizaciji. Očigledno, red će postati popunjenošć kada se indeksi “gdje_je_vrh” i “gdje_je_dno” izjednače (tj. kada smo popunili čitav “krug” elemenata). Međutim, indeksi “gdje_je_vrh” i “gdje_je_dno” su također jednakim kada je red potpuno prazan. Da bismo mogli razlikovati ove dvije situacije, potrebno je uvesti i logičku promjenljivu “popunjenošć”, koja poprima vrijednost “**true**” onog trenutka kada se prilikom ubacivanja novog elementa ustanovi da su se indeksi “gdje_je_dno” i “gdje_je_vrh” izjednačili (tj. kada se ustanovi da se red do kraja popunišć). Na taj način dobijamo sljedeću implementaciju reda:

```

template <typename Tip>
class Red {
    static const int Kapacitet = 1000;

```

```

Tip elementi[Kapacitet];
int gdje_je_vrh, gdje_je_dno;
bool je_li_popunjen;
public:
    Red() : gdje_je_vrh(0), gdje_je_dno(0), je_li_popunjen(false) {}
    bool Prazan() const {
        return !je_li_popunjen && gdje_je_vrh == gdje_je_dno;
    }
    void Ubaci(const Tip &element);
    Tip Izvadi();
};

template <typename Tip>
void Red<Tip>::Ubaci(const Tip &element) {
    if(je_li_popunjen) throw "Popunjeno kapacitet reda!\n";
    elementi[gdje_je_vrh] = element;
    gdje_je_vrh = (gdje_je_vrh + 1) % Kapacitet;
    if(gdje_je_vrh == gdje_je_dno) je_li_popunjen = true;
}

template <typename Tip>
Tip Red<Tip>::Izvadi() {
    if(Prazan()) throw "Red je prazan!\n";
    je_li_popunjen = false;
    Tip element = elementi[gdje_je_dno];
    gdje_je_dno = (gdje_je_dno + 1) % Kapacitet;
    return element;
}

```

Ovoj implementaciji se više nema šta zamjeriti, ako se ograničimo na statičke implementacije. Eventualno je još moguće zamijeniti staticki alocirani niz “elementi” dinamički alociranim nizom čija bi se alokacija vršila unutar konstruktora klase, koji bi primao kao parametar željeni kapacitet reda. Pri tome bismo klasu morali proširiti destruktorom, konstruktorom kopije i preklopnim operatorom dodjele da bi rad sa primjercima takve klase bio bezbjedan, ili bi barem trebalo zabraniti kopiranje i međusobno dodjeljivanje primjeraka klase, deklariranjem konstruktora kopije i preklopljenog operatora dodjele u privatnoj sekciji klase (razumije se da ista primjedba vrijedi i za ranije razmotrenu klasu “*Stek*”). Kasnije ćemo razmotriti fleksibilnije, *dinamičke implementacije* klase “*Red*” kod kojih se zauzeće memorije mijenja proporcionalno broju elemenata koji se trenutno nalaze u redu.

Veoma su interesantne kontejnerske klase kod kojih se pojedinim elementima unutar kontejnera ne pristupa preko običnog cjelobrojnog indeksa, nego preko tzv. *ključa*, koji uopće ne mora biti cjelobrojnog tipa već može biti npr. stringovnog tipa. Takve klase nazivamo *asocijativni nizovi*. Oni se obično izvode sa preklopnim operatorom “[]”, ali koji kao parametar ne prihvata klasični cjelobrojni indeks, nego ključ po kojem se pristupa željenom elementu. Recimo, neka je potrebno napraviti neki objekat nalik nizu nazvan “*stanovnistvo*” koji bi čuvao informacije o broju stanovnika pojedinih gradova, ali koji bi se koristio tako da kao “indeks” (odnosno *ključ*) zadajemo *ime grada*. Na primjer:

```
stanovnistvo["Sarajevo"] = 500000;
```

```

stanovnistvo["Banja Luka"] = 350000;
stanovnistvo["Mostar"] = 150000;
cout << stanovnistvo["Sarajevo"];
razlika = stanovnistvo["Banja Luka"] - stanovnistvo["Mostar"];

```

Nije veliki problem definirati klasu koja će imati definiran operator “[]” koji prihvata string kao parametar, i koji omogućava upravo gore navedene konstrukcije. U nastavku ćemo, iz edukativnih razloga, razmotriti nekoliko različitih mogućih realizacija ovakve klase (i međusobno uporediti prikazane realizacije), koju ćemo nazvati “AsocijativniNiz”, tako da ćemo uz deklaraciju poput

```
AsocijativniNiz stanovnistvo;
```

imati asocijativni niz “stanovnistvo” sa kojim se može manipulirati kako je maločas opisano. Kroz prikazane realizacije čitatelj odnosno čitateljica će imati priliku da rezimiraju mnoge od do sada opisanih koncepata, i da uoče prednosti odnosno mane pojedinih strategija.

Jedna od jednostavnijih realizacija klase “AsocijativniNiz”, uz prepostavku da smo unaprijed ograničili maksimalan broj elemenata koji će se moći smjestiti u asocijativni niz (u prikazanom primjeru na iznos 1000), mogla bi izgledati ovako:

```

class AsocijativniNiz {
    static const int MaxBrKljuceva = 1000;
    int br_kljuceva;
    const char *kljucevi[MaxBrKljuceva];
    double vrijednosti[MaxBrKljuceva];

    public:
        AsocijativniNiz() : br_kljuceva(0) {}
        double &operator[](const char kljuc[]);
    };

    double &AsocijativniNiz::operator [](const char kljuc[]) {
        for(int i = 0; i < br_kljuceva; i++)
            if(strcmp(kljuc, kljucevi[i]) == 0) return vrijednosti[i];
        if(br_kljuceva >= MaxBrKljuceva) throw "Popunjeno niz!\n";
        kljucevi[br_kljuceva] = kljuc; vrijednosti[br_kljuceva] = 0;
        return vrijednosti[br_kljuceva++];
    }
}
```

Objasnimo sada rad ove klase. Privatni atribut “`br_kljuceva`” vodi računa o tome koliko je ključeva (npr. imena gradova) evidentirano. Konstruktor klase ovaj broj inicijalizira na nulu (i ne radi ništa više), a on se povećava za 1 svaki put kada se kao “indeks” upotrijebi ključ koji se nije ranije pojavljivao. Privatni atribut “`kljucevi`” čuva do tada definirane pojmove. Bolje rečeno, čuvaju se *pokazivači na ključeve* umjesto samih ključeva (nuspojave ovakve strategije razmotrićemo uskoro). Atribut “`vrijednosti`” čuva vrijednosti koje su pridružene ključevima (npr. broj stanovnika pridružen nekom gradu). Kao što smo već napomenuli, broj mogućih parova ključ/vrijednost koji se mogu pohraniti ograničen na vrijednost statickog konstantnog člana klase “`MaxBrPojmova`” (1000 u prikazanoj realizaciji). U nekoj fleksibilnijoj realizaciji klase “`AsocijativniNiz`”, prostor za čuvanje pokazivača na ključeve i odgovarajućih vrijednosti mogao bi se definirati dinamički (što bi zahtjevalo i definiranje destruktora, konstruktora kopije, i prekloppljenog operatora dodjele). Međutim, uskoro ćemo vidjeti da postoje bolja i jednostavnija rješenja.

Razmotrimo sada definiciju operatorske funkcije za operator “[]” koja predstavlja “mozak” čitave klase. Ona prvo traži da li je traženi ključ već ranije pohranjen u nizu koji čuva ključeve (odnosno pokazivače na njih). Ukoliko jeste, funkcija prosto vraća kao rezultat referencu na odgovarajući element niza koji čuva vrijednosti pridružene ključevima. Interesantna situacija nastaje ukoliko ključ nije pronađen. U tom slučaju, radi se o novom paru ključ/vrijednost, koji eventualno treba pohraniti. U slučaju da je kapacitet niza popunjen, baca se izuzetak, a u suprotnom, pokazivač na novi ključ smješta se u prvi slobodni element niza. Kao vrijednost inicijalno pridružena novoformiranom ključu upisuje se nula, a kao rezultat funkcije vraća se referenca na element niza u kojem se čuva ova vrijednost (uspust se evidencija o broju ključeva povećava za jedinicu), čime je stvorena priprema za eventualnu dodjelu. Na primjer, u izrazu oblika “`stanovnistvo["Bihać"] = 50000`”, izraz “`stanovnistvo["Bihać"]`” će vratiti referencu na element niza koji čuva vrijednost pridruženu novoevidentiranom ključu “`Bihać`”, u koji će biti upisana vrijednost 50000. Ovdje možemo primijetiti i jednu nuspojavu opisane realizacije. Naime, izraz poput

```
cout << Stanovnistvo["Kifino Selo"]
```

ispisaće nulu u slučaju da broj stanovnika za Kifino Selo nije prethodno bio definiran. Pri tome će se sam ključ “`Kifino Selo`” evidentirati u nizu, iako nismo izvršili nikakvu dodjelu. Bilo bi bolje kada bi operatorska funkcija za operator “[]” mogla da baci izuzetak ukoliko se navedeni ključ prethodno nije koristio, osim u slučaju kada se izraz sa do tada nepoznatim ključem nalazi sa lijeve strane znaka dodjele (tj. kada definiramo novi par ključ/vrijednost). Nažalost, ovo je dosta teško izvesti, s obzirom da operatorska funkcija nema načina da sazna da li se izraz sa uglastim zagradama nalazi sa lijeve strane znaka dodjele, ili ne. Problem je, uz upotrebu dosta prljavih trikova, ipak rješiv, o čemu ćemo govoriti nešto kasnije.

Primijetimo da prikazana implementacija klase “`AsocijativniNiz`” ne sadrži destruktur, konstruktor kopije niti prekloppljeni operator dodjele, bez obzira što sadrži atribute

pokazivačkog tipa (preciznije, niz pokazivača). Međutim, ovi pokazivači pokazuju na objekte (nizove znakova) koji nisu u vlasništvu same klase, tako da se ona ne mora brinuti o njihovom uništavanju ili kopiranju (niti to smije raditi, jer oni jednostavno nisu njen vlasništvo). S druge strane, činjenica da ova klasa sadrži pokazivače na objekte koje sama ne posjeduje čini samu klasu dosta ranjivom. Naime, čuvanje u nekoj klasi pokazivača na objekat koji nije u vlasništvu klase može dovesti do problema ukoliko objekat na koji pokazivač pokazuje iz bilo kojeg razloga prestane postojati prije nego što prestane postojati primjerak klase koji sadrži taj pokazivač. Na primjer, razmotrimo sljedeću hipotetičku sekvencu instrukcija:

```
AsocijativniNiz stanovnistvo;
{
    const char grad[] = "Sarajevo";
    stanovnistvo[grad] = 50000;
}
```

U ovom slučaju, u asocijativnom nizu “`stanovnistvo`” biće evidentiran pokazivač na znakovni niz “`grad`” (koji sadrži stringovnu konstantu “`Sarajevo`”), u čemu nema ništa loše. Međutim, ovaj znakovni niz prestaje postojati na kraju bloka u kojem je definiran, dok asocijativni niz “`stanovnistvo`” (deklariran izvan bloka) nastavlja postojati i nakon toga. Slijedi da će po izlasku iz bloka, asocijativni niz “`stanovnistvo`” sadržavati pokazivač na objekat koji je prestao postojati, odnosno višeći pokazivač!

Navedeni primjer jasno ilustrira da su pri realizaciji kontejnerskih tipova podataka mogući problemi uzrokovani pitanjem vlasništva nad objektima na koje se kontejner poziva, o čemu smo već govorili. Da bismo učinili ovu klasu sigurnijom, bolja je ideja učiniti samu klasu *vlasnikom* svih ključeva koji su u njoj evidentirani, odnosno sama klasa bi trebala posjedovati *svoju internu kopiju* svakog evidentiranog ključa. Ostaje problem gdje ove kopije čuvati. Jedna mogućnost je deklarirati atribut “`kljucevi`” kao dvodimenzionalni niz znakova u kojem će se čuvati sami ključevi. Međutim, kako sami ključevi mogu biti osjetno različitih dužina, pojавilo bi se pitanje koliko znakova rezervirati u svakom redu takvog dvodimenzionalnog niza. Mnogo bolje je prostor za čuvanje svakog od ključeva alocirati *dinamički* (u skladu sa aktuelnom dužinom ključa), a u samom nizu “`kljucevi`” čuvati pokazivače na alocirane prostore. Dakle, atribut “`kljucevi`” ostaje niz pokazivača, ali će oni ovaj put pokazivati na objekte koji su u vlasništvu same klase (tj. koje je sama klasa alocirala). Samim tim, ovakva realizacija klase će obavezno morati sadržavati destruktur, konstruktor kopije i prekloppljeni operator dodjele (stvar bi se osjetno pojednostavila ukoliko bismo zabranili kopiranje i međusobno dodjeljivanje primjeraka ove klase, što nismo htjeli učiniti iz edukativnih razloga). Predložena implementacija izgleda ovako:

```
class AsocijativniNiz {
    static const int MaxBrKljuceva = 1000;
    int br_kljuceva;
    char *kljucevi [MaxBrKljuceva];
```

```

double vrijednosti[MaxBrKljuceva];
void Kopiraj(const AsocijativniNiz &a);
void Unisti();
public:
    AsocijativniNiz() : br_kljuceva(0) {}
    ~AsocijativniNiz() { Unisti(); }
    AsocijativniNiz(const AsocijativniNiz &a) { Kopiraj(a); }
    AsocijativniNiz &operator =(const AsocijativniNiz &a);
    double &operator [](const char kljuc[]);
};

double &AsocijativniNiz::operator [](const char kljuc[]) {
    for(int i = 0; i < br_kljuceva; i++)
        if(strcmp(kljuc, kljucevi[i]) == 0) return vrijednosti[i];
    if(br_kljuceva >= MaxBrKljuceva) throw "Popunjen niz!\n";
    kljucevi[br_kljuceva] = new char[strlen(kljuc) + 1];
    strcpy(kljucevi[br_kljuceva], kljuc);
    vrijednosti[br_kljuceva] = 0;
    return vrijednosti[br_kljuceva++];
}

void AsocijativniNiz::Unisti() {
    for(int i = 0; i < br_kljuceva; i++) delete[] kljucevi[i];
}

void AsocijativniNiz::Kopiraj(const AsocijativniNiz &a) {
    br_kljuceva = a.br_kljuceva;
    for(int i = 0; i < br_kljuceva; i++) {
        vrijednosti[i] = a.vrijednosti[i]; kljucevi[i] = 0;
    }
    try {
        for(int i = 0; i < br_kljuceva; i++) {
            kljucevi[i] = new char[strlen(a.kljucevi[i]) + 1];
            strcpy(kljucevi[i], a.kljucevi[i]);
        }
    }
    catch(...) {
        Unisti(); throw;
    }
}

```

```

    }

}

AsocijativniNiz &AsocijativniNiz::operator =(const AsocijativniNiz &a)
{
    if(&a == this) return *this;
    Unisti(); Kopiraj(a);
    return *this;
}

```

Realizacija prekopljenog operatora “`[]`” u ovoj implementaciji slična je kao u prethodnoj implementaciji klase “`AsocijativniNiz`”. Suštinska razlika je u tome što se u slučaju kada treba evidentirati novi ključ, umjesto prostog kopiranja pokazivača prvo alocira prostor za pamćenje interne kopije ključa koji se evidentira, nakon čega se sam ključ kopira u alocirani prostor.

Interesantno je razmotriti destruktor, konstruktor kopije i prekopljeni operator dodjele u prikazanoj implementaciji. Destruktor prosto poziva privatnu metodu “`Unisti`” koja briše sve dinamički alocirane prostore za čuvanje internih kopija ključeva. Metoda “`Unisti`” je uvedena radi uštede u pisanju, zbog činjenice da se potpuno ista akcija vrši još na dva mesta (unutar realizacije prekopljenog operatora dodjele, kao i unutar metode “`Kopiraj`” u slučaju neuspješne alokacije memorije). Slično, konstruktor kopije samo poziva privatnu metodu “`Kopiraj`”, koja se također koristi unutar prekopljenog operatora dodjele. Metoda “`Kopiraj`” je prilično kompleksna. Ona zapravo obavlja duboko kopiranje svih objekata koji su u vlasništvu klase, a ne samo pokazivača na njih (za tu svrhu, potrebno je alocirati memoriju za čuvanje tih kopija). Bitno je naglasiti da metoda “`Kopiraj`” mora voditi računa da u slučaju neuspješne alokacije memorije, prije nego što i sama baci izuzetak počisti svo “smeće” koje je do tada eventualno ostavila za sobom (inače ga niko drugi neće obrisati, i doći će do curenja memorije). Prekopljeni operator dodjele prosto prvo uništava objekat sa lijeve strane znaka dodjele, a zatim vrši istu akciju kao i konstruktor kopije. Ovakvo rješenje može izgledati neefikasno, ali bi svako iole efikasnije rješenje bilo osjetno složenije, tako da vjerovatno nije vrijedno truda.

Prikazana implementacija klase “`AsocijativniNiz`”, mada znato sigurnija od prvobitno prikazane implementacije, osjetno je složenija, uglavnom zbog potrebe za konstruktorom kopije i prekopljenim operatorom dodjele koji trebaju obaviti duboko kopiranje dosta rasparčane strukture podataka. Može li se ista stvar uraditi jednostavnije, uz ostvarenje potpuno iste funkcionalnosti? Primijetimo da u prikazanoj implementaciji mi zapravo *simuliramo* dinamičke stringove koji su već implementirani u standardnoj klasi “`string`” iz istoimene biblioteke (a također i u klasi “`String`” koju smo ranije razvili). Principi efikasnog programiranja zasnivaju se na *korištenju onog što je već jednom razvijeno*, a ne na ponovnoj implementaciji već implementiranih stvari (takva ponovna implementacija može imati svrhe samo za edukativne primjene, radi demonstracije koncepata jezika C++). Stoga ćemo mnogo jednostavniju implementaciju klase “`AsocijativniNiz`” dobiti ukoliko se odlučimo da

ključeve držimo u nizu čiji su elementi dinamički stringovi tipa “string”, umjesto da dinamičke stringove realiziramo “pješke”, kao u prethodnoj implementaciji. Na ovaj način dobijamo sljedeću implementaciju, koja nije ništa složenija od prvobitne implementacije, a funkcionalno je ekvivalentna prethodnoj implementaciji (odnosno, sigurnija je od prvobitne implementacije):

```
class AsocijativniNiz {  
    static const int MaxBrKljuceva = 1000;  
    int br_kljuceva;  
    string kljucevi[MaxBrKljuceva];  
    double vrijednosti[MaxBrKljuceva];  
  
public:  
    AsocijativniNiz() : br_kljuceva(0) {}  
    double &operator [](const string &kljuc);  
};  
  
double &AsocijativniNiz::operator [](const string &kljuc) {  
    for(int i = 0; i < br_kljuceva; i++)  
        if(kljucevi[i] == kljuc) return vrijednosti[i];  
    if(br_kljuceva >= MaxBrKljuceva) throw "Popunjen niz!\n";  
    kljucevi[br_kljuceva] = kljuc; vrijednosti[br_kljuceva] = 0;  
    return vrijednosti[br_kljuceva++];  
}
```

Rad ove implementacije ne treba posebno objašnjavati, jer je gotovo identičan radu prvobitne implementacije, osim što klasa u nizovnom atributu “kljucevi” ne čuva pokazivače na ključeve, već interne kopije samih ključeva. Za razliku od prethodne implementacije, ovdje nije potrebna dinamička alokacija memorije, destruktur, konstruktor kopije i preklopmani operator dodjele, jer se o ispravnoj alokaciji memorije, uništavanju i kopiranju objekata tipa “string” pohranjenih u ovoj implementaciji klase “AsocijativniNiz” brinu konstruktori, destruktur, konstruktor kopije i preklopmani operator dodjele same klase “string”. U primjerima u ranijim poglavljima, objekte tipa “string” nismo intenzivno koristili, uglavnom radi demonstracije pojedinih koncepcata nižeg nivoa. Međutim, upravo izloženi primjer najbolje ilustrira koliko nam upotreba objekata tipa “string” može olakšati posao.

Sve prikazane implementacije klase “AsocijativniNiz” posjeduju nedostatak što je broj parova ključ/vrijednost koji se mogu pohraniti *ograničen*. Ništa se suštinski ne bi promijenilo ukoliko bismo umjesto statičkih nizova “kljucevi” i “vrijednosti” koristili dinamički alocirane nizove. Jedina razlika bila bi u tome što bismo maksimalni kapacitet mogli *zadavati* prilikom deklaracije primjeraka klase “AsocijativniNiz” (recimo, putem parametara konstruktora), ali bi kapacitet i dalje bio *fiksan* tokom života primjerka klase (pored toga, ne zaboravimo da bi prelazak na dinamičku alokaciju zahtijevao i destruktur, konstruktor kopije i

preklopljeni operator dodjele). Može li se napraviti fleksibilnija varijanta u kojoj broj parova ključ/vrijednost koji se mogu pohraniti nije unaprijed ograničen? Jedna mogućnost je *dinamička realizacija*, o čemu ćemo govoriti u kasnijim poglavljima. Druga mogućnost je da u realizaciji klase koristimo dinamički alocirane nizove, ali pri čemu ćemo s vremena na vrijeme vršiti *realokaciju* kada se njihov kapacitet popuni, kao što smo radili prilikom implementacije klase “*AdaptivniNiz*”. Međutim, zašto ponovo otkrivati toplu vodu ukoliko takva funkcionalnost već postoji implementirana u generičkoj klasi “*vector*”? Rješenje se sad prosto samo nameće: ključeve i odgovarajuće vrijednosti nećemo čuvati u običnim nizovima, već u objektima tipa “*vector*”, koji će inicijalno biti prazni, a u koje ćemo nove parove ključ/vrijednost dodavati pomoću metode “*push_back*”. To nas vodi do sljedeće implementacije:

```
class AsocijativniNiz {
    vector<string> kljucevi;
    vector<double> vrijednosti;
public:
    double &operator [](const string &kljuc);
};

double &AsocijativniNiz::operator [](const string &kljuc) {
    for(int i = 0; i < kljucevi.size(); i++)
        if(kljucevi[i] == kljuc) return vrijednosti[i];
    kljucevi.push_back(kljuc); vrijednosti.push_back(0);
    return vrijednosti[vrijednosti.size() - 1];
}
```

Primijetimo da je prikazana implementacija čak kraća od prvobitne implementacije, a znatno je fleksibilnija od nje. Ovaj primjer jasno ilustrira u kojoj mjeri jednostavnost i fleksibilnost implementacije može ovisiti od pogodnog izbora struktura podataka na kojoj će se zasnivati struktura podataka koju želimo razviti. Mnogo ćemo posla uštediti ukoliko razvoj nekog korisničkog tipa podataka zasnujemo na drugim, već razvijenim tipovima podataka koji posjeduju funkcionalnosti koje su nam potrebne za razvoj. Na taj način, nećemo morati čitav razvoj počinjati od nule.

U posljednjoj implementaciji klase “*AsocijativniNiz*” nestala je potreba za atributom “*br_kljuceva*”, s obzirom da se ista informacija može dobiti primjenom metode “*size*” nad atributima “*kljucevi*” ili “*vrijednosti*”. Samim tim, otpala je potreba i za konstruktorom, čija je jedina funkcija bila inicijalizacija atributa “*br_kljuceva*” na nulu (propisnu inicijalizaciju atributa “*kljucevi*” i “*vrijednosti*” će obaviti sam konstruktor generičke klase “*vector*”). Destruktor, konstruktor kopije, i preklopljeni operator dodjele također nisu potrebni, s obzirom da će njihov zadatak ispravno obaviti destruktur, konstruktor kopije i

preklopjeni operator dodjele klase “vector”.

Razvijena klasa “AsocijativniNiz” ograničena je na ključeve stringovnog tipa, i vrijednosti tipa “**double**”. Sastavim je jednostavno prepraviti realizaciju ove klase tako da postane generička klasa, u kojoj se tip ključa i tip vrijednosti mogu zadavati. Na primjer, prepravićemo prethodnu realizaciju klase (s obzirom da je najjednostavnija i najfleksibilnija od svih prikazanih realizacija) tako da postane generička klasa:

```
template <typename TipKljuca, typename TipVrijednosti>
class AsocijativniNiz {
    vector<TipKljuca> kljucevi;
    vector<TipVrijednosti> vrijednosti;
public:
    TipVrijednosti &operator[] (const TipKljuca &kljuc);
};

template <typename TipKljuca, typename TipVrijednosti>
TipVrijednosti &AsocijativniNiz<TipKljuca, TipVrijednosti>::operator[]
{
    (const TipKljuca &kljuc) {
        for(int i = 0; i < kljucevi.size(); i++)
            if(kljucevi[i] == kljuc) return vrijednosti[i];
        kljucevi.push_back(kljuc);
        vrijednosti.push_back(TipVrijednosti());
        return vrijednosti[vrijednosti.size() - 1];
    }
}
```

Uz ovaku implementaciju, asocijativni niz “stanovnistvo” koji smo ranije uzimali kao primjer možemo deklarirati ovako:

```
AsocijativniNiz<string, double> stanovnistvo;
```

Također, primijetimo da ukoliko asocijativni niz “stanovnistvo” deklariramo ovako:

```
AsocijativniNiz<const char *, double> stanovnistvo;
```

on će biti funkcionalno ekvivalentan asocijativnom nizu dobijenom pomoću prve razmotrenе implementacije klase “AsocijativniNiz”, odnosno implementacije koja je samo čuvala pokazivače na ključeve, a ne njihove interne kopije.

Interesantno je da je moguće kreirati asocijativne nizove čiji će ključevi biti također cjelobrojnog tipa. Na primjer, ništa nas ne sprečava da zadamo sljedeću deklaraciju:

```
AsocijativniNiz<int, int> a;
```

Tako deklarirani asocijativni niz “a” može se koristiti na način koji podsjeća na način na koji se koriste klasični nizovi. Na primjer, uz prethodnu deklaraciju, sasvim su moguće sljedeće konstrukcije:

```
a[5] = 12;
a[32] = 1;
a[1149] = 131;
a[23398] = 91;
cout << a[5] << " " << a[32] << " " << a[1149] << " " << a[23398];
```

Međutim, bitno je primijetiti da bi realizacija iste konstrukcije sa običnim nizom (ili vektorom) zahtijevala niz od barem 23399 elemenata, s obzirom da je najveći “indeks” koji se koristi 23398. S druge strane, ukoliko je “a” asocijativni niz, u njemu su zapravo zapamćena samo četiri para ključ/vrijednost, odnosno parovi 5/12, 32/1, 1149/131 i 23398/91. Drugim riječima, upotreba asocijativnih nizova može dovesti do izuzetne uštede u memoriji pri radu sa tzv. *rijetkim nizovima*, odnosno nizovima kod kojih je samo mali broj elemenata različit od nule.

Vrijedi napomenuti da u standardnoj biblioteci “*map*” jezika C++ postoji već definirana gotova istoimena generička klasa, čija je funkcionalnost veoma slična generičkoj klasi “*AsocijativniNiz*” koju smo upravo razvili, samo što je implementirana na znatno složeniji ali efikasniji način koji omogućava brže pronalaženje elemenata sa zadanim ključem i umetanje novih ključeva (ona je efektivno zasnovana na *binarnoj pretrazi*, dok se naša implementacija zasnovana na *sekvencijalnoj pretrazi*). Pored toga, standardna generička klasa “*map*” poznaje i izvjesne metode koje omogućavaju dodatne manipulacije sa primjercima ove klase, koje naša klasa “*AsocijativniNiz*” ne poznaje. U svakom slučaju, u praktičnom programiranju, klase poput “*AsocijativniNiz*” ne treba implementirati, već treba koristiti gotovu klasu “*map*” (uz uključivanje zaglavlja istoimene biblioteke). Tako bi se ranije razmotreni primjeri asocijativnih nizova “stanovnistvo” i “a” mogli deklarirati ovako:

```
map<string, double> stanovnistvo;
map<int, int> a;
```

Kao ograničenje generičke klase “`map`” možemo navesti činjenicu da ključevi moraju biti tipovi koji se mogu i smiju porediti pomoću operatora “`<`”. Zainteresirani čitatelji i čitateljice se upućuju na širu literaturu koja obrađuje standardne biblioteke jezika C++.

Razmotrimo sada kako bismo mogli riješiti ranije opisani problem nepostojećih ključeva. Zamislimo da koristimo asocijativni niz “`stanovnistvo`”, i da broj stanovnika za Zenicu nismo još definirali. Tada bismo željeli da izraz poput “`cout << stanovnistvo["Zenica"]`” baci izuzetak, ali da izraz poput “`stanovnistvo["Zenica"] = 100000`” radi ispravno. Kako operatorska funkcija za operator “[]” ne može znati sa koje strane znaka jednakosti je upotrijebljen, jedina mogućnost je da ova operatorska funkcija ne vrati direktno vrijednost pridruženu ključu kao rezultat (koja je realan broj), već neki *pomoćni objekat*. Taj pomoćni objekat posjedovaće preklopljeni operator dodjele (koji će obaviti upisivanje novog ključa), kao i operator konverzije u tip “`double`” (koji će vratiti vrijednost pridruženu ključu, ali samo ako traženi ključ postoji). Ideja se zasniva u tome da ukoliko je takav objekat iskorišten sa lijeve strane operatora dodjele, biće pozvana njegova operatorska funkcija za operator dodjele. Kako za taj objekat nije definiran niti jedan drugi operator, u svim drugim slučajevima biće pozvana njegova operatorska funkcija za konverziju u tip “`double`”, koja će dati traženu vrijednost. Pošto ovaj pomoćni objekat ne treba da bude vidljiv izvan same klase “`AsocijativniNiz`”, klasu koja ga opisuje (nazovimo je “`Pristupnik`”) deklariraćemo u privatnoj sekciji same klase “`AsocijativniNiz`”. Naravno, takav pomoćni objekat mora znati o kojem se ključu radi, i na koji se primjerak asocijativnog niza odnosi. Zbog toga će njegovi atributi biti ključ, i referenca na pripadni primjerak asocijativnog niza kojem pristupa. Sama implementacija mogla bi izgledati ovako (da ne bismo previše komplikirali, prikazali smo negeneričku verziju, kod koje su ključevi stringovi, a vrijednosti realni brojevi):

```
class AsocijativniNiz {
    class Pristupnik {
        AsocijativniNiz &ref;
        string kljuc;
        Pristupnik(AsocijativniNiz &ref, const string &kljuc)
            : ref(ref), kljuc(kljuc) {}

    public:
        operator double();
        Pristupnik &operator =(double vrijednost);
        friend class AsocijativniNiz;
    };

    vector<string> kljucevi;
    vector<double> vrijednosti;

    public:
        Pristupnik operator [](const string &kljuc) {
            return Pristupnik(*this, kljuc);
```

```

    }

};

AsocijativniNiz::Pristupnik::operator double() {
    for(int i = 0; i < ref.kljucevi.size(); i++)
        if(ref.kljucevi[i] == kljuc) return ref.vrijednosti[i];
    throw "Ključ nije definiran!\n";
}

AsocijativniNiz::Pristupnik &AsocijativniNiz::Pristupnik::operator =
(double vrijednost) {
    for(int i = 0; i < ref.kljucevi.size(); i++)
        if(ref.kljucevi[i] == kljuc) {
            ref.vrijednosti[i] = vrijednost;
            return *this;
        }
    ref.kljucevi.push_back(kljuc);
    ref.vrijednosti.push_back(vrijednost);
    return *this;
}

```

Treba ipak napomenuti da je prikazana tehnika prilično komplikirana i “prJAVA”, tako da se analiza prikazane implementacije ostavlja samo ambicioznijim čitateljima i čitateljkama. S druge strane, iz pažljive analize ove implementacije može se zaista mnogo naučiti.

U sljedećem primjeru ćemo razmotriti veoma jednostavnu generičku klasu nazvanu “Referenca”, koja simulira ponašanje referenci, i koja ilustrira šta reference zapravo jesu, i kako su tipično interna implementirane:

```

template <typename Tip>
class Referenca {
    Tip *pok;
public:
    Referenca(Tip &a) { pok = &a; };
    operator Tip() { return *pok; }
    Tip *operator &() { return pok; }
    Referenca &operator =(const Tip &v) { *pok = v; }

```

```
};
```

Smisao ove klase je u tome da je deklaracija poput

```
Referenca<int> ref = a;
```

funkcionalno potpuno ekvivalentna deklaraciji

```
int &ref = a;
```

Kako ova klasa radi? Njen jedini atribut je pokazivač “pok” koji čuva adresu vezanog objekta, što također sadrže i prave reference. Konstruktor klase “Referenca” uzima adresu svog parametra, i smješta je u pokazivač “pok”. Kako se parametar prenosi po referenci, stvarni parametar mora biti l-vrijednost (što oponaša svojstvo referenci da se mogu vezati samo za l-vrijednosti). Pored toga, prenos po referenci garantira da su adresa stvarnog i formalnog parametra identične (jer se radi faktički o istim objektima). Dalje, klasa “Referenca” sadrži preklopljeni unarni operator “&” koji osigurava da njegova primjena na objekat tipa “Referenca” ne vrati kao rezultat *njegovu adresu* nego *adresu vezanog objekta* (pohranjenu u pokazivaču “pok”). Klasa “Referenca” sadrži i preklopljeni operator dodjele, koji obezbeđuje da pokušaj dodjele objektu ove klase zapravo izvrši dodjelu vezanom objektu, kao što se i dešava prilikom upotrebe pravih referenci. Konačno, ova klasa sadrži i operator konverzije u tip vezanog objekta, koji obezbeđuje da će se upotreba objekta klase “Referenca” u bilo kojem drugom kontekstu automatski konvertirati u vezani objekat (koji je određen kao sadržaj na koji pokazuje pokazivač “pok”). Drugim riječima, razvili smo klasu koja u potpunosti oponaša ponašanje referenci, i na taj način ilustrira principe njihovog rada.

Posljednja klasa koju ćemo demonstrirati u ovom poglavlju je poboljšana verzija klase “Matrica” koju smo već razvijali u prethodnim poglavlјima, ali koja je ovaj put proširena podrškom za operatore. Da ne bismo isuviše komplikirali, klasu nećemo izvesti kao generičku, nego ćemo se ograničiti na elemente tipa “**double**”:

```
class Matrica {
    int br_redova, br_kolona;
    double **elementi;
    char ime_matrice[10];
    void AlocirajMemoriju(int br_redova, int br_kolona);
    void DealocirajMemoriju(int br_redova);
public:
    Matrica(int br_redova, int br_kolona, const char ime[] = "");
    Matrica(const Matrica &m);
    ~Matrica() { DealocirajMemoriju(br_redova); }
    Matrica &operator =(const Matrica &m);
    friend istream &operator >>(istream &cin, Matrica &m);
    friend ostream &operator <<(ostream &cout, const Matrica &m);
    friend const Matrica operator +(const Matrica &m1,
        const Matrica &m2);
```

```

        double *operator [](int i) { return Elementi[i]; }
        double &operator ()(int i, int j);
    };

void Matrica::AlocirajMemoriju(int br_redova, int br_kolona) {
    elementi = new double*[br_redova];
    for(int i = 0; i < br_redova; i++) elementi[i] = 0;
    try {
        for(int i = 0; i < br_redova; i++)
            elementi[i] = new double[br_kolona];
    }
    catch(...) {
        DealocirajMemoriju(br_redova);
        throw;
    }
}

void Matrica::DealocirajMemoriju(int broj_redova) {
    for(int i = 0; i < broj_redova; i++) delete[] elementi[i];
    delete[] elementi;
}

Matrica::Matrica(int broj_redova, int broj_kolona, const char ime[]) :
    br_redova(br_redova), br_kolona(br_kolona) {
    if(strlen(ime) > 9) throw "Predugačko ime!\n";
    strcpy(ime_matrice, ime);
    AlocirajMemoriju(br_redova, br_kolona);
}

Matrica::Matrica(const Matrica &m) : br_redova(m.br_redova),
    br_kolona(m.br_kolona) {
    strcpy(ime_matrice, m.ime_matrice);
    AlocirajMemoriju(br_redova, br_kolona);
    for(int i = 0; i < br_redova; i++)
        for(int j = 0; j < br_kolona; j++)
            elementi[i][j] = m.elementi[i][j];
}

Matrica &Matrica::operator =(const Matrica &m) {
    if(br_redova < m.br_redova || br_kolona < m.br_kolona) {
        DealocirajMemoriju(br_redova);
        AlocirajMemoriju(m.br_redova, m.br_kolona);
    }
    strcpy(ime_matrice, m.ime_matrice);
    br_redova = m.br_redova; br_kolona = m.br_kolona;
    for(int i = 0; i < br_redova; i++)
        for(int j = 0; j < br_kolona; j++)
            elementi[i][j] = m.elementi[i][j];
    return *this;
}

istream &operator >>(istream &cin, Matrica &m) {
    for(int i = 0; i < m.br_redova; i++)
        for(int j = 0; j < m.br_kolona; j++) {
            cout << m.ime_matrice << "(" << i + 1 << "," << j + 1 << ")" = ";
            cin >> m.elementi[i][j];
        }
    return cin;
}

```

```

ostream &operator <<(ostream &cout, const Matrica &m) {
    int sirina_ispisa = cout.width();
    for(int i = 0; i < m.br_redova; i++) {
        for(int j = 0; j < m.br_kolona; j++)
            cout << setw(sirina_ispisa) << m.elementi[i][j];
        cout << endl;
    }
    return cout;
}

const Matrica operator +(const Matrica &m1, const Matrica &m2) {
    if(m1.br_redova != m2.br_redova || m2.br_redova != m2.br_redova)
        throw "Matrice nemaju jednake dimenzije!\n";
    Matrica m3(m1.br_redova, m1.br_kolona);
    for(int i = 0; i < m1.br_redova; i++)
        for(int j = 0; j < m1.br_kolona; j++)
            m3.elementi[i][j] = m1.elementi[i][j] + m2.elementi[i][j];
    return m3;
}

double &Matrica::operator ()(int i, int j) {
    if(i < 1 || i > br_redova || j < 1 || j > br_kolona)
        throw "Indeksi izvan opsega!\n";
    return elementi[i - 1][j - 1];
}

```

Prije nego što detaljnije objasnimo pojedine detalje ove klase, ilustrirajmo koliko je sada elegantan rad sa objektima klase “`Matrica`” na sljedećem testnom primjeru:

```

int m, n;
cout << "Unesi broj redova i kolona za matrice:\n";
cin >> m >> n;
try {
    Matrica a(m, n, "A"), b(m, n, "B");
    cout << "Unesi matricu A:\n";
    cin >> a;
    cout << "Unesi matricu B:\n";
    cin >> b;
    cout << "Zbir ove dvije matrice je:\n" << setw(7) << a + b << endl;
    cout << "Element matrice A na poziciji (1, 1) je ";
    cout << a(1, 1) << " odnosno " << a[0][0] << endl;
}
catch(...) {
    cout << "Nema dovoljno memorije!\n";
}

```

U deklaraciji i implementaciji ove klase ima mnogo detalja koji su praktično identični u našim ranijim definicijama klase “`Matrica`”, pa ćemo na ovom mjestu ukazati samo na izmjene koje smo ovde napravili. Konstruktor, destruktur, konstruktor kopije i preklopljeni operator dodjele ostali su isti kao u verziji klase “`Matrica`” koju smo ranije razvili. Operatorske funkcije za operatore “`+`” i “`>>`” zamijenile su raniju funkciju “`ZbirMatrica`” i metodu (funkciju članicu) “`Unesi`”, pri čemu je njihova implementacija ostala praktično ista (osim što operatorska funkcija za operator “`>>`” nije funkcija članica). Također operatorska funkcija za operator “`<<`” zamijenila je raniju metodu “`Ispisi`”. Međutim, metoda ispis imala je i

parametar kojim smo zadavali željenu širinu ispisa. Ovdje smo omogućili da se širina ispisa zadaje na prirodan način pomoću manipulatora “`setw`”. Na primjer, ukoliko je “`a`” matrica, možmo zadati konstrukciju poput

```
cout << setw(10) << a;
```

da omogućimo ispis matrice “`a`” pri čemu će za svako polje biti zauzeto po 10 mesta. Interesantno je razmotriti na koji način operatorska funkcija za operator “`<<`” saznaje koja je širina bila postavljena manipulatorom “`setw`”. Ovo je ostvareno pozivom metode “`width`” bez parametara nad objektom “`cout`” koja vraća kao rezultat trenutno postavljenu širinu ispisa.

Konačno, klasa “`Matrica`” definira i operatore “`()`” i “`[]`”. Operatorska funkcija za operator “`()`” omogućava pristup elementima matrice navodenjem indeksa reda i kolone razdvojenih zarezima unutar zagrada (npr. kao u izrazu “`a(2, 3)`”) pri čemu indeksi redova i kolona idu od jedinice, a podržana je i provjera ispravnosti indeksa. Pored toga, podržana je i upotreba operatora “[]” i sintakse poput “`a[2][3]`”, ali ovaj put indeksi se kreću *od nule*, i provjera ispravnosti indeksa nije podržana. Naime, operatorsku funkciju za ovaj operator implementirali smo tako što operator “[]” primijenjen na matricu (npr. kao u izrazu “`A[2]`”) vraća *pokazivač* na odgovarajući red matrice, dok se druga primjena operatora “[]” primjenjuje na vraćeni pokazivač na uobičajeni način. Na taj način, na tretman prvog indeksa bismo i mogli uticati (npr. da mu provjeravamo opseg i da ga pomjerimo da kreće od jedinice), ali na tretman drugog indeksa ne možemo. Radi konzistencije, odlučili smo se da ne vršimo nikakve manipulacije ni nad prvim indeksom.

Ukoliko je zbog nekog razloga zaista neophodno da elementima matrice pristupamo koristeći standardnu sintaksu za rad sa dvodimenzionalnim nizovima “`a[i][j]`”, a želimo da imamo mogućnost uticaja na tretman *oba indeksa*, jedino rješenje je da operator “[]” primijenjen na matricu vrati kao rezultat ponovo objekat neke klase koja ima preopterećen operator “[]”, i da u odgovarajućoj operatorskoj funkciji za tu klasu izvršimo obradu i tretman drugog indeksa. Ovu tehniku demonstrirali smo u narednom primjeru, u kojem smo u privatnoj sekciji klase “`Matrica`” definirali lokalnu pomoćnu klasu nazvanu “`Pristupnik`”, u koju smo “upakovali” pokazivač na red matrice. Tako, operator “[]” primijenjen na matricu vraća kao rezultat objekat ove klase u koju je upakovani pokazivač na odgovarajući red matrice, a zatim druga primjena operatora “[]” djeluje na vraćeni objekat ove klase. Pri tome odgovarajuća operatorska funkcija dohvaća traženi element matrice preko ovog pokazivača, i vraća referencu na njega, kao što je i uobičajeno. U ovom primjeru smo iskoristili mogućnost da utičemo na oba indeksa, tako da smo realizirali da se kontrolira opseg oba indeksa, i da se oba indeksa kreću od jedinice. Prikazaćemo neophodne izmjene u klasi “`Matrica`” koje omogućavaju ovakav tretman (uključujući i kompletну deklaraciju ugniježdene pomoćne klase “`Pristupnik`”):

```
class Matrica {  
    class Pristupnik {  
        double *pok_na_red;  
    }  
};
```

```

int br_kolona;

Pristupnik(double *pok_na_red, int br_kolona)
    : pok_na_red(pok_na_red), br_kolona(br_kolona) {}

public:
    double &operator [](int j);
    friend class Matrica;
};

...

public:
    ...
Pristupnik operator[](int i);
...
};

Pristupnik Matrica::operator [](int i) {
    if(i < 1 || i > br_redova) throw "Indeksi izvan opsega!\n";
    return Pristupnik(elementi[i - 1], br_kolona);
}

double &Matrica::Pristupnik::operator [](int j) {
    if(j < 1 || j > br_kolona) throw "Indeksi izvan opsega!\n";
    return pok_na_red[j - 1];
}

```

Klasu "Pristupnik" smo deklarirali unutar privatne sekcije klase "Matrica", a klasu "Matrica" proglašili smo njenim prijateljem (što njenim metodama daje pravo pristupa privatnim elementima klase "Matrica", što se samo po sebi ne podrazumijeva). Na taj način je onemogućeno da se klasa "Pristupnik" koristi izašta drugo osim za predviđenu svrhu. Preciznije, ostatak programa nije uopće svjestan njenog postojanja!

35. Nasljeđivanje i polimorfizam

Kada smo govorili o filozofiji objektno orijentiranog programiranja, rekli smo da su njena četiri osnovna načela *sakrivanje informacija*, *enkapsulacija*, *nasljeđivanje* i *polimorfizam*. Sa sakrivanjem informacija i enkapsulacijom smo se već detaljno upoznali. Sada je vrijeme da se upoznamo i sa preostala dva načela, nasljeđivanjem i polimorfizmom. Tek kada programer ovlada i ovim načelima i počne ih upotrebljavati u svojim programima, može reći da je usvojio koncepte objektno orijentiranog programiranja.

Nasljeđivanje (engl. *inheritance*) je metodologija koja omogućava definiranje klase koje *nasljeđuju* većinu svojstava nekih već postojećih klasa. Na primjer, ukoliko definiramo da je klasa “B” nasljeđena iz klase “A”, tada klasa “B” automatski nasljeđuje sve atribute i metode koje je posjedovala i klasa “A”, pri čemu je moguće u klasi “B” dodati nove atribute i metode koje klasa “A” nije posjedovala, kao i promijeniti definiciju neke od nasljeđenih metoda. Klasa “A” tada se naziva *bazna, osnovna ili roditeljska klasa* (engl. *base class, parent class*) za klasu “B”, a za klasu “B” kažemo da je *izvedena* ili *nasljeđena* iz klase “A” (engl. *derived class*).

Veoma je važno ispravno shvatiti u kojim slučajevima treba koristiti nasljeđivanje. Naime, cijeli mehanizam nasljeđivanja zamišljen je tako da se neka klasa (recimo klasa “B”) treba nasljediti iz neke druge klase (recimo klasa “A”) jedino u slučaju kada se svaki primjerak izvedene klase može shvatiti kao specijalan slučaj primjeraka bazne klase, pri čemu eventualni dodatni atributi i metode izvedene klase opisuju specifičnosti primjeraka izvedene klase u odnosu na primjerke bazne klase. Na primjer, neka imamo klase “Student” i “DiplomiraniStudent” (npr. student postdiplomskog ili doktorskog studija). Činjenica je da će klasa “DiplomiraniStudent” sigurno sadržavati neke atribute i metode koje klasa “Student” ne sadrži (npr. godinu diplomiranja, temu diplomskog rada, ocjenu sa odbrane diplomskog rada, itd.). Međutim, neosporna je činjenica da svaki diplomirani student *jestе* ujedno i student, tako da sve što ima smisla da se radi sa primjercima klase “Student” ima smisla da se radi i sa primjercima klase “DiplomiraniStudent”. Stoga je klasu “DiplomiraniStudent” razumno definirati kao nasljeđenu klasu iz klase “Student”. Generalno, prije nego što se odlučimo da neku klasu “B” definiramo kao izvedenu klasu iz klase “A”, trebamo sebi postaviti pitanje da li se svaki primjerak klase “B” može ujedno shvatiti kao primjerak klase “A”, kao i da li se primjerci klase “B” uvijek mogu koristiti u istom kontekstu u kojem i primjerci klase “A”. Ukoliko su odgovori na oba pitanja potvrđni, svakako treba koristiti nasljeđivanje. Ukoliko je odgovor na prvo pitanje odrečan, nasljeđivanje ne treba koristiti, s obzirom da ćemo tada od nasljeđivanja imati više štete nego koristi. U slučaju da je odgovor na prvo pitanje potvrđan, a na drugo odrečan nasljeđivanje može, ali i ne mora biti dobro rješenje, zavisno od situacije. O tome ćemo detaljnije govoriti nešto kasnije.

Nasljeđivanje nipošto ne treba koristiti samo zbog toga što neka klasa posjeduje sve atribute i metode koje posjeduje i neka druga klasa. Na primjer, pretpostavimo da želimo da napravimo klase nazvane “Vektor2” i “Vektor3” koje redom predstavljaju dvodimenzionalni odnosno trodimenzionalni vektor. Očigledno će klasa “Vektor3” sadržavati sve atribute i metode kao i klasa “Vektor2” (i jednu koordinatu više), tako da na prvi pogled djeluje prirodno koristiti

nasljeđivanje i definirati klasu “Vektor3” kao izvedenu klasu iz klase “Vektor2”. Međutim, činjenica je da svaki trodimenzionalni vektor *nije* ujedno i dvodimenzionalni vektor, tako da u ovom primjeru nije nimalo uputno koristiti nasljeđivanje. Naravno, sam jezik nam ne brani da to uradimo (kompajler ne ulazi u filozofske diskusije tipa da li je nešto što ste vi uradili *logično ili nije*, nego samo da li je *sintaksno dopušteno ili nije*). Međutim, ukoliko bismo uradili tako nešto, mogli bi se uvaliti u nevolje. Naime, vidjećemo da jezik C++ dopušta da se primjeri naslijedene klase koriste u svim kontekstima u kojem se mogu koristiti i primjeri bazne klase. To znači da bi se primjeri klase “Vektor3” mogli koristiti svugdje gdje i primjeri klase “Vektor2”. S druge strane, to ne mora biti opravданo jer trodimenzionalni vektori *nisu* dvodimenzionalni vektori, i sasvim je moguće zamisliti operacije koje su definirane za dvodimenzionalne vektore a nisu za trodimenzionalne vektore. Stoga je najbolje definirati klase “Vektor2” i “Vektor3” posve neovisno jednu od druge, bez korištenja ikakvog nasljeđivanja. Možda djeluje pomalo absurdno, ali ukoliko baš želimo da koristimo javno nasljeđivanje, tada je bolje *klasu “Vektor2” naslijediti iz klase “Vektor3”!* Zaista, dvodimenzionalni vektori jesu specijalni slučaj trodimenzionalnih vektora, i mogu se koristiti u svim kontekstima gdje i trodimenzionalni vektori. Na prvi pogled djeluje dosta čudno da je ovdje nasljeđivanje opravданo, s obzirom da dvodimenzionalni vektori imaju jednu koordinatu manje. Međutim, istu stvar možemo posmatrati i tako da smatramo da dvodimenzionalni vektori također sadrže tri koordinate, ali da im je *treća koordinata uvijek jednaka nuli!* Ipak, neovisna realizacija klase “Vektor2” i “Vektor3” je vjerovatno najbolje rješenje.

Tipična greška u razmišljanju nastaje kada programer pokušava da relaciju “sadrži” izvede preko nasljeđivanja, odnosno da nasljeđivanje izvede samo zbog toga što neka klasa posjeduje sve što posjeduje i neka već postojića klasa. Na primjer, ukoliko utvrdimo da neka klasa “B” konceptualno treba da *sadrži* klasu “A”, velika greška u pristupu je takav odnos izraziti tako što će klasa “B” naslijediti klasu “A”. Čak vrijedi obnuto, odnosno to je praktično siguran znak da klasa “B” *ne treba* da bude naslijedena iz klase “A”. Umjesto toga, klasa “B” treba da sadrži *atribut koji je tipa “A”*. Na primjer, klasa “Datum” sigurno će imati attribute koji čuvaju pripadni dan, mjesec i godinu. Klasa “Student” mogla bi imati te iste attribute (koji bi mogli čuvati dan, mjesec i godinu rođenja studenta), i iste metode koje omogućavaju pristup tim atributima, ali to definitivno *ne znači* da klasu “Student” treba naslijediti iz klase “Datum” (s obzirom da student *nije* datum). Pravo rješenje je u klasi “Student” umjesto posebnih atributa za dan, mjesec i godinu rođenja koristiti jedan atribut tipa “Datum” koji opisuje datum rođenja. Na taj način postižemo da klasa “Student” *sadrži* klasu “Datum”. Ovakav tip odnosa između dvije klase, u kojem jedna klasa sadrži drugu, a koji smo već ranije koristili, naziva se *agregacija* i suštinski se razlikuje od nasljeđivanja.

Nakon što smo objasnili kad treba, a kad ne treba koristiti nasljeđivanje, možemo reći *kako* se ono ostvaruje, i *šta se njim postiže*. Pretpostavimo da imamo klasu “Student”, u kojoj ćemo, radi jednostavnosti, definirati samo attribute koji definiraju ime studenta (sa prezimenom) i broj indeksa, konstruktor sa dva parametra (ime i broj indeksa) kao i neke posve elementarne metode:

```

class Student {
    string ime;
    int indeks;
public:
    Student(string ime, int ind) : ime(ime), indeks(ind) {}
    string VratiIme() const { return ime; }
    int VratiIndeks() const { return indeks; }
    void Ispisi() const {
        cout << "Student " << ime << " ima indeks " << indeks;
    }
};

```

Ukoliko želimo da definiramo klasu “DiplomiraniStudent” koja je nasljedena iz klase “Student” a posjeduje novi atribut koji predstavlja godinu diplomiranja i odgovarajuću metodu za pristup tom atributu, to možemo uraditi na sljedeći način:

```

class DiplomiraniStudent : public Student {
    int godina_diplomiranja;
public:
    DiplomiraniStudent(string ime[], int ind, int god_dipl)
        : Student(ime, ind), godina_diplomiranja(god_dipl) {}
    int VratiGodinuDiplomiranja() const { return godina_diplomiranja; }
};

```

Nasljeđivanje u jeziku C++ se realizira putem mehanizma koji se naziva *javno izvođenje* (engl. *public derivation*), koji se postiže tako što u deklaraciji klase iza naziva klase stavimo dvotačku iza koje slijedi ključna riječ “**public**” i ime bazne klase. U izvedenoj klasi treba deklarirati samo attribute ili metode koje dodajemo (ili metode koje *mijenjam*, što ćemo uskoro demonstrirati). Svi ostali attribute i metode prosto se nasljeđuju iz bazne klase. Međutim, ovdje postoji jedan važan izuzetak: *konstruktori se nikada ne nasljeđuju*. Ukoliko je bazna klasa posjedovala konstruktoare, izvedena klasa ih mora ponovo definirati. Razlog za ovo je činjenica da su konstruktori namijenjeni da *inicijaliziraju* elemente objekta, a objekti nasljedene klase gotovo uvijek imaju dodatne attribute koje konstruktori bazne klase ne mogu da inicijaliziraju (s obzirom da ne postoje u baznoj klasi). Ukoliko zaboravimo definirati konstruktor u izvedenoj klasi, tada će u slučaju da bazna klasa posjeduje konstruktor bez parametara, taj konstruktor biti iskorišten da inicijalizira one attribute koji postoje u baznoj klasi, dok će novododani attribute ostati neinicijalizirani. U svim ostalim slučajevima kompjajler će prijaviti grešku. U navedenom primjeru, u klasi “DiplomiraniStudent” definiran je konstruktor sa tri parametra, koji pored imena i broja indeksa zahtijeva da zadamo i godinu diplomiranja studenta.

Konstruktor izvedene klase gotovo uvijek treba da odradi sve što je radio i konstruktor bazne klase, a nakon toga da odradi akcije specifične za izvedenu klasu. Da bismo izbjegli potrebu za prepisivanjem koda, konstruktor izvedene klase gotovo po pravilu treba da *pozove konstruktor bazne klase*. To se može izvesti jedino u *konstruktorskoj inicijalizacijskoj listi*, tako što se navede ime bazne klase i u zagradama parametri koje treba proslijediti konstruktoru bazne klase. Tako, u navedenom primjeru, konstruktor klase “DiplomiraniStudent” poziva konstruktor klase “Student” da inicijalizira attribute “ime” i “indeks”, a pored toga (također u konstruktorskoj inicijalizacijskoj listi) inicijalizira i svoj specifični atribut “godina_diplomiranja”.

Važno je naglasiti da atributi i metode koji su privatni u baznoj klasi nisu dostupni čak ni metodama klase koja je iz nje nasljeđena. Drugim riječima, metode klase “DiplomiraniStudent” *nemaju direktni pristup* atributima “ime” i “indeks”, iako ih ova klasa sadrži! Da nije tako, zlonamjerni programer bi veoma jednostavno mogao dobiti pristup privatnim elementima neke klase tako što bi prosto definirao novu klasu koja nasljeđuje tu klasu, nakon čega bi koristeći metode nasljeđene klase mogao pristupati privatnim elementima bazne klase. Upravo zbog toga, konstruktor klase “DiplomiraniStudent” *nismo mogli napisati* kao u sljedećem primjeru, jer bi kompjuter prijavio da atributi “ime” i “indeks” nisu dostupni unutar ove klase:

```
class DiplomiraniStudent : public Student {
    int godina_diplomiranja;
public:
    DiplomiraniStudent(string ime[], int ind, int god_dipl) {
        indeks = ind; godina_diplomiranja = god_dipl;
        Student::ime = ime;
    }
    int VratiGodinuDiplomiranja() const { return godina_diplomiranja; }
};
```

Činjenica da privatni elementi bazne klase nisu dostupni metodama izvedene klase često može da bude veliko ograničenje. Zbog toga je pored prava pristupa koji se definiraju pomoću ključnih riječi “**private**” (koja označava elemente koji su dostupni samo funkcijama članicama klase u kojima su definirani, i funkcijama i klasama koje su deklarirane kao prijatelji te klase) i “**public**” (koja označava elemente koji su dostupni u čitavom programu) uveden i treći tip prava pristupa, koji se po pravima nalazi negdje između ova dva tipa. Ovaj tip prava pristupa naziva se *zaštićeni pristup*, a definira se pomoću ključne riječi “**protected**”. Metodi i atributi čija su prava pristupa označena sa “**protected**” ponašaju se slično metodama i atributima sa privatnim pravom pristupa, ali su dostupni i metodama *svih klasa nasljeđenih iz klase u kojoj su definirani*. Stoga, ukoliko bismo atributima “ime” i “indeks” dali zaštićeno pravo pristupa, sve metode klase “DiplomiraniStudent”

(uključujući i konstruktor) moglo bi im pristupiti, i definicija konstruktora poput gore prikazane postala bi korektna (u navedenom primjeru to i nije previše bitno, jer je prethodna definicija konstruktora koja poziva konstruktor bazne klase svakako elegantnija). Slijedi modificirana verzija klase "Student", u kojoj atributi "ime" i "indeks" imaju zaštićeno pravo pristupa:

```
class Student {  
protected:  
    string ime;  
    int indeks;  
  
public:  
    Student(string ime, int ind) : ime(ime), indeks(ind) {}  
    string VratiIme() const { return ime; }  
    int VratiIndeks() const { return indeks; }  
    void Ispisi() const {  
        cout << "Student " << ime << " ima indeks " << indeks;  
    }  
};
```

U nastavku ćemo podrazumijevati da su atributima "ime" i "indeks" data zaštićena prava pristupa. Ipak, bez obzira na ova nešto slobodnija prava pristupa, bitno je naglasiti da se konstruktor klase "DiplomiraniStudent" nije mogao napisati ovako, jer je u konstruktorskoj inicijalizacijskoj listi moguće inicijalizirati samo attribute koji su neposredno deklarirani u toj klasi, bez obzira na prava pristupa atributa u baznoj klasi:

```
class DiplomiraniStudent : public Student {  
int godina_diplomiranja;  
  
public:  
    DiplomiraniStudent(string ime, int ind, int god_dipl)  
        : indeks(ind), ime(ime), godina_diplomiranja(god_dipl) {}  
    int VratiGodinuDiplomiranja() const { return godina_diplomiranja; }  
};
```

Nad objektima nasljedene klase mogu se koristiti sve metode kao i nad objektima bazne klase. Tako su sljedeće konstrukcije sasvim korektne:

```
Student s1("Paja Patak", 1234);  
DiplomiraniStudent s2("Miki Maus", 3412, 2004);  
s1.Ispisi();
```

```
cout << endl;
s2.Ispisi();
```

Ovaj primjer dovešće do sljedećeg ispisa:

Student Paja Patak ima indeks 1234
Student Miki Maus ima indeks 3412

S druge strane, često se javlja potreba da se u nasljđenoj klasi *promijene* definicije nekih metoda koje su definirane u baznoj klasi. Na primjer, u klasi “DiplomiraniStudent” ima smisla promijeniti definiciju metodu “*Ispisi*” tako da ona uzme u obzir i godinu diplomiranja studenta. Izmjenjena klasa mogla bi izgledati ovako:

```
class DiplomiraniStudent : public Student {
    int godina_diplomiranja;
public:
    DiplomiraniStudent(string ime, int Ind, int god_dipl)
        : Student(ime, ind), godina_diplomiranja(god_dipl) {}
    int VratiGodinuDiplomiranja() const { return godina_diplomiranja; }
    void Ispisi() const {
        cout << "Student " << ime << ", diplomirao " << godina_diplomiranja
            << ". godine, ima indeks " << indeks;
    }
};
```

Primijetimo da ovakva izmjena ne bi bila legalna da atributima “*ime*” i “*indeks*” nije dato zaštićeno pravo pristupa (u suprotnom bismo morali koristiti metode “*VratiIme*” i “*VratiIndeks*” da pristupimo ovim atributima). Uz prikazanu izmjenu, prethodni primjer doveo bi do sljedećeg ispisa:

Student Paja Patak ima indeks 1234
Student Miki Maus, diplomirao 2004. godine, ima indeks 3412

Drugim riječima, nad objektom “*s2*”, koji je tipa “DiplomiraniStudent”, poziva se njegova vlastita metoda “*Ispisi*”, a ne metoda nasljđena iz klase “Student”. Nasljđena verzija metode “*Ispisi*” i dalje je dostupna u klasi “DiplomiraniStudent”, ali ukoliko iz bilo kojeg razloga želimo pozvati upravo nju, tu želju moramo eksplicitno naznačiti pomoću operadora “*: :*”. Tako, ukoliko bismo nad objektom “*s2*” željeli da pozovemo metodu “*Ispisi*”

nasljedenu iz klase "Student", a ne istoimenu metodu definiranu u klasi "DiplomiraniStudent", trebali bismo pisati

```
s2.Student::Ispisi();
```

Na isti način je moguće u nekoj od metoda koje modificiramo u nasljedenoj klasi pozvati istoimenu metodu nasljedenu iz bazne klase. Na primjer, u sljedećoj definiciji klase "DiplomiraniStudent" metoda "Ispisi" poziva istoimenu metodu nasljedenu iz njene bazne klase:

```
class DiplomiraniStudent : public Student {
    int godina_diplomiranja;
public:
    DiplomiraniStudent(string ime, int int, int god_dipl)
        : Student(ime, ind), godina_diplomiranja(god_dipl) {}
    int VratiGodinuDiplomiranja() const { return godina_diplomiranja; }
    void Ispisi() const {
        Student::Ispisi();
        cout << ", a diplomirao je " << godina_diplomiranja << ". godine";
    }
}
```

Sada bi ranije navedeni primjer upotrebe ovih klasa doveo do sljedećeg ispisa:

Student Paja Patak ima indeks 1234

Student Miki Maus ima indeks 3412, a diplomirao je 2004. godine

Treba napomenuti da je kvalifikator "Student::" ispred poziva metode "Ispisi" veoma bitan, jer bi bez njega kompjaler shvatio da metoda "Ispisi" poziva samu sebe, što bi bilo protumačeno kao beskonačna rekurzija (tj. rekurzija bez izlaza).

Već smo rekli da se konstruktori ne nasljeđuju, nego da izvedena klasa uvijek mora definirati svoje konstruktore (uključujući i konstruktor kopije, ukoliko ga je bazna klasa imala). Zbog toga se ne nasljeđuju ni automatske pretvorbe tipova koje se ostvaruju eventualnim konstruktorima sa jednim parametrom (koji nisu označeni sa "**explicit**"), nego ih nasljedena klasa mora ponovo definirati ukoliko želi da zadrži mogućnost automatske pretvorbe u objekte izvedene klase. Pored konstruktora, jedina svojstva bazne klase koja se ne nasljeđuju su *operatorska funkcija za operator dodjele "=" i deklaracije prijateljstva*. Prekopljeni operator dodjele se ne nasljeđuje zbog toga što je njegovo ponašanje obično tijesno vezano uz

konstruktor kopije, koji se ne nasljeđuje. Dalje, deklaracije prijateljstva se ne nasljeđuju iz prostog razloga što bi njihovo automatsko nasljeđivanje omogućilo razne zloupotrebe. Stoga, ukoliko je neka funkcija “*f*” deklarirana kao prijatelj klase “A”, ona nije ujedno i prijatelj klase “B” nasljeđene iz klase “A”. Ukoliko želimo da zadržimo prijateljstvo i u nasljeđenoj klasi, funkciju “*f*” ponovo treba proglašiti prijateljem klase unutar deklaracije klase “B”. Osim konstruktora, operatora dodjele i prijateljstva, svi ostali elementi bazne klase nasljeđuju se u izvedenoj klasi (uključujući i operatorske funkcije).

Sasvim je moguće da više klasa bude nasljeđeno iz iste bazne klase. Pored toga, možemo imati i čitav lanac nasljeđivanja. Na primjer, klasa “C” može biti nasljeđena iz klase “B”, koja je opet nasljeđena iz klase “A”. Konačno, moguće je da neka klasa naslijedi više klasa, tj. da bude nasljeđena iz više od jedne bazne klase. Na primjer, sljedeća konstrukcija

```
class C : public A, public B {  
    ...  
};
```

deklarira klasu “C” koja je nasljeđena iz baznih klasa “A” i “B”. U ovom slučaju se radi o tzv. *višestrukom nasljeđivanju*. Mada višestruko nasljeđivanje može ponekad biti i korisno, ono sa sobom vuče mnoge nedoumice koje se moraju posebno razmotriti (npr. šta se dešava ukoliko više od jedne bazne klase imaju neke attribute ili metode istih imena, zatim šta se dešava ukoliko su bazne klase također izvedene klase, ali koje su izvedene iz jedne te iste bazne klase, itd.). Zbog toga se ovdje nećemo zadržavati na višestrukom nasljeđivanju, tim prije što nisu rijetka mišljenja da višestruko nasljeđivanje često unosi više zbrke nego što donosi koristi.

Već je rečeno da se svaki objekat nasljeđene klase može koristiti u kontekstima u kojima se može koristiti objekat bazne klase. Tako se objekat bazne klase može inicijalizirati objektom nasljeđene klase, zatim može se izvršiti dodjela objekta nasljeđene klase objektu bazne klase, i konačno, funkcija koja prihvata kao parametar ili vraća kao rezultat objekat bazne klase može prihvatiti kao parametar ili vratiti kao rezultat objekat nasljeđene klase. Ovo je dozvoljeno zbog toga što svaki objekat nasljeđene klase sadrži sve elemente koje sadrže i objekti bazne klase. Međutim, da bi ovakva mogućnost imala smisla, veoma je važno se svaki objekat nasljeđene klase može ujedno shvatiti i kao objekat bazne klase, što smo već ranije naglasili. U suprotnom bismo mogli doći u potpuno neprirodne situacije. Na primjer, ukoliko bismo klasu “Student” naslijedili iz klase “Datum”, tada bi svaka funkcija koja prima objekat klase Datum prihvatala i objekat tipa “Student” kao parametar, što teško da može imati smisla. Još je bitno naglasiti da se u svim navedenim situacijama sve specifičnosti izvedene klase gube. Tako, ukoliko promjenljivu “*s2*” koja je tipa “DiplomiraniStudent” pošaljemo kao parametar nekoj funkciji koja kao parametar prima objekat tipa “Student”, podaci o godini diplomiranja iz promjenljive “*s2*” biće ignorirani. Također, ukoliko izvršimo dodjelu poput “*s1 = s2*” gdje je “*s1*” promjenljiva tipa “Student”, u promjenljivu “*s1*” se kopiraju samo ime i broj indeksa iz promjenljive “*s2*”, dok se informacija o godini diplomiranja ignorira (s obzirom da promjenljiva “*s1*”, koja je tipa “Student”, nije u stanju da ovu

informaciju prihvati).

Veoma važno je shvatiti da je odnos između bazne i naslijedene klase *strogo jednosmjeran*, tako da se u općem slučaju objekti bazne klase *ne mogu koristiti* u kontekstima u kojima se mogu koristiti objekti naslijedene klase, s obzirom da se, općenito posmatrano, objekti bazne klase ne mogu posmatrati kao objekti izvedene klase. Na primjer, svaki diplomirani student *jeste* student, ali svaki student *nije* diplomirani student. Stoga dodjela poput “ $s_2 = s_1$ ” gdje su “ s_1 ” i “ s_2 ” promjenljive iz prethodnog primjera nije legalna, iako dodjela “ $s_1 = s_2$ ” jeste. Ovo je sasvim razumljivo, jer promjenljiva “ s_2 ” ima dodatni atribut koji promjenljiva “ s_1 ” nema (godinu diplomiranja), pa je nejasno šta bi trebalo dodijeliti atributu “*godina_diplomiranja*” objekta “ s_2 ”. Zbog istog razloga, bilo koja funkcija koja kao parametar prima objekat tipa “DiplomiraniStudent” ne može primiti kao parametar objekat tipa “Student”, s obzirom da takva funkcija može definirati specifične radnje koje nisu moguće sa običnim studentima. Objekti bazne klase se mogu koristiti u kontekstima u kojima se koriste objekti izvedene klase jedino u slučaju da izvedena klasa posjeduje *neeksplicitni konstruktor sa jednim parametrom koji je tipa bazne klase*, koji omogućava pretvorbu tipa bazne klase u objekat izvedene klase. U tom slučaju, takav konstruktor će nedvosmisleno odrediti kako treba tretirati attribute izvedene klase koji ne postoje u baznoj klasi.

Ranije smo rekli da se nasljeđivanje u jeziku C++ ostvaruje pomoću mehanizma *javnog izvođenja*. Pored javnog izvođenja, jezik C++ poznaje još i *privatno izvođenje* (engl. *private derivation*) kao i *zaštićeno izvođenje* (engl. *protected derivation*), koji se realiziraju na sličan način kao i javno izvođenje, samo što se ispred imena bazne klase a iza dvotačke umjesto ključne riječi “**public**” navode ključne riječi “**private**” odnosno “**protected**”. I dok kod javnog izvođenja elementi osnovne klase koji su imali javna prava pristupa zadržavaju ta prava pristupa i u izvedenoj klasi, kod privatnog odnosno zaštićenog izvođenja elementi osnovne klase koji su imali javna prava pristupa u izvedenoj klasi će imati privatna odnosno zaštićena prava pristupa. To zapravo znači da izvedena klasa neće imati isti interfejs kao i bazna klasa, odnosno interfejs izvedene klase će činiti samo one metode koje su eksplisitno definirane kao metode sa javnim pristupom unutar izvedene klase (koje, naravno, mogu pozivati neke od metoda bazne klase). Odavde direktno slijedi da se tako izvedene klase ne mogu smatrati kao specijalni slučajevi bazne klase, niti se objekti tako izvedenih klasa mogu koristiti u istim kontekstima kao i objekti bazne klase. Stoga je jasno da se privatnim ili zaštićenim nasljeđivanjem *ne realizira nasljeđivanje*. Objekti klase koja je privatno odnosno zaštićeno izvedena iz bazne klase tretiraju se posve neovisno od objekata bazne klase, i nikakvo njihovo međusobno miješanje nije dozvoljeno.

Privatno ili zaštićeno izvođenje najčešće se koristi kada neka klasa (recimo, klasa “B”) dijeli neke izvedbene detalje sa nekom drugom klasom (recimo, klasom “A”), ali pri čemu se klasa “B” ne može tretirati kao specijalan slučaj klase “A”. U tom slučaju, može se prištediti na dupliranju kôda ukoliko se klasa “B” privatno izvede iz klase “A”. Na primjer, klasa “Vektor3” bi se mogla realizirati kao privatno izvedena iz klase “Vektor2”. Međutim, kako privatno nasljeđivanje nosi i svoje komplikacije, vjerovatno je najbolje klase “Vektor2” i “Vektor3” realizirati kao posve neovisne klase. U suštini, privatnim i zaštićenim izvođenjem

umjesto nasljeđivanja se zapravo realizira neka vrsta *agregacije* bazne klase u izvedenu. Preciznije, sve što se može postići privatnim ili zaštićenim nasljeđivanjem može se postići i klasičnom agregacijom, samo što se kod privatnog odnosno zaštićenog nasljeđivanja koristi drugačija sintaksa, analogna sintaksi koja se koristi kod nasljeđivanja, što u nekim slučajevima može biti dosta praktično. Kako se privatnim i zaštićenim izvođenjem ne realizira koncept nasljeđivanja, o tim vrstama izvođenja na ovom mjestu nećemo dalje govoriti.

Nasljeđivanje dobija svoju punu snagu i primjenu tek u kombinaciji sa *pokazivačima i referencama*. Naime, slično kao što se objektu bazne klase može dodijeliti objekat nasljeđene klase (uz neminovan gubitak specifičnosti koje nosi objekat izvedene klase), tako se i pokazivaču na tip bazne klase može dodijeliti adresa nekog objekta nasljeđene klase ili neki drugi pokazivač na tip nasljeđene klase (ovo vrijedi samo za *nasljeđivanje* odnosno *javno izvođenje* – prilikom privatnog odnosno zaštićenog izvođenja ova konvencija ne vrijedi). Također, svaka funkcija koja kao prima kao parametar ili vraća kao rezultat pokazivač na tip bazne klase može primiti kao parametar ili vratiti kao rezultat pokazivač na tip izvedene klase (ili adresu nekog objekta izvedene klase). Na primjer, neka su “*s1*” i “*s2*” deklarirani kao u prethodnim primjerima, i neka imamo deklaraciju

```
Student *pok1, *pok2;
```

Tada su sljedeće dodjele sasvim legalne, bez obzira što su oba pokazivača “*pok1*” i “*pok2*” deklarirani kao pokazivači na tip “*Student*”, a objekat “*s2*” je tipa “*DiplomiraniStudent*”:

```
pok1 = &s1;  
pok2 = &s2;
```

Razumije se da smo isto tako mogli odmah izvršiti inicializaciju prilikom deklariranja ovih pokazivača, tako da bismo isti efekat postigli deklaracijom

```
Student *pok1 = &s1, *pok2 = &s2;
```

Razmotrimo sada kako će se ponašati ovi pokazivači ukoliko nad objektima na koje oni ukazuju primijenimo neku metodu pomoću operatora “*->*”, na primjer, sljedećom konstrukcijom:

```
pok1->Ispisi();  
cout << endl;  
pok2->Ispisi();
```

Ovaj primjer će dovesti do sljedećeg ispisa:

Student Paja Patak ima indeks 1234

Student Miki Maus ima indeks 3412

Vidimo da je u oba slučaja pozvana metoda “Ispisi” iz klase “Student”, bez obzira što pokazivač “pok2” pokazuje na objekat tipa “DiplomiraniStudent”. Kompajler je odluku o tome koju metodu treba pozvati donio na osnovu toga kako su *deklarirani* pokazivači “pok1” i “pok2”, a ne na osnovu toga na šta oni pokazuju. Ovakva strategija naziva se *rano povezivanje* (engl. *early binding*), s obzirom da kompjajler povezuje odgovarajući pokazivač sa odgovarajućom metodom samo na osnovu njegove deklaracije, što se može izvršiti još u fazi prevođenja programa, prije nego što se program počne izvršavati.

Zbog ranog povezivanja, ponovo izgleda kao da su sve informacije o specifičnosti objekata tipa “DiplomiraniStudent” izgubljene kada je pokazivač na tip “Student” dodijeljena adresa objekta tipa “DiplomiraniStudent”. Kada se ne koriste pokazivači, jasno je da do gubljenja specifičnosti mora doći, jer objekat bazne klase ne može da prihvati sve specifičnosti objekta izvedene klase (koji sadrži više informacija). S druge strane, također je jasno da u slučaju kada se koriste pokazivači, do ovakvog gubljenja specifičnosti *ne bi moralо doći*. Zaista, bez obzira što je “pok2” pokazivač na tip “Student”, on je usmјeren da pokazuje na objekat tipa “DiplomiraniStudent”, koji *postoji*, i nalazi se u memoriji, tako da ne postoji nikakav gubitak informacija. Međutim, da ne bi došlo do gubljenja specifičnosti izvedene klase (odnosno da bi poziv metode “Ispisi” nad objektom na koji pokazuje pokazivač “pok2” ispisao informacije kao da se radi o diplomiranom a ne običnom studentu), odluku o tome koja se metoda “Ispisi” poziva treba donijeti na osnovu toga *na šta pokazivač zaista pokazuje*, a ne na osnovu toga *kako je on deklariran*. Drugim riječima, odluku o tome koja se metoda poziva ne treba se donositi unaprijed, u fazi prevođenja programa, nego je treba odgoditi do samog trenutka poziva metode (u vrijeme izvršavanja programa), jer u općem slučaju tek tada može biti poznato na šta pokazivač zaista pokazuje (Što ćemo vidjeti u jednom od narednih primjera). Ova strategija naziva se *kasno povezivanje* (engl. *late binding*).

Da bismo ostvarili kasno povezivanje, ispred deklaracije metode na koju želimo da se primjenjuje kasno povezivanje treba dodati ključnu riječ “**virtual**”. Metode koje se pozivaju strategijom kasnog povezivanja nazivaju se *virtualne metode*, odnosno *virtualne funkcije članice*. Stoga ćemo metodu “Ispisi” u klasi “Student” deklarirati tako da postane virtualna metoda:

```
class Student {  
protected:  
    string ime;  
    int indeks;  
public:  
    Student(string ime[], int ind) : ime(ime), indeks(ind) {}
```

```

char VratiIme() const { return ime; }

int VratiIndeks() const { return indeks; }

virtual void Ispisi() const {
    cout << "Student " << ime << " ima indeks " << indeks;
}

};

```

Sa ovakvom izmjenom, indirektni poziv metode “*Ispisi*” pomoću operatora “->” nad pokazivačima “*pok1*” i “*pok2*” dovešće do željenog ispisa

Student Paja Patak ima indeks 1234

Student Miki Maus ima indeks 3412, a diplomirao je 2004. godine

Bitno je naglasiti da se ključna riječ “**virtual**” piše samo unutar deklaracije klase, tako da u slučaju da virtualnu metodu implementiramo izvan deklaracije klase, ključnu riječ “**virtual**” ne trebamo (i ne smijemo) ponavljati.

Slična logika vrijedi i za reference. Referenca na objekat bazne klase može se vezati za objekat naslijedene klase, što predstavlja izuzetak od pravila da se reference mogu vezati samo za objekat istog tipa. Naravno, ovo vrijedi i za formalne parametre funkcija koji su reference (uključujući i reference na konstantne objekte). Pri tome, također ne mora doći ni do kakvog gubitka informacija, jer su reference u suštini sintaksna varijanta pokazivača. Stoga, mehanizam virtualnih funkcija radi i sa referencama. Pretpostavimo, na primjer, da imamo deklariranu sljedeću funkciju, čiji je formalni parametar referencia na objekat tipa “*Student*”:

```

void NekaFunkcija(const Student &s) {
    s.Ispisi();
}

```

Prepostavimo dalje da je “*s2*” objekat tipa “*DiplomiraniStudent*”, kao u ranijim primjerima. Tada je sljedeći poziv posve legalan, bez obzira na nepodudarnost tipova formalnog i stvarnog argumenta:

```
NekaFunkcija(s2);
```

Pri tome će, ukoliko je metoda “*Ispisi*” unutar klase “*DiplomiraniStudent*” deklarirana kao virtualna metoda, iz funkcije “*NekaFunkcija*” biti pozvana metoda “*Ispisi*” iz klase “*DiplomiraniStudent*” bez obzira što je formalni parametar “*s*” deklariran kao referencia na objekat tipa “*Student*”. Ovo je postignuto zahvaljujući kasnom povezivanju. Do ovoga neće

doći ukoliko metoda “*Ispisi*” nije virtualna, nego će se uvijek pozvati metoda “*Ispisi*” iz klase “Student”, bez obzira na tip stvarnog parametra.

Treba napomenuti da mehanizam virtualnih funkcija i kasnog povezivanja nikada ne djeluje kada se neki objekat naslijedene klase *kopira* u drugi objekat bazne klase. Naime, takvim kopiranjem gube se sve specifičnosti objekta izvedene klase, tako da se mehanizam virtualnih funkcija *ne smije primijeniti* (s obzirom da se odgovarajuća virtualna metoda iz naslijedene klase skoro sigurno oslanja na specifičnosti naslijedene klase). Tako, ukoliko bi u prethodnom primjeru formalni parametar “*s*” umjesto kao referenca na objekat tipa “Student” bio deklariran prosti kao objekat tipa “Student” (tj. ukoliko bi se parametar prenosio *po vrijednosti*), mehanizam virtualnih funkcija ne bi djelovao, i uvijek bi se pozivala metoda “*Ispisi*” iz klase “Student”. Naime, pri prenosu po vrijednosti, dolazilo bi do kopiranja stvarnog parametra u formalni, čime bi sve eventualne specifičnosti stvarnog parametra bile izgubljene, tako da mehanizam virtualnih funkcija ne bi smio djelovati.

Kasno povezivanje i virtualne metode omogućavaju mnogobrojne interesantne primjene. Na primjer, razmotrimo sljedeći programski isječak:

```
Student *s = new Student("Paja Patak", 1234);
s->Ispisi();
cout << endl;
delete s;
s = new DiplomiraniStudent("Miki Maus", 3412, 2004);
s->Ispisi();
delete s;
```

Ovaj primjer dovodi do istog ispisa kao i ranije prikazani primjer sa pokazivačkim promjenljivim “*pok1*” i “*pok2*”. Međutim, u ovom primjeru se u oba slučaja metoda “*Ispisi*” poziva (indirektno) nad *istom promjenljivom* “*s*”, pri čemu se prvi put ova promjenljiva ponaša poput objekta klase “Student”, a drugi put poput objekta klase “DiplomiraniStudent” (ako zanemarimo činjenicu da je “*s*” zapravo pokazivač na objekat, a ne sam objekat, kao i činjenicu da zbog toga metodu “*Ispisi*” pozivamo indirektno operatorom “*->*” a ne direktno operatorom “*.*”). Metodologija koja omogućava da se ista promjenljiva u različitim trenucima ponaša kao da ima različite tipove, tako da poziv *iste metode* nad *istom promjenljivom u različitim trenucima* izaziva *različite akcije* naziva se *polimorfizam*. Preciznije, ovdje govorimo o tzv. *jakom polimorfizmu*, s obzirom da se definira i tzv. *slabi polimorfizam*, koji prosti podrazumijeva da promjenljive različitih tipova mogu imati metode istog imena koje rade različite stvari, tako da primjena iste metode nad promjenljivim različitog tipa izaziva različite akcije. U nastavku kada budemo govorili o polimorfizmu, mislićemo uglavnom na jaki polimorfizam, ukoliko eksplisitno ne naglasimo drugačije.

U prethodnom slučaju, "s" je tipičan primjer *polimorfne promjenljive*. U jeziku C++ samo pokazivačke promjenljive i reference mogu biti polimorfne, pri čemu reference mogu djelovati još ubjedljivije kao polimorfne promjenljive nego pokazivači, s obzirom da se za poziv metoda nad referencom koristi isti operator “.” kao za poziv metoda nad običnim objektima. Polimorfizam je ključna metodologija objektno orijentiranog programiranja, koja čak ne mora nužno biti vezana za nasljeđivanje. Međutim, u jeziku C++ nasljeđivanje i virtualne metode predstavljaju osnovno sredstvo pomoću kojeg se realizira polimorfizam, mada teoretski postoje i drugi načini (npr. pomoću pokazivača na funkcije, kao što ćemo vidjeti na kraju ovog poglavlja).

Već smo rekli da se polimorfno ponašanje neke promjenljive tipično realizira tako da se ona deklarira kao pokazivač na neku baznu klasu koja sadrži makar jednu virtualnu metodu, čija je definicija promijenjena u nekoj od klasa nasljeđenih iz te bazne klase. U tom slučaju se odluka o tome koju zaista metodu treba pozvati (tj. iz koje klase) odgađa sve do samog trenutka poziva. Da je to zaista neophodno demonstrira sljedeći primjer, u kojem se odluka o tome koja se od dvije metode "Ispisi" (iz klase "Student" ili iz klase "DiplomiraniStudent") ne može donijeti prije samog trenutka poziva, jer odluka zavisi od podataka koje korisnik unosi sa tastature, koji se ne mogu unaprijed predvidjeti:

```
string ime;
int indeks, god_dipl;
cout << "Unesi ime i prezime studenta:";
getline(cin, ime);
cout << "Unesi broj indeksa studenta:";
cin >> indeks;
cout << "Unesi godinu diplomiranja ili 0 ukoliko student "
    << "još nije diplomirao:";
cin >> god_dipl;
Student *s;
if(god_dipl == 0) s = new Student(ime, indeks);
else s = new DiplomiraniStudent(ime, indeks, god_dipl);
s->Ispisi();
```

Jedna od tipičnih primjena kasnog povezivanja i polimorfizma je kreiranje *heterogenih nizova* (ili, općenitije, *heterogenih kontejnerskih tipova*) koji kao svoje elemente mogu pravidno sadržavati objekte *različitih tipova*. Na primjer, razmotrimo sljedeću sekvencu naredbi:

```
Student *studenti[5];
studenti[0] = new Student("Paja Patak", 1234);
```

```
studenti[1] = new DiplomiraniStudent("Miki Maus", 3412, 2004);
studenti[2] = new Student("Duško Dugouško", 4123);
studenti[3] = new Student("Tom Mačak", 2341);
studenti[4] = new DiplomiraniStudent("Džeri Miš", 4321, 1997);
```

Strogo posmatrano, ove naredbe deklariraju niz nazvan "studenti" od pet pokazivača na objekte tipa "Student", čijim elementima kasnije dodjeljujemo adrese pet dinamički stvorenih objekata, od kojih su tri tipa "Student", a dva tipa "DiplomiraniStudent" (ovo je posve legalno, s obzirom da se pokazivaču na objekat neke klase smije legalno dodijeliti pokazivač na objekat ma koje klase naslijedene iz te klase). Međutim, sjetimo se da se nizovi pokazivača na primjerke neke klase mogu koristiti skoro identično kao da se radi o nizovima čiji su elementi primjeri te klase, samo što za pristup atributima i metodama umjesto operatora "." koristimo operator "->". Dalje, kako je metoda "Ispisi" virtuelna, njena primjena nad pojedinim pokazivačima dovodi do različitog dejstva u zavisnosti od toga na kakav objekat konkretni pokazivač pokazuje. Stoga će naredba poput

```
for(int i = 0; i < 5; i++) studenti[i]->Ispisi();
```

dovesti do sljedećeg ispisa:

Student Paja Patak ima indeks 1234

Student Miki Maus ima indeks 3412, a diplomirao je 2004. godine

Student Duško Dugouško ima indeks 4123

Student Tom Mačak ima indeks 2341

Student Džeri Miš ima indeks 4321, a diplomirao je 1997. godine

Ako zanemarimo sitni detalj da koristimo operator "->" umjesto operatora ".", vidimo da možemo zamisliti kao da se niz "studenti" zapravo ponaša kao heterogeni niz čiji članovi mogu biti kako obični, tako i diplomirani studenti, odnosno ponaša se kao da se radi o nizu čiji su elementi različitog tipa. Vidimo da ovdje presudnu ulogu igraju virtualne metode. Da metoda "Ispisi" nije bila virtualna, svi elementi niza bi se tretirali na identičan način, kao da se radi o objektima klase "Student", u skladu sa načinom na koji je niz deklariran.

Važno je naglasiti da, bez obzira na polimorfizam, ne možemo preko pokazivača na baznu klasu pozvati neku od metoda koja postoji samo u naslijedenoj klasi a ne i u baznoj klasi, čak i ukoliko taj pokazivač trenutno pokazuje na objekat naslijedene klase (analogna primjedba vrijedi i za reference). Na primjer, sljedeća konstrukcija nije dozvoljena, zbog toga što je "s" deklariran kao pokazivač na klasu "Student" koja ne posjeduje metodu "VratiGodinuDiplomiranja", bez obzira što je on inicijaliziran tako da pokazuje na objekat klase "DiplomiraniStudent":

```

Student *s = new DiplomiraniStudent("Miki Maus", 3412, 2004);
cout << s->VratiGodinuDiplomiranja();

```

Ovo nije dozvoljeno zbog toga što u trenutku prevođenja programa prevodilac nema garancije da promjenljiva zaista pokazuje na objekat koji posjeduje ovu metodu. Zaista, kada bi bio dozvoljen poziv metode "VratiGodinuDiplomiranja" nad promjenljivom "s", moglo bi doći do velikih problema, s obzirom na činjenicu da je "s" pokazivač na tip "Student", tako da može pokazivati na objekat koji uopće ne sadrži podatak o godini diplomiranja. Stoga su autori jezika C++ usvojili da ovakvi pozivi ne budu dozvoljeni. Ukoliko nam je baš neophodno da nad pokazivačem koji je deklariran da pokazuje na baznu klasu primijenimo neku metodu koja je definirana samo u naslijедenoj klasi, a *sigurni smo* da u posmatranom trenutku taj pokazivač *zaista pokazuje na objekat izvedene klase*, možemo na pokazivač primijeniti eksplisitnu konverziju tipa u tip pokazivača na izvedenu klasu pomoću operadora za pretvorbu tipa (type-casting operatora), a zatim na rezultat pretvorbe primijeniti metodu koju želimo. Na primjer, nad promjenljivom "s" iz prethodnog primjera mogli bismo uraditi sljedeće:

```

cout << ((DiplomiraniStudent *) s)->VratiGodinuDiplomiranja();

```

Ovakve konstrukcije preduzimamo *na vlastitu odgovornost*, s obzirom da posljedice mogu biti posve nepredvidljive ukoliko "s" ne pokazuje na objekat koji je tipa "DiplomiraniStudent". Sličnu konverziju morali bismo izvršiti ukoliko je "s" polimorfna referenca (samo bismo u oznaci tipa prilikom konverzije umjesto znaka "*" pisali znak "&" čime naznačavamo da vršimo konverziju ponovo u referencu). Ukoliko nam je potrebno da u toku izvršavanja programa *ispitamo* na šta pokazuje neki polimorfni pokazivač (ili referenca), možemo koristiti operator "**typeid**", koji se koristi na način koji je sasvim jasan iz sljedećeg primjera:

```

if(typeid(*s) == typeid(DiplomiraniStudent))
    cout << ((DiplomiraniStudent *) s)->VratiGodinuDiplomiranja();
else
    cout << "Žalim, s ne pokazuje na diplomiranog studenta!";

```

Bez obzira na činjenicu da pretvorba pokazivačkih tipova iz jednog u drugi i "**typeid**" operator mogu ponekad biti korisni, pa čak i neophodni (inače ne bi bili ni uvedeni), njihova intenzivna upotreba gotovo sigurno ukazuje na pogrešan pristup problemu koji se rješava.

Mada je pokazivaču na baznu klasu moguće dodijeliti pokazivač na objekat naslijedene klase odnosno adresu nekog objekta naslijedene klase, obrнута dodjela *nije moguća*. Tako, pokazivaču na objekat naslijedene klase nije moguće dodijeliti pokazivač na objekat bazne klase, odnosno adresu nekog objekta bazne klase. Također, funkcija koja prima kao parametar ili vraća kao rezultat pokazivač na objekat naslijedene klase ne može prihvati kao parametar ili vratiti kao rezultat pokazivač na objekat bazne klase. Analogno vrijedi i za reference. Razlog za ovu zabranu je veoma jednostavan: preko pokazivača (ili reference) na naslijedenu

klasu može se pristupiti atributima i metodama koje u baznoj klasi uopće ne moraju postojati, pa bi mogućnost dodjele adrese nekog objekta bazne klase pokazivaču na naslijedenu klasi moglo dovesti do kobnih posljedica. Ukoliko smo apsolutno sigurni šta radimo, ovakve dodjele ipak možemo izvesti korištenjem operatora za pretvorbu tipa, ali ako imamo i djelić sumnje u ono što radimo, ovakve vratolomije treba izbjegći po svaku cijenu.

Mada se virtualne funkcije obično koriste za realizaciju polimorfizma, sam koncept virtualnih funkcija i kasnog povezivanja nije nužno vezan za polimorfizam (mada je *uvijek* vezan za nasljeđivanje). Kako se radi o veoma važnom konceptu, koji čini jezgro objektno orijentirane filozofije, razmotrićemo jedan ilustrativan primjer koji uvodi virtualne funkcije nevezano od polimorfizma, a zatim ćemo isti primjer proširiti upotrebotom polimorfizma. Pretpostavimo da želimo deklarirati nekoliko klasa koji opisuju razne geometrijske likove, na primjer krugove, pravougaonike i trouglove. Svim likovima je zajedničko da posjeduju obim i površinu. Stoga možemo deklarirati klasu nazvanu “*Lik*” koja će sadržavati samo svojstva zajednička za sve likove, kao što je atribut “*Naziv*” koji će sadržavati naziv lika (“Krug”, “Trougao”, itd.), metode “*Obim*” i “*Povrsina*”, i metodu “*Ispisi*” koja ispisuje osnovne podatke o liku (naziv, obim i površina). Klasa “*Lik*” nije namijenjena da se koristi samostalno, nego isključivo kao bazna klasa za klasе “*Krug*”, “*Pravougaonik*” i “*Trougao*”, koje ćemo naslijediti iz klase “*Lik*”, čime zapravo izražavamo činjenicu da krugovi, pravougaonici i trouglovi jesu likovi. Pored toga, definiraćemo i klasu “*Kvadrat*” naslijedenu iz klase “*Pravougaonik*”, koja odražava činjenicu da su kvadrati specijalna forma pravougaonika, kao i klase “*PravougliTrougao*” i “*PravilniTrougao*” naslijedene iz klase “*Trougao*” koje redom predstavljaju pravougli i pravilni (jednakostranični) trougao kao specijalne forme trouglova. Na taj način dobijamo cijelu hijerarhiju klasa, kao na sljedećoj slici:

Odmah na početku moramo istaći da se mogu postaviti izvjesne dileme da li se klasa “*Kvadrat*” treba definirati kao klasa naslijedena iz klase “*Pravougaonik*”. S jedne strane, neosporna je činjenica da kvadrati jesu pravougaonici, tako da je prvi uvjet za svrshodnost nasljeđivanja ispunjen. Mnogo je spornije da li se kvadrati mogu koristiti u svim kontekstima kao i pravougaonici. Općenito gledano, odgovor na ovo pitanje je negativan. Kasnije ćemo vidjeti kakve implikacije unosi ovaj zaključak. Slična razmišljanja vrijede za odnos između

klasa “PravougliTrougao” odnosno “PravilniTrougao” i njima pretpostavljene bazne klase “Trougao”.

Razmotrimo sada moguće definicije ovih klasa. Krenimo prvo od definicije osnovne klase “Lik”. Kako ova klasa ne predstavlja nikakav konkretan lik, ne možemo specificirati nikakve konkretne atribute (osim naziva), niti ikakve konkretne postupke za računanje obima i poluprečnika. Međutim, kako ćemo u klasi “Lik” definirati metodu “Ispisi”, koju će sve ostale klase naslijediti, a koja poziva metode za računanje obima i površine, te metode obavezno moramo definirati i u klasi “Lik”, pa makar i ne radile ništa smisleno. Usvojićemo da ove metode za klasu “Lik” prosto vraćaju nulu kao rezultat, a u svakoj od naslijedenih klasa definiraćemo prave postupke za računanje obima i površine, u skladu sa konkretnom vrstom lika. Stoga će prva verzija definicije klase “Lik” izgledati ovako (kasnije ćemo uvidjeti da će biti potrebne izvjesne modifikacije u deklaraciji ove klase da bi sve radilo kako treba):

```
class Lik {
protected:
    char naziv[20];
public:
    double Obim() const { return 0; }
    double Povrsina() const { return 0; }
    void Ispisi() const {
        cout << "Lik: " << naziv << endl << "Obim: " << Obim() << endl
            << " Površina: " << Povrsina() << endl;
    }
};
```

Sada možemo definirati i klasu “Krug”, naslijedenu iz klase “Lik”. U njoj ćemo deklarirati atribut “r” koji čuva poluprečnik kruga, konstruktor koji inicijalizira ovaj atribut, kao i metode “Obim” i “Povrsina” koje računaju obim i površinu kruga, a koje treba da zamijene istoimene metode iz bazne klase. Atribut “Naziv” i metodu “Ispisi” ova klasa nasljeđuje iz bazne klase “Lik”;

```
class Krug : public Lik {
    static const double PI = 3.141592654;
    double r;
public:
    Krug(double r) : r(r) { strcpy(naziv, "Krug"); }
    double Obim() const { return 2 * PI * r; }
    double Povrsina() const { return r * r * PI; }
};
```

Na sličan način ćemo definirati i klase “Pravougaonik” i “Trougao” (atributima klase “Trougao” dali smo zaštićena prava pristupa, s obzirom da će nam u klasi “PravougliTrougao” izvedenoj iz nje trebati mogućnost pristupa ovim atributima). Klasa “Pravougaonik” posjedovaće dva, a klasa “Trougao” tri atributa koji čuvaju dužine stranica. Tu su i odgovarajući konstruktori sa dva odnosno tri parametra za inicijalizaciju tih atributa, kao i odgovarajuće definicije metoda za računanje obima i površine (površinu trougla u funkciji dužine stranica računamo pomoću Heronovog obrasca):

```

class Pravougaonik : public Lik {
    double a, b;
public:
    Pravougaonik(double a, double b) : a(a), b(b) {
        strcpy(naziv, "Pravougaonik");
    }
    double Obim() const { return 2 * (a + b); }
    double Povrsina() const { return a * b; }
};

class Trougao : public Lik {
protected:
    double a, b, c;
public:
    Trougao(double a, double b, double c) : a(a), b(b), c(c) {
        strcpy(naziv, "Trougao");
    }
    double Obim() const { return a + b + c; }
    double Povrsina() const {
        double s = (a + b + c) / 2;
        return sqrt(s * (s - a) * (s - b) * (s - c));
    }
};

```

Sada ćemo iz klase “Pravougaonik” izvesti klasu “Kvadrat”, koja opisuje kvadrat kao specijalan slučaj pravougaonika. Jedino što ćemo promijeniti u ovoj klasi odnosu na njenu baznu klasu “Pravougaonik” je konstruktor (koji se svakako ne nasljeđuje), s obzirom da se kvadrat opisuje sa jednim, a ne sa dva parametra. Ovaj konstruktor postaviće oba atributa koji opisuju pravougaonik na iste vrijednosti, s obzirom da je kvadrat upravo pravougaonik sa jednakim stranicama. Svi ostali elementi (uključujući i metode za računanje obima i površine) mogu se prosti naslijediti, jer se obim i površina kvadrata mogu računati na isti način kao i za pravougaonik (ako uzmememo jednakе dužine stranica):

```

class Kvadrat : public Pravougaonik {
public:
    Kvadrat(double a) : Pravougaonik(a, a) { strcpy(naziv, "Kvadrat"); }
};

```

Primijetimo da bismo napravili veliku konceptualnu grešku da smo prvo definirali klasu “Kvadrat” a zatim iz nje izveli klasu “Pravougaonik”. Do ovakvog pogrešnog rezonovanja mogli bismo doći ukoliko bismo brzopleto zaključili da se kvadrat opisuje jednim atributom (dužinom stranice), a kvadrat sa dva atributa (dužinama dvaju stranica), tako da pravougaonik zahtijeva više atributa za opis nego kvadrat. Međutim, prava je činjenica da i kvadrat isto tako posjeduje sve attribute koje posjeduje i pravougaonik, samo su oni međusobno jednaki. Kvadrat je specijalan slučaj pravougaonika, a ne obrnuto, tako da klasa “Kvadrat” treba da bude izvedena a “Pravougaonik” bazna klasa. Svi kvadrati su pravougaonici, ali svi pravougaonici nisu kvadrati.

Na sličan način ćemo iz klase “Trougao” izvesti dvije nove klase “PravougliTrougao” i “PravilniTrougao”. I u ovom slučaju dovoljno bi bilo promijeniti samo konstruktore.

Međutim, u klasi “PravougliTrougao” smo promijenili i metodu za računanje površine. Naime, površina pravouglog trougla može se izračunati mnogo jednostavnije nego komplikovanom Heronovom formulom, tako da na ovaj način dobijamo na efikasnosti i tačnosti (jer Heronova formula može da akumulira greške uslijed zaokruživanja):

```
class PravougliTrougao : public Trougao {
public:
    PravougliTrougao(double a, double b)
        : Trougao(a, b, sqrt(a * a + b * b)) {
            strcpy(naziv, "Pravougli trougao");
    }
    double Povrsina() const { return a * b / 2; }
};

class PravilniTrougao : public Trougao {
public:
    PravilniTrougao(double a) : Trougao(a, a, a) {
        strcpy(naziv, "Pravilni trougao");
    }
};
```

Ovim smo definirali sve potrebne klase. Međutim, ukoliko bismo poželjeli da isprobamo napisane klase, veoma brzo bismo vidjeli da nešto nije u redu. Na primjer, pretpostavimo da smo sa napisanim klasama izvršili sljedeće naredbe:

```
Pravougaonik p(5, 4);
Krug k(3);

p.Ispisi();
cout << "O = " << p.Obim() << " P = " << p.Povrsina() << endl;
k.Ispisi();
cout << "O = " << k.Obim() << " P = " << k.Povrsina() << endl;
```

Ove naredbe dovele bi do sljedećeg ispisa na ekranu:

```
Lik: Pravougaonik
Obim: 0
Površina: 0
O = 18 P = 20
Krug
Obim: 0
Površina: 0
O = 18.849556 P = 28.274334
```

Vidimo da metode “Obim” i “Povrsina” same za sebe rade korektno, ali nešto nije u redu sa metodom “Ispisi”. Šta se zapravo dešava? Problem ponovo leži u *ranom povezivanju*. Naime, tačno je da klasa “Pravougaonik” posjeduje metodu “Ispisi” koja je naslijedena još iz klase “Lik”, ali je problem u tome što klasa “Lik” sadrži metode “Obim” i “Povrsina” koje vraćaju nulu kao rezultat, a kompjajler je već u fazi prevođenja povezao metodu “Ispisi” sa

metodama “Obim” i “Povrsina” iz iste klase, tako da metoda “Ispisi” zapravo poziva metode “Obim” i “Povrsina” iz klase “Lik”, bez obzira što su ove metode izmijenjene u klasi “Pravougaonik” odnosno “Krug”! Da bismo riješili ovaj problem, potrebno je odluku o tome koje metode “Obim” i “Povrsina” treba da se pozovu iz metode “Ispisi” *odgoditi do samog trenutka njihovog pozivanja*, odnosno koristiti *kasno povezivanje*. Na taj način će kada pozovemo metodu “Ispisi” nad objektom “p” koji je tipa “Pravougaonik”, unutar metode “Ispisi” biti pozvane upravo metode “Obim” i “Povrsina” iz klase “Pravougaonik”, s obzirom da se u tom trenutku zna da radimo nad objektom tipa “Pravougaonik” (što se na osnovu same deklaracije klase “Lik” nije moglo znati). Slično, kada pozovemo metodu “Ispisi” nad objektom “k” koji je tipa “Krug”, unutar metode “Ispisi” će biti pozvane metode “Obim” i “Povrsina” iz klase “Krug”. Kasno povezivanje je neophodno jer se očigledno u vrijeme prevodenja programa ne može znati nad kojim će objektom metoda “Ispisi” biti pozvana, pa prema tome ni koje metode “Obim” i “Povrsina” treba da se iz nje pozovu. Prema tome, rješenje je da metode “Obim” i “Povrsina” u klasi “Lik” proglašimo za *virtualne*, kao u sljedećoj deklaraciji klase “Lik”:

```
class Lik {
protected:
    char naziv[20];
public:
    virtual double Obim() const { return 0; }
    virtual double Povrsina() const { return 0; }
    void Ispisi() const {
        cout << "Lik: " << naziv << endl << "Obim: " << Obim() << endl
           << "Površina: " << Povrsina() << endl;
    }
};
```

Uz ovakvu izmjenu sve će raditi u skladu sa očekivanjima. U ovom primjeru potreba za kasnim povezivanjem i virtualnim metodama javila se neovisno od polimorfizma, koji se u ovom primjeru ne koristi.

Iz izloženog primjera možemo vidjeti da virtualne metode donose jednu suštinsku novost u odnosu na sve sa čime smo se ranije susretali. Naime, klasične funkcije su omogućavale da novonapisani dijelovi programa mogu da koriste stare dijelove programa (koristeći pozive funkcija), bez ikakve potrebe da mijenjamo već napisane dijelove programa. Međutim, virtualne funkcije nam nude upravo obrnuto: da *stari* (tj. već napisani) *dijelovi programa* bez ikakve potrebe za izmjenama mogu pozivati *dijelove programa koji će tek biti napisani!* Zaista, posmatrajmo metodu “Ispisi” iz prethodnog primjera. Ova metoda poziva metode “Obim” i “Povrsina”, ali kako se radi o virtualnim metotama, unaprijed se ne zna na koje se metode “Obim” i “Povrsina” (tj. iz koje klase) ti pozivi odnose sve do samog trenutka poziva, kada će to biti određeno objektom nad kojim se metoda “Ispisi” pozove. Sasvim je moguće da se kasnije odlučimo da u program dodamo podršku za nove likove (npr. elipse, kružne isječke, poligone, itd.). Ukoliko ove likove implementiramo kao klase naslijedene iz klase “Lik” i definiramo odgovarajuće metode za računanje obima i površine, metoda “Ispisi” (koja je već davno napisana) će u slučaju potrebe pozivati novonapisane metode, bez potrebe

da vršimo ikakve izmjene u samoj definiciji metode “*Ispisi*”. Na taj način, virtualne funkcije omogućavaju već napisanim dijelovima programa da se, na izvjestan način, automatski “adaptiraju” na nove okolnosti koje mogu nastati uslijed proširivanja programa!

Mogućnost da stari dijelovi programa bez ikakve potrebe za izmjenama mogu pozivati novonapisane dijelove programa, a koju nam nude virtualne funkcije, leži u osnovi onoga što se naziva *objektno orijentirani pristup*. Sve dok u programu ne počnemo koristiti virtualne funkcije (čime program pripremamo da se automatski “adaptira” na eventualna proširenja, što je znak da ispravno razmišljamo o budućnosti) ne možemo govoriti o objektno *orijentiranom* programu (čak i ukoliko u programu intenzivno koristimo klase i ostala načela objektno orijentirane filozofije), već samo o objektno *baziranom* programu (tj. programu zasnovanom na objektima).

Ako malo pažljivije razmislimo, sjetićemo se da smo se i ranije na jednom mjestu ipak susreli sa jednim mehanizmom koji omogućava da neka funkcija poziva drugu funkciju ne znajući o kojoj se funkciji radi sve do samog trenutka poziva. Taj mehanizam ostvaruju *pokazivači na funkcije*. Zaista, funkcija “*NulaFunkcije*” koju smo demonstrirali kada smo govorili o pokazivačima na funkcije, pozivala je “funkciju” nazvanu “*f*”, pri čemu se sve do trenutka poziva funkcije “*NulaFunkcije*” ne zna na koju se stvarnu funkciju oznaka funkcije “*f*” odnosi (stvarna funkcija koju predstavlja “*f*” zadaje se kao parametar funkcije “*NulaFunkcije*”). Tako, jednom napisana funkcija “*NulaFunkcije*” bez ikakve izmjene može pozivati svaku funkciju koja se napiše u budućnosti, pod uvjetom da joj se ona prenese kao parametar. Sličan slučaj imamo i sa funkcijama iz biblioteke “*algorithm*” koje primaju pokazivače na funkcije kao parametre. Na primjer, funkcija “*sort*” kao opcionalni treći parametar prima ime funkcije kriterija za koju pojma nema kako izgleda niti šta radi, i koja definitivno nije bila napisana u doba kada je napisana funkcija “*sort*” (funkciju kriterija piše programer koji koristi funkciju “*sort*”). Dakle, pokazivači na funkcije također na neki način nude mogućnost da stari dijelovi programa bez ikakve izmjene pozivaju novonapisane dijelove programa, samo što je upotreba virtuelnih funkcija jednostavnija. Ovo otkriće ne treba da nas pretjerano čudi. Naime, na kraju ovog poglavlja ćemo vidjeti da su virtuelne funkcije zapravo *prerušeni pokazivači na funkcije*, isto kao što su reference prerušeni pokazivači na objekte.

Izvjesni čitatelji i čitateljke će se vjerovatno zapitati zbog čega smo uopće definirali klasu “*Lik*” kao baznu klasu za klase “*Krug*”, “*Pravougaonik*” i “*Trougao*”. Naime, jedan zajednički atribut (naziv lika) i jedna metoda sa zajedničkom definicijom (metoda za ispis podataka sa likom) i nisu neki osobit razlog da izvedemo baš ovakvo nasljeđivanje, umjesto da klase “*Krug*”, “*Pravougaonik*” i “*Trougao*” prosto napišemo neovisno od ikakve zajedničke bazne klase, s obzirom da je ušteda koju smo ostvarili smještanjem ovih zajedničkih elemenata u klasu “*Lik*” neznatna u odnosu na situaciju koja bi nastala kada bismo ove elemente posebno definirali u svakoj od ovih klasa. Uštede bi mogle biti veće kada bi izvedeni objekti posjedovali veći broj zajedničkih elemenata koje bismo mogli smjestiti u baznu klasu. Međutim, postoji jedan mnogo jači razlog zbog čega smo se odlučili za ovakvu hijerarhijsku strukturu, a to je *mogućnost polimorfizma*. Naime, ukoliko definiramo neku promjenljivu koja

je tipa pokazivač na klasu “`Lik`”, takvoj promjenljivoj ćemo moći dodijeliti pokazivač na bilo koju od klasa koji su izvedeni iz klase “`Lik`” (tj. na bilo koji lik). Pored toga, kako su metode “`Obim`” i “`Povrsina`” virtualne, pozivi ovih metoda nad takvom promjenljivom proizvodiće različite efekte u zavisnosti od toga na koji lik promjenljiva pokazuje, odnosno imaćemo polimorfnu promjenljivu. Na primjer:

```
Lik *lik = new Pravougaonik(5, 4);
lik->Ispisi();
cout << "O = " << lik->Obim() << " P = " << lik->Povrsina() << endl;
delete lik;
lik = new Krug(3);
lik->Ispisi();
cout << "O = " << lik->Obim() << " P = " << lik->Povrsina() << endl;
```

U navedenom primjeru promjenljiva “`lik`” je iskorištena kao polimorfna promjenljiva. Da metode “`Obim`” i “`Povrsina`” nisu deklarirane kao virtualne, u ovom primjeru ne bi ispravno radio čak i njihov samostalni poziv nad promjenljivom “`lik`” (a ne samo posredni poziv iz metode “`Ispisi`”), jer bi zbog ranog povezivanja kompjajler zaključio da treba pozvati metode “`Obim`” i “`Povrsina`” iz klase “`Lik`” (s obzirom da je promjenljiva lik deklarirana kao pokazivač na “`Lik`”). Sličan efekat mogli bismo dobiti i ukoliko bismo imali neku funkciju koja kao formalni parametar ima referencu na objekat tipa “`Lik`”. Takvoj funkciji bismo mogli kao stvarne parametre prenosići objekte različitih tipova izvedenih iz klase “`Lik`”, pri čemu bi se uvijek pozivali odgovarajući metodi iz klase kojoj pripada stvarni argument. Dakle, ponovo bismo imali polimorfno ponašanje (odnosno, formalni parametar funkcije ponašao bi se kao polimorfna promjenljiva).

Možemo zaključiti da su virtualne metode *uvijek potrebne* kada želimo koristiti polimorfizam, ali s druge strane, prethodni primjer nas je uvjerio da one mogu biti potrebne i neovisno od polimorfizma (s obzirom da nam se potreba za virtualnim metodama pojavila i prije nego što smo se odlučili da koristimo polimorfizam). One su zapravo neophodne kad god neka od metoda treba da poziva druge metode, pri čemu se u trenutku pisanja klase ne zna na koje se tačno metode (tj. na koje metode iz mnoštva istoimenih metoda koje mogu biti definirane u klasama izvedenim iz te klase) ti pozivi odnose.

Zahvaljujući polimorfizmu, možemo deklarirati heterogeni niz likova, odnosno niz koji se ponaša kao da su mu elementi likovi koji mogu biti različitih tipova. Sljedeći primjer demonstrira jedan takav niz. Nakon deklaracije i inicijalizacije, nad svakim elementom ovog niza poziva se metoda “`Ispisi`”, koja proizvodi različite ispise u zavisnosti od toga koji lik sadrži koji element niza:

```
Lik *likovi[10] = {new Krug(3), new Pravougaonik(5, 2),
```

```

new Krug(5.2), new Kvadrat(4), new Trougao(3, 5, 6),
new Pravougaonik(7.3, 2.16), new PravilniTrougao(4.15),
new Krug(10), new PravougliTrougao(4, 2), new Kvadrat(3) };
for(int i = 0; i < 10; i++)
    likovi[i]->Ispisi();

```

Razumije se da “likovi” nije stvarno niz koji *sadrži različite objekte*, već niz čiji elementi *pokazuju na različite objekte*. Međutim, za nas je bitno da ovaj niz možemo koristiti *kao da se radi o nizu koji sadrži različite objekte*.

Da bismo u potpunosti iskoristili polimorfizam, sve metode bilo koje bazne klase koje mogu eventualno biti promijenjene u nekoj od izvedenih klasa trebale bi bez izuzetka biti deklarirane kao virtualne (u slučaju da *ne koristimo polimorfizam* ovo nije neophodno). Tako bi i metoda “Povrsina” u klasi “Trougao” (koja je izmijenjena u klasi “PravougliTrougao”) također trebala biti deklarirana kao virtualna. Da bismo se uvjerili u to, razmotrimo sljedeći primjer:

```

Trougao *t = new PravougliTrougao(5, 4);
cout << t -> Povrsina();

```

Jasno je da je gornja inicijalizacija legalna, jer pokazivač na klasu “Trougao” može pokazivati na objekat klase “PravougliTrougao” koji je izведен iz nje. Međutim, ukoliko metoda “Povrsina” nije deklarirana kao virtualna, njen poziv nad pokazivačem “t” pozvaće metodu “Povrsina” iz klase “Trougao” a ne iz klase “PravougliTrougao”, s obzirom da je “t” deklariran kao pokazivač na “Trougao” (rano povezivanje)!

Na ovom mjestu trebamo reći nekoliko riječi zbog čega smo rekli da nije sigurno da li klasu “Kvadrat” treba naslijediti iz klase “Pravougaonik”. Naime, bez obzira na činjenicu da kvadri jesu pravougaonici, sve operacije koje se mogu primijeniti na pravougaonike ne moraju se nužno moći primijeniti na kvadrate tako da oni ostanu kvadri. U navedenom primjeru, ne postoji operacija koje su podržane za pravougaonike a koje se ne bi smjele bezbjedno primijeniti na kvadrate, tako da je u ovom primjeru nasljeđivanje opravdano. Međutim, kvadrat se, na primjer, ne može izdužiti po širini uz očuvanje iste visine, tako da on ostane i dalje kvadrat (takva operacija će ga pretvoriti u pravougaonik koji *nije kvadrat*), dok će ista operacija primjenjena na pravougaonik i dalje kao rezultat dati pravougaonik. Stoga, ukoliko bi klasa “Pravougaonik” posjedovala i neku metodu koja omogućava takvu transformaciju, izvedba klase “Kvadrat” kao naslijedene klase iz klase “Pravougaonik” ne bi bila dobro rješenje, s obzirom da bi se takva metoda mogla primijeniti i na objekte tipa “Kvadrat”, što bi neizbjegno dovelo do promjene njihove prirode (oni bi prestali biti kvadri). Slična razmatranja vrijede za odnose između klase “Trougao” i iz nje naslijedenih klasa. O ovakvim detaljima treba dobro razmišljati prilikom donošenja odluke o hijerarhijskom dizajnu klasa i odnosa između njih.

Primijetimo da je u jeziku C++ polimorfizam ograničen samo na tipove koji su izvedeni iz *iste bazne klase*. Stoga se polimorfne promjenljive mogu ponašati kao promjenljive različitih tipova, ali samo manje ili više *srodnih tipova*. Na primjer, promjenljiva “lik” mogla je sadržavati (preciznije, pokazivati na) *različite vrste likova*, ali je mogla sadržavati *samo likove*, a ne i npr. nešto drugo (recimo studente). Slično, heterogeni nizovi se ponašaju kao nizovi koji sadrže elemente raličitih, ali *međusobno srodnih* tipova. Upravo zbog toga nam je i bila potrebna klasa “Lik”. Da sve klase koje opisuju različite likove nisu kao svog zajedničkog pretka imale zajedničku klasu “Lik”, polimorfizam ne bi bio moguć. Ovakve bazne klase koje služe samo da bi druge klase bile izvedene iz njih u cilju omogućavanja polimorfizma nazivaju se *apstraktne bazne klase*. Drugim riječima, one definiraju samo “kostur” odnosno opće karakteristike neke familije objekata, dok će specifičnosti svakog pripadnika te familije biti definirane u klasama koje su iz nje izvedene.

U normalnim okolnostima, nikada se ne deklariraju primjeri apstraktne bazne klase, niti se primjeri bazne klase kreiraju dinamički pomoću operatorka “**new**”. Međutim, do sada nas niko ne sprečava da uradimo tako nešto, npr. da izvršimo sljedeće deklaracije:

```
Lik l;  
Lik *pok_l = new Lik;
```

Jasno je da su ovakve deklaracije posve beskorisne, jer se sa primjercima klase “Lik” ne može uraditi ništa korisno. Stoga postoji način da se ovakve deklaracije *formalno zbrane*. Za tu svrhu dovoljno je u klasi “Lik” neke od virtualnih metoda (najbolje sve) *ostaviti neimplementirane* (umjesto da im dajemo neku manje ili više besmislenu implementaciju). Da bismo to uradili, umjesto tijela metode trebamo prosto napisati oznaku “= 0”, kao u sljedećoj deklaraciji klase “Lik”:

```
class Lik {  
public:  
    virtual double Obim() const = 0;  
    virtual double Povrsina() const = 0;  
    void Ispis() const {  
        cout << "Lik: " << naziv << endl << "Obim: " << Obim() << endl  
            << " Površina: " << Povrsina() << endl;  
    }  
};
```

Ovom oznakom govorimo da metoda neće biti implementirana niti unutar klase, niti izvan klase, nego tek eventualno u nekoj od klasa koje nasljeđuju klasu “Lik”. Samo virtualne metode mogu ostati neimplementirane, s obzirom da se one pozivaju mehanizmom kasnog povezivanja (tako da mogu biti implementirane i naknadno). Također, ostavljanje virtualnih metoda neimplementiranim ima smisla samo u apstraktnim baznim klasama (koje će svakako biti naslijedene). Stoga se često uvodi i *formalna definicija apstraktne bazne klase* kao klase

koja posjeduje barem jednu neimplementiranu virtualnu funkciju, ili kako se to još često kaže, *čisto virtualnu funkciju* (engl. *pure virtual function*) Primijetimo da oznaka “`= 0`” umjesto tijela metode nije ekvivalentna pisanju praznog tijela funkcije, tj. praznog para vitičastih zagrada “`{ }`”. Praznim tijelom funkcije, mi funkciju ipak *implementiramo*, bez obzira što njena implementacija *ne radi ništa*. Ukoliko smo definirali apstraktnu baznu klasu (tj. klasu sa čisto virtualnim funkcijama) kompjuler nam neće dozvoliti da deklariramo niti jedan primjerak takve klase, niti da pozivom operatora “`new`” dinamički kreiramo primjerak takve klase. S druge strane, biće dozvoljeno da deklariramo *pokazivač na takvu klasu* (jer jedino tako možemo ostvariti polimorfizam), ali će on uvijek pokazivati isključivo na objekte neke konkretnе klase naslijedene iz nje!

Konstruktori se ne mogu deklarirati kao virtualni, ali destruktori mogu. Zapravo, kada se god koristi polimorfizam, a bazna klasa i naslijedene klase posjeduju destruktore koji se međusobno razlikuju, destruktor bazne klase *obavezno mora biti deklariran kao virtualan*. Naime, u suprotnom bi u situaciji kada pokazivač na baznu klasu pokazuje na objekat izvedene klase, prilikom uništavanja objekta bio pozvan pogrešan destruktor (destruktor bazne klase, umjesto destruktora izvedene klase)!

Za kraj ostaje još da objasnimo kako zapravo radi mehanizam pozivanja virtualnih funkcija. Kao što smo već rekli, virtualne funkcije su zapravo preruseni pokazivači na funkcije, što i ne treba da čudi, jer u osnovi svake indirekcije (a kod virtualnih funkcija se očigledno radi o indirektnim pozivima) leže pokazivači. Naime, svakoj virtualnoj funkciji pridružuje se jedan pokazivač na funkciju, koji se u konstruktoru klase inicijalizira da pokazuje upravo na tu funkciju. U svakoj od naslijedjenih klasa koje modifiraju tu funkciju, konstruktor inicijalizira taj pokazivač da pokazuje na modificiranu verziju te funkcije. Virtualna funkcija se nikada ne poziva direktno, nego isključivo preko njoj pridruženog pokazivača, koji je prethodno inicijaliziran da pokazuje na “pravu” verziju funkcije, tj. onu verziju koju zaista u tom trenutku treba pozvati. Stoga se ponašanje virtualnih funkcija može simulirati korištenjem pokazivača na funkcije (razumije se da to u praksi ne treba raditi, jer je mnogo jednostavnije koristiti virtualne funkcije, kad već postoje). Slijedi jedan vještački konstruisan primjer koji ostvaruje isto polimorfno ponašanje klasa “`Student`” i “`DiplomiraniStudent`” kakvo smo već ranije imali pomoću korištenja virtualnih funkcija, ali ovaj put bez njihovog korištenja:

```
class Student {
protected:
    string ime[50];
    int indeks;
    void (*pok_na_fn_za_ispis)(Student *s);
public:
    Student(string ime[], int ind) : ime(ime), indeks(ind) {
        pok_na_fn_za_ispis = IspisiStudenta;
    }
    string VratiIme() const { return ime; }
    int VratiIndeks() const { return indeks; }
    void Ispisi() const { pok_na_fn_za_ispis(this); }
    friend void IspisiStudenta(Student *s);
};
```

```

class DiplomiraniStudent : public Student {
    int godina_diplomiranja;
public:
    DiplomiraniStudent(string ime, int ind, int god_dipl)
        : student(ime, ind), godina_diplomiranja(god_dipl) {
            pok_na_fn_za_ispis = IspisiDiplomiranogStudenta;
    }
    int VratiGodinuDiplomiranja() const { return godina_diplomiranja; }
    friend void IspisiDiplomiranogStudenta(Student *s);
};

void IspisiStudenta(Student *s) {
    cout << "Student " << s->ime << " ima indeks " << s->indeks;
}

void IspisiDiplomiranogStudenta(Student *s) {
    IspisiStudenta(s);
    cout << ", a diplomirao je "
        << ((DiplomiraniStudent *)s)->godina_diplomiranja << ". godine";
}

```

Ovaj primjer je funkcionalno potpuno ekvivalentan primjeru koji koristi virtualne funkcije, samo se u ovom primjeru one simuliraju pomoću pokazivača na funkcije. Zapravo, kad koristimo virtualne funkcije, računar radi praktično iste stvari kao u ovom primjeru, samo što zavrzlame oko upotrebe pokazivača na funkcije i njihove pravilne inicijalizacije umjesto nas obavlja sam kompjajler, čime nas pošteđuje mnogih muka. Razumjevanje izloženog primjer zahjeva prilično dobro poznavanje velikog broja dosada izloženih koncepata, tako da ukoliko u potpunosti razumijete ovaj primjer, to je dobar znak da ste vrlo uspješno usvojili do sada izložene koncepte.

36. Rad sa datotekama

Svi programi koje smo do sada pisali posjedovali su suštinski nedostatak da se svi uneseni podaci gube onog trenutka kada se program završi. Ovaj problem rješava se uvođenjem *datoteka* (engl. *files*). Datoteke predstavljaju strukturu podataka koja omogućava *trajno smještanje informacija* na nekom od uređaja vanjske memorije (obično na hard disku). Sve primjene računara u kojima podaci *moraju biti sačuvani* između dva pokretanja programa *obavezno zahtijevaju upotrebu datoteka*. Na primjer, bilo koji program za obradu teksta mora posjedovati mogućnost da se dokument sa kojim radimo sačuva na disk, tako da u budućnosti u bilo kojem trenutku možemo ponovo pokrenuti program, otvoriti taj dokument (koji je bio sačuvan kao datoteka), i nastaviti raditi sa njim. Slično, bilo kakav ozbiljan program za vođenje evidencije o studentima mora posjedovati mogućnost trajnog čuvanja svih unesenih podataka o studentima, inače bi bio potpuno neupotrebljiv (jer bismo pri svakom pokretanju programa morali ponovo unositi već unesene informacije).

U jeziku C++ podržana su dva načina za rad sa datotekama. Prvi način naslijeden je iz jezika C, i u njemu se rad sa datotekama zasniva na primjeni izvjesnih funkcija čija imena uglavnom počinju slovom “f” (npr. “`fopen`”, “`fclose`”, “`fget`”, “`fput`”, “ `fread`”, “`fwrite`”, “`fseek`”, “`fprintf`”, “`fscanf`”, itd.), i koje kao jedan od parametara obavezno zahtijevaju pokazivač na strukturu nazvanu “`FILE`”, koja predstavlja tzv. *deskriptor datoteke*. Na ovom načinu rada sa datotekama se nećemo zadržavati, s obzirom da jezik C++ podržava i mnogo jednostavniji i fleksibilniji način rada sa datotekama zasnovan na ulaznim i izlaznim tokovima, koji je veoma sličan načinu korištenja objekata ulaznog i izlaznog toka “`cin`” i “`cout`”, Zbog toga ćemo u nastavku razmatrati samo takav način rada sa datotekama.

Već smo rekli da su “`cin`” i “`cout`” zapravo instance klase sa imenima “`istream`” i “`ostream`” koje su definirane u biblioteci “`iostream`”, a koje predstavljaju objekte ulaznog i izlaznog toka povezane sa standardnim ulaznim odnosno izlaznim uređajem (tipično tastaturom i ekranom). Da bismo ostvarili rad sa datotekama, moramo sami definirati svoje instance klase nazvanih “`ifstream`” i “`ofstream`”, koje su definirane u biblioteci “`fstream`”, a koje predstavljaju ulazne odnosno izlazne tokove povezane sa datotekama. Obje klase posjeduju konstruktor sa jednim stringovnim parametrom (preciznije, parametrom tipa znakovnog niza) koji predstavlja ime datoteke sa kojom se tok povezuje. Pored toga, klasa “`ifstream`” je naslijedena iz klase “`istream`”, a klasa “`ofstream`” iz klase “`ostream`”, tako da instance klase “`ifstream`” i “`ofstream`” posjeduju sve metode i operatore koje posjeduju instance klase “`istream`” i “`ostream`”, odnosno objekti “`cin`” i “`cout`”. To uključuje i operatore “`<<`” odnosno “`>>`”, zatim manipulatore poput “`setw`”, itd. Tako se upis u datoteku vrši na isti način kao ispis na ekran, samo umjesto objekta “`cout`” koristimo vlastitu instancu klase “`ofstream`”. Pri tome će kreirana datoteka imati istu logičku strukturu kao i odgovarajući ispis na ekran. Na primjer, sljedeći primjer kreiraće datoteku pod nazivom “`BROJEVI.TXT`” i u nju upisati spisak prvih 100 prirodnih brojeva i njihovih kvadrata. Pri tome će broj i njegov kvadrat biti razdvojeni zarezom, a svaki par broj–kvadrat biće smješten u novom redu:

```

#include <fstream>

using namespace std;

int main() {
    ofstream izlaz("BROJEVI.TXT");
    for(int i = 1; i <= 100; i++)
        izlaz << i << "," << i * i << endl;
    return 0;
}

```

Nakon pokretanja ovog programa, ukoliko je sve u redu, u direktoriju (folderu) u kojem se nalazi sam program biće kreirana nova datoteka pod imenom "BROJEVI.TXT". Njen sadržaj se može pregledati pomoću bilo kojeg tekstualnog editora (npr. NotePad-a ukoliko radite pod nekim od Windows serije operativnih sistema). Tako, ukoliko pomoću nekog tekstualnog editora otvorite sadržaj novokreirane datoteke "BROJEVI.TXT", njen sadržaj će izgledati isto kao da su odgovarajući brojevi ispisani na ekran korištenjem objekta izlaznog toka "cout". Drugim riječima, njen sadržaj bi trebao izgledati tačno ovako (ovdje su upotrijebljene tri tačke da ne prikazujemo čitav sadržaj):

```

1,1
2,4
3,9
4,16
5,25
...
99,9801
100,10000

```

Ime datoteke može biti bilo koje ime koje je u skladu sa konvencijama operativnog sistema na kojem se program izvršava. Tako, na primjer, pod MS-DOS operativnim sistemom, ime datoteke ne smije biti duže od 8 znakova i ne smije sadržavati razmake, dok pod Windows serijom operativnih sistema imena datoteka mogu sadržavati razmake i mogu biti dugačka do 255 znakova. Ni pod jednim od ova dva operativna sistema imena datotema ne smiju sadržavati neki od znakova "\", "/", ":" , "*" , "?" , "<" , ">" , "|" , kao ni znak navoda. Također, izrazito je nepreporučljivo korištenje znakova izvan engleskog alfabetu u imenima datoteke (npr. naših slova), jer takva datoteka može postati nečitljiva na računaru na kojem nije instalirana podrška za odgovarajući skup slova. Slične konvencije vrijede i pod UNIX odnosno Linux operativnim sistemima, samo je na njima skup zabranjenih znakova nešto širi. Najbolje se držati kratkih imena sastavljenih samo od slova engleskog alfabetu i eventualno

cifara. Takva imena su legalna praktično pod svakim operativnim sistemom.

Imena datoteka tipično sadrže tačku, iza koje slijedi nastavak (ekstenzija) koja se obično sastoji od tri slova. Korisnik može zadati ekstenziju kakvu god želi (pod uvjetom da se sastoji od legalnih znakova), ali treba voditi računa da većina operativnih sistema koristi ekstenziju da utvrdi šta predstavlja sadržaj datoteke, i koji program treba automatski pokrenuti da bi se prikazao sadržaj datoteke ukoliko joj probamo neposredno pristupiti izvan programa, npr. duplim klikom miša na njenu ikonu pod Windows operativnim sistemima (ovo je tzv. *asocijativno pridruživanje* bazirano na ekstenziji). U operativnim sistemima sa grafičkim okruženjem, ikona pridružena datoteci također može zavisiti od njene ekstenzije. Stoga je najbolje datotekama koje sadrže tekstualne podatke davati ekstenziju “.TXT”, čime označavamo da se radi o tekstualnim dokumentima. Windows operativni sistemi ovakvim datotekama automatski dodjeljuju ikonu tekstualnog dokumenta, a prikazuju ih pozivom programa NotePad. Ukoliko bismo kreiranoj datoteci dali na primjer ekstenziju “.BMP”, Windows bi pomislio da se radi o slikovnom dokumentu, dodijelio bi joj ikonu slikovnog dokumenta, i pokušao bi da je otvori pomoću programa Paint, što sigurno ne bi dovelo do smislenog ponašanja. Moguće je ekstenziju izostaviti u potpunosti. U tom slučaju, Windows dodjeljuje datoteci ikonu koja označava dokument nepoznatog sadržaja (ista stvar se dešava ukoliko se datoteci dodijeli ekstenzija koju operativni sistem nema registriranu u popisu poznatih ekstenzija). Pokušaj pristupa takvoj datoteci izvan programa pod Windows operativnim sistemima dovodi do prikaza dijaloga u kojem nas operativni sistem pita koji program treba koristiti za pristup sadržaju datoteke.

Ime datoteke može sadržavati i lokaciju (tj. specifikaciju uredaja i foldera) gdje želimo da kreiramo datoteku. Ova lokacija se zadaje u skladu sa konvencijama konkretnog operativnog sistema pod kojim se program izvršava. Windows serija operativnih sistema naslijedila je ove konvencije iz MS–DOS operativnog sistema. Tako, ukoliko želimo da kreiramo datoteku “**BROJEVI.TXT**” u folderu “**RADNI**” na flopi disku, pod MS–DOS ili Windows operativnim sistemima to možemo uraditi ovako:

```
ofstream izlaz("A:\\RADNI\\BROJEVI.TXT");
```

Podsetimo se da po ovoj konvenciji “**A:**” označava flopi disk, dok znak “****” razdvaja imena foldera u putanji do željene datoteke. Znak “****” je *uduplan* zbog toga što u jeziku C++ znak “****” koji se pojavi između znakova navoda označava da iza njega može slijediti znak koji označava neku specijalnu akciju (poput oznake “**\n**” za novi red), tako da ukoliko želimo da između navodnika imamo bukvalno znak “****”, moramo ga pisati udvojeno. Na ovu činjenicu se često zaboravlja kada se u specifikaciji imena datoteke treba da pojave putanje. Usput, iz ovog primjera je jasno zbog čega sama imena datoteka ne smiju sadržavati znakove poput “**:**” ili “****”. Njima je očito dodijeljena specijalna uloga.

Kako je formalni parametar konstruktora klase “**ofstream**” stringovnog tipa, možemo kao

stvarni parametar navesti bilo koji niz znakova, a ne samo stringovnu konstantu. Na primjer, sljedeća konstrukcija je sasvim legalna, i prikazuje kako raditi sa datotekom čije ime nije unaprijed poznato:

```
char ime[100];  
cout << "Unesite ime datoteke koju želite kreirati:";  
cin.getline(ime, sizeof ime);  
ofstream izlaz(ime);
```

Ipak, treba napomenuti da konstruktor klase “ofstream” neće kao parametar prihvati dinamički string, odnosno objekat tipa “string”. Međutim, ranije smo govorili da se objekti tipa “string” uvijek mogu pretvoriti u klasične nul-terminalizirane stringove izvršavanjem operacije (zapravo metode) “c_str”. Tako, ukoliko želimo koristiti tip “string”, možemo prethodni primjer napisati ovako:

```
string ime;  
cout << "Unesite ime datoteke koju želite kreirati:";  
getline(cin, ime);  
ofstream izlaz(ime.c_str());
```

Može se desiti da upisivanje podataka u datoteku zbog nekog razloga ne uspije (npr. ukoliko se disk popuni). U tom slučaju, izlazni tok dospjeva u neispravno stanje, i operator “!” primijenjen na njega daje kao rezultat jedinicu. Može se desiti da ni samo kreiranje datoteke ne uspije (mogući razlozi su disk popunjen do kraja na samom početku, disk zaštićen od upisa, neispravno ime datoteke, nepostojeća lokacija, itd.). U tom slučaju, izlazni tok je u neispravnom stanju odmah po kreiranju. Sljedeći primjer pokazuje kako možemo preuzeti kontrolu nad svim nepredviđenim situacijama prilikom kreiranja izlaznog toka vezanog sa datotekom:

```
ofstream izlaz("BROJEVI.TXT");  
if(!izlaz)  
    cout << "Kreiranje datoteke nije uspjelo!\n";  
else {  
    for(int i = 1; i <= 100; i++) {  
        izlaz << i << "," << i * i << endl;  
        if(!izlaz) {  
            cout << "Nešto nije u redu sa upisom u datoteku!\n";  
            break;  
        }  
    }
```

```
    }
}
```

Nakon što smo pokazali kako se može kreirati datoteka, potrebno je pokazati kako se već kreirana datoteke može *pročitati*. Za tu svrhu je potrebno kreirati objekat ulaznog toka povezan sa datotekom, odnosno instancu klase “*ifstream*”. Pri tome, datoteka sa kojom vežemo tok *mora postojati*, inače objekat ulaznog toka dolazi u neispravno stanje odmah po kreiranju. U slučaju uspješnog kreiranja, čitanje iz datoteke se obavlja na isti način kao i čitanje sa tastature. Pri tome je korisno zamisliti da se svakoj datoteci pridružuje neka vrsta pokazivača odnosno *kurzora* koji označava mjesto odakle se vrši čitanje. Nakon kreiranja ulaznog toka, kurzor se nalazi na početku datoteke, a nakon svakog čitanja, kurzor se pomjera iza pročitanog podatka, tako da naredno čitanje čita sljedeći podatak, s obzirom da se podaci uvijek čitaju od mjesta na kojem se nalazi kurzor. Tako, datoteku “*BROJEVI.TXT*” koju smo kreirali prethodnim programom možemo isčitati i ispisati na ekran pomoću sljedećeg programa:

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    ifstream ulaz("BROJEVI.TXT");
    if(!ulaz)
        cout << "Otvaranje datoteke nije uspjelo!\n";
    else {
        int broj_1, broj_2;
        char znak;
        for(int i = 1; i <= 100; i++) {
            ulaz >> broj_1 >> znak >> broj_2;
            cout << broj_1 << " " << broj_2 << endl;
        }
    }
    return 0;
}
```

U ovom programu smo namjerno prilikom ispisa na ekranu brojeve razdvajali razmakom, bez obzira što su u datoteci razdvojeni zarezom. Svrha je da se pokaže kako se vrši čitanje

podataka iz datoteke, dok sa pročitanim podacima program može raditi šta god želi.

Nedostatak prethodnog programa je u tome što se isčitavanje podataka vrši “**for**” petljom, pri čemu unaprijed moramo znati koliko podataka sadrži datoteka. U praksi se obično javlja potreba za čitanjem datoteke za koje *ne znamo* koliko elemenata sadrže. U tom slučaju se čitanje vrši u petlji koju prekidamo nakon što se dostigne kraj datoteke. Dostizanje kraja testiramo na taj način što pokušaj čitanja nakon što je dostignut kraj datoteke dovodi ulazni tok u neispravno stanje, što možemo testirati primjenom operatora “!”. Sljedeći primjer radi isto kao i prethodni, ali ne prepostavlja prethodno poznavanje broja elemenata u datoteci (radi kratkoće, pisaćemo samo sadržaj tijela “**main**” funkcije):

```
ifstream Ulaz("BROJEVI.TXT");
if(!ulaz)
    cout << "Otvaranje datoteke nije uspjelo!\n";
else {
    int broj_1, broj_2;
    char znak;
    for(;;) {
        ulaz >> broj_1 >> znak >> broj_2;
        if(!ulaz) break;
        cout << broj_1 << " " << broj_2 << endl;
    }
}
```

Zbog poznate činjenice da operator “>>” primijenjen na neki ulazni tok vraća kao rezultat referencu na taj ulazni tok, kao i činjenice da se tok upotrijebljen kao uvjet (npr unutar “**if**” naredbe) ponaša kao logička neistina u slučaju kada je neispravnom stanju, prethodni primjer možemo kraće napisati i ovako:

```
ifstream ulaz("BROJEVI.TXT");
if(!ulaz)
    cout << "Otvaranje datoteke nije uspjelo!\n";
else {
    int broj_1, broj_2;
    char znak;
    while(ulaz >> broj_1 >> znak >> broj_2)
        cout << broj_1 << " " << broj_2 << endl;
}
```

Nedostatak navedenih rješenja je u tome što ulazni tok može dospjeti u neispravno stanje ne samo zbog pokušaja čitanja nakon kraja datoteke, nego i iz drugih razloga, kao što je npr. nailazak na nenumeričke znakove prilikom čitanja brojčanih podataka, ili nailazak na fizički oštećeni dio diska. Stoga su uvedene metode “eof”, “bad” i “fail” (bez parametara), pomoći kojih možemo saznati razloge dolaska toka u neispravno stanje. Metoda “eof” vraća logičku istinu (tj. logičku vrijednost “**true**”) ukoliko je tok dospio u neispravno stanje zbog pokušaja čitanja nakon kraja datoteke, a u suprotnom vraća logičku neistjinu (tj. logičku vrijednost “**false**”). Metoda “bad” vraća logičku istinu ukoliko je razlog dospijevanja toka u neispravno stanje neko fizičko oštećenje, ili zabranjena operacija nad tokom. Metoda “fail” vraća logičku istinu u istim slučajevima kao i metoda “bad”, ali uključuje i slučajeve kada je do dospijevanja toka u neispravno stanje došlo uslijed nailaska na neočekivane podatke prilikom čitanja (npr. na nenumeričke znakove prilikom čitanja broja). Na ovaj način možemo saznati da li je do prekida čitanja došlo prosto uslijed dostizanja kraja datoteke, ili je u pitanju neka druga greška. Ovo demonstrira sljedeći programski isječak u kojem se vrši čitanje datoteke sve do dostizanja njenog kraja, ili nailaska na neki problem. Po završetku čitanja ispisuje se razlog zbog čega je čitanje prekinuto:

```
ifstream ulaz("BROJEVI.TXT");
if(!ulaz)
    cout << "Otvaranje datoteke nije uspjelo!\n";
else {
    int broj_1, broj_2;
    char znak;
    while(ulaz >> broj_1 >> znak >> broj_2)
        cout << broj_1 << " " << broj_2 << endl;
}
if(ulaz.eof()) cout << "Nema više podataka!\n";
else if(ulaz.bad()) cout << "Datoteka je vjerovatno oštećena!\n";
else cout << "Datoteka sadrži neočekivane podatke!\n";
```

Treba napomenuti da nakon što tok (bilo ulazni, bilo izlazni) dospije u neispravno stanje, iz bilo kakvog razloga (npr. zbog pokušaja čitanja iza kraja datoteke), sve dalje operacije nad tokom se ignoriraju sve dok korisnik ne vrati tok u ispravno stanje pozivom metode “clear” nad objektom toka.

Veoma je važno je napomenuti da se svi podaci u tekstualnim datotekama čuvaju isključivo kao *slijed znakova*, bez obzira na stvarnu vrstu upisanih podataka. Tako se, prilikom upisa u tekstualne datoteke, u njih upisuje tačno onaj niz znakova koji bi se pojavio na ekranu prilikom ispisa istih podataka. Slično, prilikom čitanja podataka iz tekstualne datoteke, računar će se ponašati isto kao da je niz znakova od kojih se datoteka sastoji unesen putem tastature.

Stoga je moguće, uz izvjesnu dozu opreza, u tekstualnu datoteku upisati podatak jednog tipa (npr. cijeli broj), a zatim ga iščitati iz iste datoteke kao podatak drugog tipa (npr. kao niz znakovnih promjenljivih). Drugim riječima, tekstualne datoteke nemaju precizno utvrđenu strukturu, već je njihova struktura, kao i način interpretacije već kreiranih tekstualnih datoteka, isključivo pod kontrolom programa koji ih obrađuje. Ova činjenica omogućava veliku fleksibilnost pri radu sa tekstualnim datotekama, ali predstavlja i čest uzrok grešaka, pogotovo ukoliko se njihov sadržaj ne interpretira na pravi način prilikom čitanja. Na primjer, ukoliko u tekstualnu datoteku upišemo zaredom prvo broj 2, a zatim broj 5, u istom redu i bez ikakvog razmaka između njih, prilikom čitanja će isti podaci biti interpretirani kao jedan broj – broj 25!

Ulagni i izlazni tokovi se mogu *zatvoriti* pozivom metode “`close`”, nakon čega tok prestaje biti povezan sa nekom konkretnom datotekom. Sve dalje operacije sa tokom su nedozvoljene sve dok se tok ponovo ne otvori (tj. poveže sa konkretnom datotekom) pozivom metode “`open`”, koja prima iste parametre kao i konstruktor objekata “`istream`” odnosno “`ostream`” (i pri tome obavlja iste akcije kao i navedeni konstruktori). Ovo omogućava da se prekine veza toka sa nekom datotekom i preusmjeri na drugu datoteku, tako da se ista promjenljiva izlaznog toka može koristiti za pristup različitim datotekama, kao u sljedećem programskom isječku:

```
ofstream Izlaz("PRVA.TXT");
...
Izlaz.close();
Izlaz.open("DRUGA.TXT");
...
```

Tok se može nakon zatvaranja ponovo povezati sa datotekom na koju je prethodno bio povezan. Međutim, treba napomenuti da svako otvaranje ulaznog toka postavlja kurzor za čitanje *na početak datoteke*, tako da nakon svakog otvaranja čitanje datoteke kreće od njenog početka (ukoliko sami ne pomjerimo kurzor pozivom metode “`seekg`”, o čemu će kasnije biti govor). Ovo je ujedno jedan od načina kako započeti čitanje datoteke ispočetka. Klase “`ifstream`” i “`ofstream`” također posjeduju i konstruktoare bez parametara, koje kreiraju ulazni odnosno izlazni tok koji nije vezan niti na jednu konkretnu datoteku, tako da je deklaracija poput

```
ifstream ulaz;
```

sasvim legalna. Ovako definiran tok može se koristiti samo nakon što se eksplicitno otvori (i poveže sa konkretnom datotekom) pozivom metode “`open`”. Također, treba napomenuti da klase “`ifstream`” i “`ofstream`” sadrže destruktur koji (između ostalog) poziva metodu “`close`”, tako da se tokovi automatski zatvaraju kada odgovarajući objekat koji predstavlja tok prestane postojati. Drugim riječima, nije potrebno eksplicitno zatvarati tok, kao što je neophodno u nekim drugim programskim jezicima.

Već smo vidjeli da prilikom kreiranja objekta izlaznog toka datoteka sa kojom se vezuje izlazni tok ne mora od ranije postojati, već da će odgovarajuća datoteka automatski biti kreirana. Međutim, ukoliko datoteka sa navedenim imenom *već postoji*, ona će biti *uništena*, a umjesto nje će biti kreirana nova prazna datoteka sa istim imenom. Isto vrijedi i za otvaranje izlaznog toka pozivom metode “*open*”. Ovakvo ponašanje je često poželjno, međutim u nekim situacijama to nije ono što želimo. Naime, često se javlja potreba da *dopišemo* nešto na kraj već postojeće datoteke. Za tu svrhu, konstruktori klase “*ifstream*” i “*ofstream*”, kao i metoda “*open*” posjeduju i drugi parametar, koji daje dodatne specifikacije kako treba rukovati sa datotekom. U slučaju da želimo da podatke upisujemo u *već postojeću datoteku*, tada kao drugi parametar konstruktoru odnosno metodi *open* trebamo proslijediti vrijednost koja se dobija kao *binarna disjunkcija* pobrojanih konstanti “*ios::out*” i “*ios::app*”, definiranih unutar klase “*ios*” (binarna disjunkcija dobija se primjenom operatora “|”). Konstanta “*ios::out*” označava da želimo *upis*, a konstanta “*ios::app*” da želimo *nadovezivanje* na već postojeću datoteku (od engl. *append*). Tako, ukoliko želimo da proširimo postojeću datoteku “*BROJEVI.TXT*” brojem 101 i njegovim kvadratom, izvršićemo naredbe

```
ofstream izlaz("BROJEVI.TXT", ios::out | ios::app);
izlaz << 101 << "," << 101 * 101 << endl;
```

I u ovom slučaju, ukoliko datoteka “*BROJEVI.TXT*” ne postoji, prosto će biti kreirana nova.

Interesantan je i sljedeći primjer, koji poziva metodu “*get*” da ispiše sadržaj proizvoljne tekstualne datoteke na ekran, znak po znak:

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    char ime[100], znak;
    cout << "Unesi ime datoteke koju želite prikazati: ";
    cin >> ime;
    ifstream ulaz(ime);
    while((znak = ulaz.get()) != EOF)
        cout << znak;
    return 0;
}
```

Iz ovog primjera možemo naučiti još jednu interesantnu osobinu: metoda “*get*” vraća kao rezultat konstantu “*EOF*” (čija je vrijednost -1) u slučaju da ulazni tok dospije u neispravno stanje (npr. zbog pokušaja čitanja iza kraja datoteke). Ovu osobinu smo iskoristili da skratimo program.

Bitno je naglasiti da se i sam izvorni program pisan u jeziku C++ također na disku čuva kao obična tekstualna datoteka. Tako, na primjer, možemo prethodni program snimiti na disk pod

imenom "PRIKAZ.CPP", a zatim ga pokrenuti i kao ime datoteke koju želimo prikazati unijeti njegovo vlastito ime "PRIKAZ.CPP". Kao posljedicu, program će na ekran izlistati samog sebe.

Pored klase "ifstream" i "ofstream", postoji i klasa "fstream", koja djeluje kao kombinacija klasa "ifstream" i "ofstream". Objekti klase "fstream" mogu se po potrebi koristiti kao ulazni, ili kao izlazni tokovi. Zbog toga, konstruktor klase "fstream", kao i metoda "open", obavezno zahtijevaju drugi parametar kojim se specificira da li tok otvaramo kao ulazni tok ili izlazni tok (ovu specifikaciju navodimo tako što kao parametar zadajemo vrijednost konstante "ios::in" odnosno "ios::out"). Tako možemo istu promjenljivu zavisno od potrebe koristiti kao ulazni ili kao izlazni tok. Na primjer, sljedeći program traži od korisnika da unosi sa tastature slijed brojeva, koji se zatim upisuju u datoteku "RAZLICITI.TXT", ali samo ukoliko uneseni broj već ranije nije unijet u datoteku (tako da će na kraju datoteka sadržavati samo različite brojeve):

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    fstream datoteka ("RAZLICITI.TXT", ios::in);

    if (!datoteka) {

        datoteka.clear();

        datoteka.open ("RAZLICITI.TXT", ios::out);

    }

    datoteka.close();
    for (;;) {
        int broj, tekuci;
        cout << "Unesite broj (0 za izlaz): ";
        cin >> broj;
        if (broj == 0) break;
        datoteka.open ("RAZLICITI.TXT", ios::in);
        while (datoteka >> tekuci)
            if (tekuci == broj) break;
        if (datoteka.eof ()) {
            datoteka.clear();
            datoteka.close();
            datoteka.open ("RAZLICITI.TXT", ios::out | ios::app);
            datoteka << broj << endl;
        }
        datoteka.close();
    }
    return 0;
}
```

U ovom primjeru, tok se prvo otvara za čitanje. U slučaju da odgovarajuća datoteka ne postoji, tok dospijeva u neispravno stanje. U tom slučaju, nakon oporavljanja toka, tok se otvara za pisanje, što dovodi do kreiranja nove prazne datoteke. Na kraju se u svakom slučaju tok

zatvara prije ulaska u glavnu petlju. Unutar glavne petlje, nakon što korisnik unese broj, tok se otvara za čitanje, i datoteka se iščitava da se utvrdi da li ona već sadrži traženi broj. Ukoliko se traženi broj pronađe, traganje se prekida, a novounesen broj se ne upisuje. U slučaju da traženi broj nije nađen, prilikom traganja će biti dostignut kraj datoteke, i tok će doći u neispravno stanje. U tom slučaju, tok se prvo oporavlja pozivom metode “`clear`”, zatim se zatvara, i otvara za upisivanje (podsjetimo se da oznaka “`ios::app`” govori da datoteku ne treba brisati, već da upis treba izvršiti na njen kraj), a novounesen broj se upisuje u datoteku. Primijetimo da se tok svaki put unutar petlje iznova otvara i zatvara, što osigurava da će se u svakom prolazu kroz petlju čitanje datoteke vršiti od njenog početka. Također, bitno je napomenuti da se metoda “`open`” ne smije primijeniti na tok koji je već otvoren, nego ga je uvijek prethodno potrebno zatvoriti prije nego što ga eventualno ponovo otvorimo. Primjena metode “`open`” na otvoren tok doveće tok u neispravno stanje.

Ono što čini objekte klase “`fstream`” posebno interesantnim je mogućnost da se oni mogu koristiti istovremeno i kao ulazni i izlazni tokovi. Za tu svrhu kao drugi parametar treba zadati binarnu disjunkciju konstanti “`ios::in`” i “`ios::out`”. Ova mogućnost se rijetko koristi pri radu sa tekstualnim datotekama, pa ćemo o njoj govoriti kada budemo govorili o binarnim datotekama.

Nema nikakvog razloga da u istom programu ne možemo imati više objekata ulaznih ili izlaznih tokova vezanih na datoteke. Pri tome je obično više različitih tokova vezano za različite datoteke, ali sasvim je legalno imati i više tokova vezanih na *istu datoteku*. Više ulaznih odnosno izlaznih tokova se obično koristi u slučajevima kada program treba da istovremeno obrađuje više datoteka. Na primjer, pretpostavimo da imamo program koji treba da obrađuje podatke o studentima, koji su pohranjeni u dvije datoteke: “`STUDENTI.TXT`”, koja čuva podatke o studentima, i “`OCJENE.TXT`” koja čuva podatke o ostvarenim rezultatima. Datoteka “`STUDENTI.TXT`” organizirana je tako da za svakog studenta prvi red sadrži njegovo ime i prezime, a drugi red njegov broj indeksa. Datoteka “`OCJENE.TXT`” organizirana je tako da se za svaki položeni ispit u jednom redu nalazi prvo broj indeksa studenta koji je položio ispit, a zatim ocjena koju je student ostvario na ispitu. Na primjer, datoteke “`STUDENTI.TXT`” i “`OCJENE.TXT`” mogli bi izgledati ovako:

STUDENTI.TXT:

Pero Perić

1234

Huso Husić

4132

Marko Marković

3142

Meho Mehic

2341

OCJENE.TXT:

```
1234 7  
4132 8  
1234 6  
2341 9  
1234 10  
4132 9  
1234 8
```

Naredni program za svakog studenta iz datoteke “STUDENTI.TXT” pronalazi njegove ocjene u datoteci “OCJENE.TXT”, računa njegov prosjek, i ispisuje njegovo ime i izračunati prosjek na ekranu (odnosno komentar “NEOCIJENJEN” ukoliko student nema niti jednu ocjenu, kao što je recimo slučaj sa studentom “Marko Marković” u gore navedenom primjeru). U razmatranom programu se za svakog pročitanog studenta iz datoteke “STUDENTI.TXT” vrši prolazak kroz čitavu datoteku “OCJENE.TXT” sa ciljem da se pronađu sve njegove ocjene (datoteka “OCJENE.TXT” se svaki put iznova otvara pri svakom prolasku kroz vanjsku petlju). Slijedi i kompletan prikaz programa:

```
#include <iostream>  
#include <iomanip>  
#include <fstream>  
  
using namespace std;  
  
int main() {  
    ifstream studenti("STUDENTI.TXT");  
    for(;;) {  
        char ime[100];  
        int indeks, tekuci_indeks, tekuca_ocjena, broj_ocjena(0);  
        double suma_ocjena(0);  
        studenti.getline(ime, sizeof ime);  
        if (!studenti) break;  
        studenti >> indeks;  
        studenti.ignore(10000, '\n');  
        ifstream ocjene("OCJENE.TXT");  
        while (ocjene >> tekuci_indeks >> tekuca_ocjena)  
            if (tekuci_indeks == indeks) {  
                suma_ocjena += tekuca_ocjena;  
                broj_ocjena++;  
            }  
        cout << ime << " ";  
        if (suma_ocjena == 0) cout << "NEOCIJENJEN\n";  
    }  
}
```

```

    else cout << setprecision(2) << suma_ocjena / broj_ocjena << endl;
}
return 0;
}

```

Obratimo pažnju na poziv metode “`ignore`” nad ulaznim tokom “`studenti`”. Ovo je, kao što već znamo, potrebno raditi kada god iz bilo kakvog ulaznog toka nakon čitanja brojčanih podataka pomoću operatora “`>>`” želimo čitati tekstualne podatke pozivom metode “`getline`”. U suprotnom, oznaka za kraj reda koja je ostala u spremniku može dovesti do pogrešne interpretacije, kao i pri unosu sa tastature. Također, primijetimo da se objekat ulaznog toka “`ocjene`” svaki put iznova stvara i uništava unutar tijela “`for`” petlje (kao i svi objekti koji su lokalno deklarirani unutar tijela petlje). Ovo garantira da će se čitanje datoteke “`OCJENE.TXT`” uvijek vršiti ispočetka pri svakom prolasku kroz petlju.

Datoteke koje smo do sada razmatrali bile su isključivo *tekstualne datoteke*, što znači da su svi podaci u njima bili pohranjeni isključivo kao niz znakova, koji su interno predstavljeni pomoću svojih ASCII šifri (stoga se tekstualne datoteke često nazivaju i ASCII datoteke). Na primjer, broj 35124318 u tekstualnoj datoteci čuva se kao slijed znakova ‘3’, ‘5’, ‘1’, ‘2’, ‘4’, ‘3’, ‘1’ i ‘8’, odnosno, kao slijed brojeva 51, 53, 49, 50, 52, 51, 49 i 56 uzmemu li u obzir njihove ASCII šifre. Međutim, podaci u memoriji računara nisu ovako interno zapisani. Na primjer, ukoliko se isti broj 35124318 nalazi u nekoj cijelobrojnoj promjenljivoj, ona će (uz pretpostavku da cijelobrojne promjenljive zauzimaju 32 bita) biti predstavljena kao odgovarajući binarni broj, tj. broj 0000001000010111111010001011110. Razložimo li ovaj binarni broj u 4 bajta, dobijamo binarne brojeve 00000010, 00010111, 11110100 i 01011110, što nakon pretvaranja u dekadni brojni sistem daje 2, 23, 244 i 94, što se očigledno osjetno razlikuje od zapisa u obliku ASCII šifri pojedinih cifara.

Pored tekstualnih datoteka, jezik C++ podržava i tzv. *binarne datoteke*, čiji sadržaj u potpunosti odražava način na koji su podaci zapisani u memoriji računara. U ovakvim datotekama, podaci nisu zapisani kao slijed znakova sa ASCII šiframa, stoga se njihov sadržaj ne može kreirati niti pregledati pomoću tekstualnih editora kao što je Notepad (preciznije, pokušaj njihovog učitavanja u tekstualni editor prikazaće potpuno besmislen sadržaj, s obzirom da će editor pokušati da interpretira njihov sadržaj kao ASCII šifre, što ne odgovara stvarnosti). Na fundamentalnom fizičkom nivou između tekstualnih i binarnih datoteka nema nikakve razlike – datoteka je samo hrpa povezanih bajtova na eksternoj memoriji. Stoga je jedina razlika između tekstualnih i binarnih razlika u *interpretaciji*. Drugim riječima, korisnik datoteke (programer) mora biti svjestan šta njen sadržaj predstavlja, i da se ponaša u skladu sa tim (tj. da ne pokušava interpretirati sadržaj binarne datoteke kao slijed znakova i obrnuto).

Binarne datoteke imaju izvjesne prednosti u odnosu na tekstualne datoteke. Kako one direktno odražavaju stanje računarske memorije, računar može efikasnije čitati njihov sadržaj i vršiti upis u njih, jer nema potrebe za konverzijom podataka iz zapisa u vidu ASCII šifara u internu binarnu reprezentaciju i obrnuto. Također, zapis u binarnim datotekama je nerijetko kraći u odnosu na zapis u binarnim datotekama (kao u prethodnom primjeru zapisa broja 35124318, koji zahtijeva 8 bajtova u tekstualnoj, a 4 bajta u binarnoj datoteci), ali ne uvijek. Pored toga, programi koji barataju sa binarnim datotekama često su znatno kraći i efikasniji nego programi

koji manipuliraju sa tekstualnim datotekama (mada su ponekad nešto teži za shvatiti, s obzirom da se barata sa internom reprezentacijom podataka u memoriji, koja ljudima nije toliko bliska koliko odgovarajući tekstualni zapis). Kao manu binarnih datoteka možemo navesti nemogućnost njihovog kreiranja i pregledanja njihovog sadržaja putem tekstualnih editora. Naime, dok tekstualnu datoteku koju želimo obrađivati u nekom programu lako možemo kreirati nekim tekstualnim editorom, za kreiranje binarne datoteke nam je potreban vlastiti program. Ovaj nedostatak binarnih datoteka može nekad da postane i prednost, s obzirom da je sadržaj binarnih datoteka zaštićen od radoznalaca koji bi željeli da pregledaju ili eventualno izmijene sadržaj datoteke. Na taj način, podaci pohranjeni u binarnim datotekama su u nekom smislu "sigurniji" od podataka pohranjenih u tekstualnim datotekama.

Za rad sa binarnim datotekama namijenjene su metode "read" i "write". Objekti klase "ifstream" poznaju metodu "read", objekti klase "ofstream" metodu "write", dok objekti klase "fstream" poznaju obje metode. Ove metode zahtijevaju dva parametra. Prvi parametar je adresa u memoriji računara od koje treba započeti smještanje podataka pročitanih iz datoteke odnosno adresa od koje treba započeti prepisivanje podataka iz memorije u datoteku (ovaj parametar je najčešće adresa neke promjenljive ili niza), dok drugi parametar predstavlja broj bajtova koje treba pročitati ili upisati, i obično se zadaje preko "sizeof" operatora. Nažalost, metode "read" i "write" su deklarirane tako da kao svoj prvi argument obavezno zahtijevaju parametar koji je *pokazivač na znakove* (tj. pokazivač na tip "char"), a ne pokazivač proizvoljnog tipa. Zbog toga će u praksi gotovo uvijek biti potrebna konverzija stvarnog pokazivača koji sadrži adresu nekog objekta u memoriji u pokazivač na tip "char" primjenom operatora za konverziju tipova.

Rad sa binarnim datotekama ćemo prvo ilustrirati na jednom jednostavnom primjeru. Sljedeći program traži od korisnika da unese 10 cijelih brojeva sa tastature, koji se prvo smještaju u niz, a zatim se čitav sadržaj niza "istresa" u binarnu datoteku nazvanu "BROJEVI.DAT":

```
#include <fstream>
#include <iostream>

using namespace std;

int main() {
    int niz[10];
    for(int i = 0; i < 10; i++) cin >> niz[i];
    ofstream izlaz("BROJEVI.DAT", ios::out | ios::binary);
    izlaz.write((char*)niz, sizeof niz);
    return 0;
}
```

Nakon pokretanja ovog programa, u tekućem direktoriju biće kreirana datoteka

“BROJEVI.DAT”. Ukoliko bismo ovu datoteku učitali u neki tekstualni editor, dobili bismo besmislen sadržaj, iz razloga koje smo već spomenuli. Stoga, binarnim datotekama nije dobro davati nastavke poput “.TXT” itd, jer njihov sadržaj nije u tekstualnoj formi, i ekstenzija “.TXT” mogla bi zbuniti korisnika (a možda i operativni sistem). Ukoliko kasnije želimo da pročitamo sadržaj ovakve datoteke, moramo za tu svrhu napraviti program koji će koristiti metodu “read”. Na primjer, sljedeći program će učitati sadržaj kreirane binarne datoteke “BROJEVI.DAT” u niz, a zatim ispisati sadržaj učitanog niza na ekran (radi jednostavnosti, ovom prilikom ćemo izostaviti provjeru da li datoteka zaista postoji):

```
#include <fstream>
#include <iostream>

using namespace std;

int main() {
    int niz[10];
    ifstream ulaz("BROJEVI.DAT", ios::in | ios::binary);
    ulaz.read((char*)niz, sizeof niz);
    for(int i = 0; i < 10; i++) cout << niz[i] << endl;
    return 0;
}
```

U oba slučaja, kao prvi parametar metoda “write” i “read” iskorišteno je ime niza “niz”. Poznato je da se ime niza upotrijebljeno samo za sebe automatski konvertira u pokazivač na prvi element niza (tj. u “&niz[0]”), koji se operatorom konverzije “(char*)” konvertira u pokazivač na znakove čisto da bi se zadovoljila forma metoda “write” odnosno “read”. Dalje, treba primijetiti da smo u oba slučaja prilikom otvaranja binarne datoteke kao drugi parametar konstruktoru naveli i opciju “ios::binary”, koja govori da se radi o binarnoj datoteci. Može se postaviti pitanje zbog čega je ova opcija neophodna, ukoliko na fizičkom nivou ne postoji nikakva razlika između tekstualnih i binarnih datoteka. Razlog je u sljedećem: pri radu sa tekstualnim datotekama, program ima pravo da izmjeni brojčanu reprezentaciju izvjesnih znakova da omogući njihov ispravan tretman na operativnom sistemu na kojem se program izvršava (to se najčešće dešava sa znakom za prelaz u novi red ‘\n’ koji se na nekim operativnim sistemima predstavlja kao bajt 10, na nekim kao bajt 13, a na nekim kao par bajta 13,10). S druge strane, opcija “ios::binary” prosto govori da nikakve izmjene bajtova ne smiju da se vrše, jer one ne predstavljaju šifre znakova, već interni zapis podataka u računarskoj memoriji. Na nekim operativnim sistemima, kao što je UNIX, opcija “ios::binary” nema nikavog efekta i prosto se ignorira, dok kod drugih operativnih sistema izostavljanje ove opcije može ponekad dovesti do neželjenih efekata pri radu sa binarnim datotekama. Stoga je izuzetno preporučljivo uvijek navoditi ovu opciju prilikom bilo kakvog

rada sa binarnim datotekama.

U navedenim primjerima, cijeli niz smo prosto jednom naredbom “istresli” u binarnu datoteku, a zatim smo ga također jednom naredbom “pokupili” iz datoteke. Na taj način smo dobili veoma jednostavne programe. Međutim, ovdje nastaje problem što smo pri tome morali znati da niz ima 10 elemenata. Nema nikakvog razloga da i elemente binarne datoteke ne čitamo jedan po jedan, odnosno da ih ne upisujemo jedan po jedan. Naime, i kod binarnih datoteka postoji “kurzor” koji govori dokle smo stigli sa čitanjem, odnosno pisanjem. Tako, ukoliko smo svjesni činjenice da su elementi nekog niza smješteni u memoriju sekvencijalno, jedan za drugim, sasvim je jasno da su i elementi niza pohranjeni u datoteci također organizirani tako da iza prvog elementa slijedi drugi, itd. (jedino što elementi nisu pohranjeni u vidu skupine znakova). Ovo nam omogućava da elemente datoteke “**BROJEVI.DAT**” možemo iščitavati i ispisivati na ekran *jedan po jedan*, bez potrebe za učitavanjem čitavog niza. Na taj način uopće ne moramo znati koliko je elemenata niza zapisano u datoteku, kao što je izvedeno u sljedećem programskom isječku:

```
ifstream ulaz("BROJEVI.DAT", ios::in | ios::binary);
for(;;) {
    int broj;
    ulaz.read((char*)&broj, sizeof broj);
    if(!ulaz) break;
    cout << broj << endl;
}
```

Primjetimo da je u ovom slučaju bilo neophodno korištenje adresnog operatora “**&**” ispred imena promjenljive “**broj**”. Naime, metoda “**read**” kao prvi parametar zahtijeva *adresu* objekta u koji se podatak treba učitati, a za razliku od imena nizova, imena običnih promjenljivih se ne konvertiraju automatski u pokazivače. Interesantno je još napomenuti da metode “**read**” i “**write**” kao rezultat vraćaju referencu na tok nad kojim su pozvane, što omogućava da se prethodni programski isječak skraćeno napiše na sljedeći način (s obzirom da znamo da se tok koji je u neispravnom stanju ponaša kao logička neistina ukoliko se upotrijebi kao uvjet unutar “**if**” naredbe):

```
ifstream ulaz("BROJEVI.DAT", ios::in | ios::binary);
int broj;
while(ulaz.read((char*)&broj, sizeof broj))
    cout << broj << endl;
```

Kao što elemente binarne datoteke možemo čitati jedan po jedan, moguće je vršiti i upis elemenata jedan po jedan. Naravno, u slučaju da treba čitav niz pohraniti u datoteku, najbolje je to uraditi jednom naredbom (što je velika ušteda u pisanju u odnosu na slučaj kada bismo

elemente niza htjeli da pohranimo u tekstualnu datoteku, naročito ukoliko se radi o nizu čiji su elementi strukture ili klase). Međutim, ukoliko imamo jakog razloga za to, niko nam ne brani da elemente niza u binarnu datoteku upisujemo jedan po jedan. Na primjer, ukoliko želimo da u binarnu datoteku smjestimo sve elemente niza osim elementa sa indeksom 6, to možemo uraditi ovako:

```
ofstream izlaz("BROJEVI.DAT", ios::out | ios::binary);
for(int i = 0; i < 10; i++)
    if(i != 6) izlaz.write((char*)&niz[i], sizeof niz[i]);
```

Adresni operator “`&`” ispred “`niz[i]`” je također potreban, jer se *indeksirani* element niza ne interpretira automatski kao adresa tog elementa (izostavljanjem adresnog operatora bi sama vrijednost a ne adresa elementa “`niz[i]`” bila pretvorena u pokazivač operatorom konverzije “`(char*)`”, što bi na kraju dovelo do upisa sasvim pogrešnog dijela memorije u datoteku). Također, kao argument operatora “`sizeof`” umjesto “`niz[i]`” mogli smo upotrijebiti i izraz “`niz[0]`”, jer svi elementi nekog niza imaju istu veličinu. Naravno, mogli smo pisati i “`sizeof(int)`” s obzirom da znamo da su svi elementi niza tipa “`int`”. Međutim, prethodna rješenja su fleksibilnija, s obzirom da ne zahtijevaju nikakve izmjene ukoliko se odlučimo da promjenimo tip elemenata niza “`niz`”.

Iz izloženih primjera nije teško izvući jedan opći zaključak. Bez obzira da li radimo sa tekstualnim ili binarnim datotekama, onaj ko čita njihov sadržaj *mora biti svjestan njihove strukture* da bi mogao da ispravno interpretira njihov sadržaj. Drugim riječima, nije potrebno poznavati tačno sadržaj datoteke, ali se mora znati šta njen sadržaj predstavlja. Bitno je shvatiti da sama datoteka predstavlja samo hrpu bajtova, i nikakve informacije o njenoj strukturi nisu u njoj pohranjene. O tome treba da vodi računa isključivo onaj ko čita datoteku. Na primjer, sasvim je moguće kreirati niz studenata i istresti čitav sadržaj tog niza u binarnu datoteku, a zatim učitati tu istu binarnu datoteku u niz realnih brojeva. Nikakva greška neće biti prijavljena, ali će sadržaj niza biti potpuno besmislen. Naime, program će prosto hrpu binarnih brojeva koji su predstavljeni zapise o studentima učitati u niz realnih brojeva, i interpretirati istu hrpu binarnih brojeva kao zapise realnih brojeva!

U normalnim okolnostima, prilikom čitanja iz datoteke ili upisa u datoteku, kurzor koji određuje mjesto odakle se vrši čitanje, odnosno mjesto gdje se vrši pisanje uvijek se pomjera *unaprijed*. Međutim, moguće je kurzor pozicionirati na *proizvoljno mjesto* u datoteci. Za tu svrhu, mogu se koristiti metode “`seekg`” odnosno “`seekp`” koje respektivno pomjeraju kurzor za čitanje odnosno pisanje na poziciju zadalu parametrom ovih metoda. Ova mogućnost se najčešće koristi prilikom rada sa binarnim datotekama, mada je principijelno moguća i pri radu sa tekstualnim datotekama (pod uvjetom da veoma dobro pazimo šta radimo). Jedinica mjere u kojoj se zadaje pozicija kurzora je *bajt*, a početak datoteke računa se kao pozicija 0. Tako, ukoliko želimo da čitanje datoteke započne ponovo od njenog početka, ne moramo zatvarati i ponovo otvarati tok, već je dovoljno da izvršimo nešto poput

```
ulaz.seekg(0);
```

U slučaju potrebe, trenutnu poziciju kurzora za čitanje odnosno pisanje možemo saznati pozivom metoda “`tellg`” odnosno “`tellp`”. Ove metode nemaju parametara.

Metode `seekg` odnosno “`seekp`” mogu se koristiti i sa dva parametra, pri čemu drugi parametar može imati samo jednu od sljedeće tri vrijednosti: “`ios::beg`”, “`ios::end`” i “`ios::cur`”. Vrijednost “`ios::beg`” je vrijednost koja se podrazumijeva ukoliko se drugi parametar izostavi, i označava da poziciju kurzora želimo da zadajemo računajući od *početka datoteke*. S druge strane, ukoliko je drugi parametar “`ios::end`”, tada se nova pozicija kurzora određena prvim parametrom zadaje računajući od *kraja datoteke*, i treba da bude *negativan broj* (npr. `-1` označava poziciju koja se nalazi jedan bajt prije kraja datoteke). Konačno, ukoliko je drugi parametar “`ios::cur`”, tada se nova pozicija kurzora određena prvim parametrom zadaje *relativno u odnosu na tekuću poziciju kurzora*, koja tada može biti kako pozitivan, tako i negativan broj. Sljedeći veoma jednostavan primjer ispisuje na ekran koliko je dugačka (u bajtima) datoteka povezana sa ulaznim tokom “`ulaz`”:

```
ulaz.seekg(0, ios::end);
cout << "Datoteka je dugačka " << ulaz.tellg() << " bajtova";
```

Metoda “`seekg`” nam omogućava da, uz izvjestan trud, možemo iščitavati datoteke proizvoljnim redom, a ne samo sekvensijalno (od početka ka kraju). Ovo je iskorišteno u sljedećem programskom isječku koji iščitava ranije kreiranu datoteku “`BROJEVI.DAT`” u obrnutom poretku, od kraja ka početku:

```
ifstream ulaz("BROJEVI.DAT", ios::in | ios::binary);
ulaz.seekg(0, ios::end);
int duzina = ulaz.tellg();
int broj_elemenata = duzina / sizeof(int);
cout << "Datoteka je duga " << duzina << " bajtova, a sadrži "
    << broj_elemenata << " elemenata.\n";
cout << "Slijedi prikaz sadržaja datoteke naopačke:\n";
for(int i = broj_elemenata - 1; i >= 0; i--) {
    int broj;
    ulaz.seekg(i * sizeof(int));
    ulaz.read((char*)&broj, sizeof broj);
    cout << broj << endl;
}
```

Slično, metoda “`seekp`” nam omogućava da vršimo upis na proizvoljno mjesto u datoteci, a ne samo na njen kraj. Ovo svojstvo iskorišteno je u sljedećem programskom isječku koji udvostručava sadržaj svih elemenata binarne datoteke “`BROJEVI.DAT`”, bez potrebe da se prethodno svi elementi učitaju u niz, pomnože sa dva, a zatim modificirani niz vrati nazad u datoteku. Ovdje je također iskorištena osobina da se objekti klase “`fstream`” mogu otvoriti istovremeno kao ulazni i kao izlazni tokovi:

```
int main() {
    fstream datoteka("BROJEVI.DAT", ios::in | ios::out | ios::binary);
    int broj;
    while(datoteka.read((char*)&broj, sizeof broj)) {
        cout << broj << endl;
        broj *= 2;
        datoteka.seekp(-int(sizeof broj), ios::cur);
        datoteka.write((char*)&broj, sizeof broj);
    }
    return 0;
}
```

U ovom programu, metoda “`seekp`” je iskorištena da pomjeri kurzor za pisanje tako da se element koji se upisuje *prepiše* preko upravo procitanog elementa. Neočekivana i na prvi pogled suvišna konverzija rezultata “`sizeof`” operatorka u tip “`int`” neophodna je zbog činjenice da je tip rezultata koji vraća operatorka “`sizeof`” tipa nepredznačnog cijelog broja (“`unsigned`” tipa), tako da na njega operator unarni operator negacije “`-`” ne djeluje ispravno (ovo je česta greška, koju je teško uočiti).

Treba napomenuti da kada se objekti klase “`fstream`” koriste istovremeno i kao ulazni i kao izlazni tokovi, nije dozvoljeno u bilo kojem trenutku prelaziti sa čitanja toka na pisanje u tok i obrnuto. Sa čitanja na pisanje smijemo preći samo nakon što eksplisitno pozicioniramo kurzor za pisanje pozivom metode “`seekp`” (što smo i radili u prethodnom primjeru). Analogno, sa pisanja na čitanje smijemo preći nakon što eksplisitno pozicioniramo kurzor za čitanje pozivom metode “`seekg`”. Pored toga, sa pisanja na čitanje smijemo preći i u trenutku kada je spremnik izlaznog toka ispraznjen, što se uvijek dešava nakon slanja objekta “`endl`” na izlazni tok (ali ne i nakon običnog slanja znaka ‘`\n`’ za novi red, što je jedna od razlika između ponašanja objekta “`endl`” i stringa “`\n`”). Nepridržavanje ovih pravila može imati nepredvidljive posljedice po rad programa, i obično se manifestira pogrešnim čitanjem ili upisivanjem na neočekivano (pogrešno) mjesto u datoteku.

Generalizirajući prethodni primjer, uz malo vještine, mogli bismo napraviti i program koji *sortira* sadržaj datoteke na disku (npr. BubbleSort postupkom), bez njenog učitavanja u niz. Za tu svrhu trebali bismo se dosta “igrati” sa metodama “`seekg`” i “`seekp`” da ostvarimo

čitanje odnosno upis u datoteku u proizvoljnom poretku. Čitatelju odnosno čitateljici se savjetuje da probaju sastaviti takav program kao korisnu vježbu za razumijevanje rada sa binarnim datotekama. Ipak, treba voditi računa da je pristup elementima datoteke znatno sporiji nego pristup elementima u memoriji, tako da “nesekvencijalni” pristup elementima datoteke treba izbjegavati kada god je to moguće. Drugim riječima, ukoliko na primjer želimo sortirati sadržaj datoteke na disku, najbolji pristup je učitati njen sadržaj u niz u memoriji, zatim sortirati sadržaj niza u memoriji, i na kraju, vratiti sadržaj sortiranog niza nazad u datoteku. Alternativne pristupe treba koristiti jedino u slučaju da je datoteka toliko velika da ju je praktički nemoguće učitati u niz u radnoj memoriji, što se ipak dešava prilično rijetko.

Bitno je naglasiti da u slučaju kada vršimo upis u datoteku a kurzor nije na njenom kraju (recimo, ukoliko smo ga pomjerili primjenom metode “`seekp`”), podaci koje upisujemo uvijek se *prepisuju* preko postojećih podataka, odnosno *ne vrši se njihovo umetanje* na mjesto kurzora (kao što smo vidjeli iz prethodnog primjera). Generalno, ne postoji jednostavan način da se podaci *umetnu* negdje u sredinu datoteke. Najjednostavniji način da se to uradi je učitati čitavu datoteku u neki niz, umetnuti ono što želimo na odgovarajuće mjesto u nizu, a zatim tako izmijenjen niz ponovo “istresti” u datoteku. Naravno, ovo rješenje je prihvatljivo samo ukoliko datoteka nije prevelika, tako da se njen sadržaj može smjestiti u niz. U suprotnom, potrebno je koristiti komplikovanija rješenja, koja uključuju kreiranje pomoćnih datoteka (na primjer, prepisati sve podatke iz izvorne datoteke od početka do mjesta umetanja u pomoćnu datoteku, zatim u pomoćnu datoteku dopisati ono što želimo da dodamo, nakon toga prepisati ostatak izvorne datoteke u pomoćnu, i konačno, prepisati čitavu pomoćnu datoteku preko izvorne datoteke). Isto tako, nema jednostavnog načina da se iz datoteke izbriše neki element. Moguća rješenja također uključuju učitavanje čitave datoteke u niz, ili korištenje pomoćne datoteke. U svakom slučaju, pomoćnu datoteku na kraju treba izbrisati. Za brisanje datoteke može se koristiti funkcija “`remove`” iz biblioteke “`cstdio`”, koja kao parametar zahtijeva ime datoteke koju želimo izbrisati. Brisanje je moguće samo ukoliko na datoteku nije vezan ni jedan tok (eventualne tokove koji su bili vezani na datoteku koju želimo izbrisati prethodno treba zatvoriti pozivom metode “`close`”).

Binarne datoteke su naročito pogodne za smještanje sadržaja slogova ili instanci klase u datoteku, s obzirom da je cijeli sadržaj sloga ili instance klase moguće odjednom upisati u datoteku ili pročitati iz datoteke (u slučaju tekstualnih datoteka, svaku komponentu sloga ili klase potrebno je upisati odnosno čitati posebno). Ovo je ilustrirano u sljedećem programu, koji kreira binarnu datoteku “`STUDENTI.DAT`” koja će sadržavati podatke o studentima koji se prethodno unose sa tastature:

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

struct Student {
    char ime[20], prezime[20];
    int indeks, broj_ocjena;
    int ocjene[30];
};
```

```

int main() {
    int broj_studenata;
    cout << "Koliko ima studenata? ";
    cin >> broj_studenata;
    ofstream studenti("STUDENTI.DAT", ios::out | ios::binary);
    for(int i = 1; i <= broj_studenata; i++) {
        cin.ignore(1000, '\n');
        Student neki_student;
        cout << "Unesite podatke za " << i << ". studenta:\n";
        cout << "Ime: ";
        cin.getline(neki_student.ime, sizeof neki_student.ime);
        cout << "Prezime: ";
        cin.getline(neki_student.prezime, sizeof neki_student.prezime);
        cout << "Broj indeksa: ";
        cin >> neki_student.indeks;
        cout << "Broj ocjena: ";
        cin >> neki_student.broj_ocjena;
        for(int j = 1; j <= neki_student.broj_ocjena; j++) {
            cout << "Ocjena iz " << j << ". predmeta: ";
            cin >> neki_student.ocjene[j];
        }
        studenti.write((char*)&neki_student, sizeof(Student));
    }
    return 0;
}

```

Kako se sadržaj binarnih datoteka ne može pregledati tekstualnim editorima, za čitanje ovako kreirane datoteke također treba napisati program. Sljedeći program čita podatke o studentima iz kreirane datoteke, i za svakog studenta ispisuje ime, prezime, broj indeksa i prosjek (prosjek se računa na osnovu podataka o ocjenama), doduše u ne baš najljepšem formatu:

```

#include <iostream>
#include <fstream>

using namespace std;

struct Student {
    char ime[20], prezime[20];
    int indeks, broj_ocjena;
    int ocjene[30];
};

int main() {
    int broj_studenata;
    Student neki_student;
    ifstream studenti("STUDENTI.DAT", ios::in | ios::binary);
    while(studenti.read((char*)&neki_student, sizeof(Student))) {
        double prosjek(0);
        for(int i = 1; i <= neki_student.broj_ocjena; i++)
            prosjek += neki_student.ocjene[i];
        prosjek /= neki_student.broj_ocjena;
        cout << "Ime: " << neki_student.ime << " Prezime: "
            << neki_student.prezime << " " << "Indeks : "
            << neki_student.indeks << " Prosjek: " << prosjek << endl;
    }
    return 0;
}

```

U slučaju kada razvijamo neku kontejnersku klasu (tj. klasu koja je namijenjena za čuvanje neke kolekcije objekata), sasvim je prirodno predviđjeti metode koje snimaju sadržaj primjeraka te klase u datoteku, odnosno koje obnavljaju sadržaj primjeraka te klase iz datoteke. Na primjer, neka smo razvili klasu nazvanu "StudentskaSluzba", koja čuva kolekciju informacija o studentima. U takvu klasu prirodno je dodati metode "Sacuvaj" i "Obnovi" koje će obavljati ove zadatke. U slučaju kada kontejnerska klasa kolekciju objektata čuva u običnom nizu, i kada objekti koji se čuvaju ne sadrže pokazivače ili tipove podataka zasnovane na pokazivačima (kao što su "string" ili "vector"), implementacija ovih metoda je trivijalna. Na primjer, prepostavimo da je klasa "StudentskaSluzba" deklarirana ovako (ovdje su prikazane samo deklaracije koje su bitne za razmatranja koja slijede);

```
class StudentskaSluzba {  
    Student studenti[100];  
    int broj_studenata;  
    ...  
public:  
    ...  
    void Sacuvaj(const char ime_datoteke[]);  
    void Obnovi(const char ime_datoteke[]);  
};
```

Tada bi implementacija metoda "Sacuvaj" i "Obnovi" mogla izgledati ovako:

```
void StudentskaSluzba::Sacuvaj(const char ime_datoteke[]) {  
    ofstream izlaz(ime_datoteke, ios::out | ios::binary);  
    izlaz.write((char*)this, sizeof *this);  
    if(!izlaz) throw "Nešto nije u redu sa upisom!\n";  
}  
  
void StudentskaSluzba::Obnovi(const char ime[]) {  
    ifstream ulaz(ime_datoteke, ios::in | ios::binary);  
    if(!ulaz) throw "Datoteka ne postoji!\n";  
    ulaz.read((char*)this, sizeof *this);  
    if(!ulaz) throw "Nešto nije u redu sa čitanjem!\n";  
}
```

Primijetimo da smo kao prvi parametar metodama "write" odnosno "read" proslijedili pokazivač "this", čime zapravo snimamo (odnosno obnavljamo) upravo sadržaj memorije

koji zauzima objekat nad kojim su pozvane metode “Sacuvaj” ili “Obnovi”. Veličinu objekta saznajemo pomoću izraza “**sizeof *this**” jer “***this**” predstavlja objekat nad kojim je metoda pozvana. Zapravo, umjesto izraza “**sizeof *this**” smo mogli pisati i izraz “**sizeof(StudentskaSluzba)**”, ali na ovaj način ne ovisimo od stvarnog imena klase.

Situacija se komplicira ukoliko kontejnerska klasa koristi dinamičku alokaciju memorije, što je veoma čest slučaj. Zamislimo, na primjer, da je klasa “*StudentskaSluzba*” organizirana ovako:

```
class StudentskaSluzba {
    Student *studenti;
    int broj_studenata;
    const int MaxBrojStudenata;

    ...

public:
    StudentskaSluzba(int kapacitet) : broj_studenata(0),
        MaxBrojStudenata(kapacitet), studenti(new Student[kapacitet]) {}

    ...

    void Sacuvaj(const char ime_datoteke[]);
    void Obnovi(const char ime_datoteke[]);
};
```

U ovom slučaju, prethodne implementacije metoda “Sacuvaj” i “Obnovi” neće raditi kako treba. Naime, u ovom slučaju, sama klasa “*StudentskaSluzba*” *ne sadrži unutar sebe niz studenata*, nego samo *pokazivač na dinamički alociran niz studenata* koji se nalazi negdje u memoriji *izvan prostora koji zauzima sama klasa*. Stoga, metode “Sacuvaj” i “Obnovi” treba prepraviti tako da uzmu u obzir ovu činjenicu. Prepravka metode “Sacuvaj” je trivijalna: samo je pored sadržaja same klase potrebno u datoteku snimiti i sadržaj dinamički alociranog niza. Njegovu adresu znamo preko pokazivača “*studenti*”, a dužinu možemo odrediti jednostavnim računom kao broj elemenata niza pomnožen sa veličinom jednog elementa niza:

```
void StudentskaSluzba::Sacuvaj(const char ime_datoteke[]) {
    ofstream izlaz(ime_datoteke, ios::out | ios::binary);
    izlaz.write((char*)this, sizeof *this);
    izlaz.write((char*)studenti, broj_studenata * sizeof *studenti);
    if(!izlaz) throw "Nešto nije u redu sa upisom!\n";
}
```

Obnavljanje sadržaja iz datoteke je nešto komplikovanije. Naime, vrijednost pokazivača

“studenti” kakav je sačuvan u datoteci predstavlja adresu dinamičkog niza kakva je bila u vrijeme kada je izvršeno snimanje u datoteku, a to gotovo sigurno nije ista adresa na kojoj je kreiran dinamički niz u trenutku kada želimo da izvršimo obnavljanje iz datoteke. Stoga, vrijednost pokazivača “studenti” sačuvanog u datoteci treba ignorirati, a umjesto njega koristiti aktuelnu vrijednost pokazivača “studenti” koja pokazuje na aktuelno alocirani prostor u koji treba učitati obnovljeni sadržaj. Pošto će aktuelna vrijednost pokazivača “studenti” biti “ubrljana” čitanjem sadržaja klase iz datoteke, treba njegovu vrijednost sačuvati u pomoćnoj promjenljivoj. Tako bi implementacija metode “Obnovi” mogla izgledati ovako:

```
void StudentskaSluzba::Obnovi(const char ime_datoteke[]) {
    ifstream ulaz(ime_datoteke, ios::in | ios::binary);
    if(!ulaz) throw "Datoteka ne postoji!\n";
    Student *pomocna = studenti;
    ulaz.read((char*)this, sizeof *this);
    studenti = pomocna;
    ulaz.read((char*)studenti, broj_studenata * sizeof *studenti);
    if(!ulaz) throw "Nešto nije u redu sa čitanjem!\n";
}
```

Prikazano rješenje radi uglavnog dobro, ali posjeduje jedan bitan nedostatak. Naime, ukoliko bismo pokušali da sadržaj objekta klase “StudentskaSluzba” izvjesnog kapaciteta probamo obnoviti iz datoteke u koju je snimljen sadržaj objekta klase “StudentskaSluzba” većeg kapaciteta, doći će do kraha programa, jer dinamički alocirani niz u objektu čiji sadržaj želimo da obnovimo nema dovoljnu veličinu da se u njega može učitati čitav snimljeni niz. Bolje rješenje je da u slučaju da je kapacitet objekta čiji sadržaj obnavljamo manji u odnosu na kapacitet objekta koji je snimljen u datoteku izvršimo realokaciju memorije (tj. da povećamo kapacitet razmatranog objekta). Ovo rješenje je izvedeno u sljedećoj implementaciji metode “Obnovi”:

```
void StudentskaSluzba::Obnovi(const char ime_datoteke[]) {
    ifstream ulaz(ime_datoteke, ios::in | ios::binary);
    if(!ulaz) throw "Datoteka ne postoji!\n";
    Student *pomocna = studenti;
    int kapacitet = MaxBrStudenata;
    ulaz.read((char*)this, sizeof *this);
    studenti = pomocna;
    if(kapacitet < MaxBrStudenata) {
        delete[] studenti;
        studenti = new Student[MaxBrStudenata];
```

```

    }

    ulaz.read((char*) studenti, broj_studenata * sizeof *studenti);

    if(!ulaz) throw "Nešto nije u redu sa čitanjem!\n";

}

```

Kao alternativu ovom rješenju možemo navesti i jednostavnije rješenje, koje također radi veoma dobro, ali je nešto manje efikasno, jer se realokacija vrši i kad treba i kad ne treba (ipak, ovo nije velika šteta, s obzirom da je obnova sadržaja klase iz datoteke operacija koja se izvodi prilično rijetko):

```

void StudentskaSluzba::Obnovi(const char ime_datoteke[]) {
    ifstream ulaz(ime_datoteke, ios::in | ios::binary);

    if(!ulaz) throw "Datoteka ne postoji!\n";

    delete[] studenti;

    ulaz.read((char*)this, sizeof *this);
    studenti = new Student[MaxBrStudenata];
    ulaz.read((char*)studenti, broj_studenata * sizeof *studenti);
    if(!ulaz) throw "Nešto nije u redu sa čitanjem!\n";
}

```

Još složenija situacija nastupa u slučaju kada kontejnerska klasa koristi dinamički alocirane nizove koji ne sadrže same objekte, nego pokazivače na objekte, što je također čest slučaj u praksi, naročito kada su objekti instance neke klase koja sadrži samo konstruktore sa parametrima. U tom slučaju, kao što već znamo, sama kontejnerska klasa sadrži *dvojni pokazivač*. Prepostavimo, na primjer, da je klasa “StudentskaSluzba” organizirana na sljedeći način:

```

class StudentskaSluzba {
    Student **studenti;
    int broj_studenata;
    const int MaxBrojStudenata;
    ...
public:
    StudentskaSluzba(int kapacitet) : broj_studenata(0),
        MaxBrojStudenata(kapacitet), studenti(new Student*[kapacitet]) {}

    ...
    void Sacuvaj(const char ime_datoteke[]);
    void Obnovi(const char ime_datoteke[]);
};

}

```

U ovom slučaju, nakon što u datoteku snimimo sam sadržaj klase, ne trebamo u datoteku snimati sam dinamički niz na koji pokazuje pokazivač "studenti". Naime, ovaj niz ne sadrži same objekte koje želimo da snimimo, nego *pokazivače na njih*. Umjesto toga, u datoteku je potrebno snimiti *sve objekte na koje pokazuju pokazivači koji se nalaze u nizu na koji pokazuje pokazivač "studenti"*. Stoga bi u ovom slučaju, metoda "Sacuvaj" mogla izgledati ovako (primijetimo da je umjesto izraza "`sizeof **studenti`" moguće pisati izraz "`sizeof *studenti[i]`" ili "`sizeof(Student)`"):

```
void StudentskaSluzba::Sacuvaj(const char ime_datoteke[]) {
    ofstream izlaz(ime_datoteke, ios::out | ios::binary);
    izlaz.write((char*)this, sizeof *this);
    for(int i = 0; i < broj_studenata; i++)
        izlaz.write((char*)studenti[i], sizeof **studenti);
    if(!izlaz) throw "Nešto nije u redu sa upisom!\n";
}
```

Za pisanje metode "Obnovi" možemo iskoristiti sličnu logiku. Ovdje je prikazana jednostavnija izvedba metode, koja vrši realokaciju memorije neovisno od toga da li je ona zaista potrebna ili nije:

```
void StudentskaSluzba::Obnovi(const char ime_datoteke[]) {
    ifstream ulaz(ime_datoteke, ios::in | ios::binary);
    if(!ulaz) throw "Datoteka ne postoji!\n";
    for(int i = 0; i < broj_studenata; i++) delete studenti[i];
    delete[] studenti;
    ulaz.read((char*)this, sizeof *this);
    studenti = new Student*[MaxBrStudenata];
    for(int i = 0; i < broj_studenata; i++) {
        studenti[i] = new Student;
        ulaz.read((char*)studenti[i], sizeof **studenti);
    }
    if(!ulaz) throw "Nešto nije u redu sa čitanjem!\n";
}
```

U ovom primjeru, prvo vršimo potpunu destrukciju postojećeg objekta (odnosno dealokaciju alociranog prostora za sve studente, kao i niza pokazivača koji čuva njihove adrese), nakon čega vršimo ponovnu konstrukciju objekta prilikom obnavljanja sadržaja iz datoteke. Tom prilikom se unutar petlje za svakog studenta prvo vrši dinamička alokacija odgovarajućeg memoriskog prostora (pri čemu se adresa alociranog prostora dodjeljuje odgovarajućem pokazivaču u dinamičkom nizu "studenti") prije nego što se podaci o odgovarajućem studentu pročitaju iz datoteke. Ovo je neophodno s obzirom na činjenicu da nakon kreiranja

dinamičkog niza pokazivača "studenti" (preciznije, dinamičkog niza na čiji prvi element pokazuje pokazivač "studenti") svi njegovi elementi sadrže slučajne vrijednosti, odnosno pokazuju na posve slučajne adrese. U slučaju da tip "Student" nije struktura nego klasa koja zahtijeva konstruktore sa parametrima, prilikom dinamičke alokacije prostora za svakog studenta (pozivom operatora "new") konstruktoru bismo mogli proslijediti bilo kakve parametre pod uvjetom da su legalni (da se ne desi da konstruktor baci izuzetak), s obzirom da će stvarni sadržaj objekta tipa "Student" svakako biti učitan iz datoteke, bez obzira kakav je sadržaj postavio konstruktor. Čitatelju ili čitateljici se savjetuje da probaju napisati efikasniju varijantu metode "Obnovi" koja će obavljati realokaciju memorije samo u slučaju da za njom zaista postoji potreba.

Primijetimo da smo kod implementacije metode "Obnovi" u svim slučajevima imali dodatne komplikacije uzrokovane činjenicom da se ova metoda poziva nad već postojećim i konstruisanim objektom, tako da je potrebno vršiti izmjenu njegove strukture u slučaju da postojeća struktura nije adekvatna da prihvati podatke iz datoteke. Međutim, često se pokazuje praktičnim obnavljanje sadržaja objekta iz datoteke obaviti već u fazi same konstrukcije objekta. Na taj način, izbjegavamo potrebu da prvo konstruiramo potencijalno neadekvatan objekat, a da zatim vršimo njegovu rekonstrukciju u fazi obnavljanja sadržaja iz datoteke. Da bismo ostvarili taj cilj, dovoljno je u klasu dodati konstruktor koji vrši konstrukciju objekta uz obnavljanje sadržaja iz datoteke. Parametar takvog konstruktora može recimo biti ime datoteke iz koje se vrši obnavljanje (taj konstruktor bi trebao biti eksplicitan, da se izbjegne mogućnost automatske konverzije iz znakovnog niza u objekat klase "StudentskaSluzba", koja bi dovela do sasvim neočekivanog obnavljanja sadržaja objekta iz datoteke). Slijedi moguća implementacija takvog konstruktora za posljednji, najsloženiji primjer klase "StudentskaSluzba" u kojoj se koristi dvojni pokazivač (odgovarajući konstruktori za prethodne primjere mogu se napisati na analogan način):

```
StudentskaSluzba::StudentskaSluzba(const char ime_datoteke[]) {
    ifstream ulaz(ime_datoteke, ios::in | ios::binary);
    if(!ulaz) throw "Datoteka ne postoji!\n";
    ulaz.read((char*)this, sizeof *this);
    studenti = new Student*[MaxBrStudenata];
    for(int i = 0; i < broj_studenata; i++) {
        studenti[i] = new Student;
        ulaz.read((char*)studenti[i], sizeof **studenti);
    }
    if(!ulaz) throw "Nešto nije u redu sa čitanjem!\n";
}
```

Već smo rekli da u slučaju da je tip "Student" klasa koja zahtijeva konstruktore sa parametrima, morali bismo prilikom poziva operatora "new" konstruktoru klase "Student"

prosljediti fiktivne parametre, bez obzira što bi se sam sadržaj objekta tipa “Student” kasnije pročitao iz datoteke. Da izbjegnemo ovu nelegantiju, moguće je i samoj klasi “Student” dodati konstruktor koji obnavlja sadržaj jednog objekta tipa “Student” iz datoteke. Parametar ovog konstruktora mogao bi biti ulazni tok iz kojeg se trenutno obnavlja sadržaj objekta tipa “StudentskaSluzba”. Tako bi petlja u kojoj se vrši obnavljanje sadržaja pojedinačnih studenata iz datoteke mogla izgledati prosto ovako:

```
for(int i = 0; i < broj_studenata; i++) {  
    studenti[i] = new Student(ulaz);
```

Odgovarajući konstruktor klase “Student” mogao bi izgledati recimo ovako:

```
Student::Student(ifstream &ulaz) {  
    ulaz.read((char*)this, sizeof *this);  
    if(!ulaz) throw "Nešto nije u redu sa čitanjem!\n";  
}
```

Ovaj konstruktor bi bilo prirodno deklarirati u privatnoj sekciji klase “Student”, čime bismo spriječili da obični korisnici klase “Student” mogu kreirati objekte klase “Student” pomoću ovog konstruktora, koji je očigledno čisto pomoćne prirode. Pored toga bi klasu “StudentskaSluzba” trebalo deklarirati kao prijateljsku klasu klase “Student”, da bi metode klase “StudentskaSluzba” dobile mogućnost da mogu koristiti ovaj konstruktor.

Izloženi primjeri pokazuju da je uvijek potreban izvjestan oprez i vještina kad god treba snimiti u datoteku (odnosno obnoviti iz datoteke) sadržaj neke klase koja u sebi sadrži pokazivače (a pogotovo višestruke pokazivače), jer ti pokazivači tipično pokazuju na dijelove memorije koji ne pripadaju samoj klasi, već se nalaze izvan nje. Prikazani primjeri su sasvim dovoljni da čitalac shvati kako bi se moglo obaviti snimanje u datoteku ili obnavljanje iz datoteke čak i u slučaju klase koje koriste prilično složenu dinamičku alokaciju memorije. Čitateljima i čitateljicama se savjetuje za vježbu da prošire klasu “Matrica” koja je razvijena u prethodnim poglavljima metodama koje snimaju sadržaj matrice u binarnu datoteku, odnosno obnavljaju sadržaj matrice iz binarne datoteke, s obzirom da klasa “Matrica” koristi dosta komplikiran mehanizam dinamičke alokacije memorije (složeniji od svih primjera razmotren u ovom poglavlju).

Na kraju ovog poglavlja, napomenimo da po svaku cijenu treba izbjegavati snimanje u binarnu datoteku objekata čiji su atributi tipa “vector”, “string” ili bilo kojeg tipa čiji se interni rad zasniva na pokazivačima (preciznije, bilo kojeg tipa koji nije POD tip podataka). Na primjer, pretpostavimo da su struktura “Student” i niz “studenti” deklarirani ovako:

```
struct Student {
```

```
string ime, prezime;
int indeks, broj_ocjena;
vector<int> ocjene;
};

Student studenti[100];
```

Pokušaj da sadržaj niza “studenti” snimimo u binarnu datoteku na sljedeći način

```
ofstream izlaz("STUDENTI.DAT", ios::out | ios::binary);
izlaz.write((char*)studenti, sizeof studenti);
```

završiće katastrofalnim gubitkom informacija. Naime, na ovaj način, u datoteku će biti snimljen samo čisti sadržaj niza “studenti”. Njegovi su članovi tipa “Student”, i čisti sadržaj svakog od objekata tipa “Student” će zaista biti snimljen u datoteku. Međutim, struktura “Student” sadrži attribute “ime” i “prezime” tipa “string”, kao i atribut “ocjene” tipa “vector<int>”. Nažalost, objekti tipa “string” odnosno “vector” su vlasnici podataka koji uopće nisu sadržani u njima samima, već su sa njima vezani putem pokazivača. Tako, na primjer, tip “string” ne sadrži u sebi sam niz znakova koji predstavlja, već samo pokazivač na njegov početak, koji se nalazi alociran negdje drugdje u memoriji. Slično vrijedi i za tip “vector”. Slijedi da bi u odgovarajuću datoteku bili snimljeni samo pokazivači, a ne i stvarni podaci! Mada postoji način da se u binarnu datoteku ostvari propisno snimanje i ovakvih tipova podataka, odgovarajući mehanizam je prilično komplikovan i traži detaljno poznavanje internog rada ovih tipova podataka, tako da definitivno nije pogodan za početnike. Zbog toga je u takvim slučajevima najjednostavnije rješenje posve zaobići korištenje binarnih datoteka, i za čuvanje odnosno obnavljanje sadržaja objekata koristiti čisto tekstualne datoteke.

37. DINAMIČKE STRUKTURE PODATAKA

Do sada smo pretežno koristili takve strukture podataka čija je veličina uvijek bila *tačno poznata u trenutku njihovog kreiranja*. U slučaju običnih nizova, njihova veličina morala je biti poznata još u vrijeme pisanja programa, dok je dinamička alokacija memorije omogućavala odgodu zadavanja veličine niza do trenutka njegovog kreiranja. Međutim, u oba slučaja, prilikom kreiranja veličina niza je morala biti zadana. S druge strane, dinamičke strukture podataka mijenjaju svoju veličinu tokom rada: dodavanjem novih podataka u dinamičku strukturu njihova veličina raste, a vađenjem podataka iz dinamičke strukture njihova veličina se *smanjuje*.

Dinamičke strukture podataka ne postoje kao ugrađeni tipovi podataka u jeziku C++, ali se mogu relativno lako napraviti vještim korištenjem pokazivača i dinamičke alokacije memorije. Mada se za kreiranje dinamičkih struktura podataka ne moraju koristiti klase (sto omogućava njihovo kreiranje i u jeziku C koji ne poznaje klase), pogodnost enkapsulacije i sakrivanja informacija koju pružaju klase razlog su što se u jeziku C++ dinamičke strukture podataka realiziraju gotovo isključivo kao klase. Stoga ćemo se i mi odlučiti za takav pristup. Treba napomenuti da se mnoge od stanadrdnih dinamičkih struktura podataka već nalaze implementirane u vidu generičkih klasa u standardnim bibliotekama jezika C++. Među njima su najpoznatije generičke klase ‐list”, ‐stack”, ‐queue”, ‐deque”, ‐set”, ‐multiset”, ‐map”, ‐multimap” i još neke druge. Upotrebu ovih klasa lako je savladati, pa se zainteresirani čitatelji i čitateljice upućuju na raznovrsnu šиру literaturu koja obrađuje standardnu biblioteku jezika C++ (primjere upotrebe nekih od ovih klasa biće navedeni i u ovom tekstu). Za edukativne svrhe znatno je korisnije da probamo da sami razvijemo neke od sličnih klasa, pa ćemo tako i ovdje učiniti. U neku ruku, generička klasa ‐vector” (odnosno generička klasa ‐AdaptivniNiz” koju smo sami razvili) na neki način također predstavlja dinamičku strukturu podataka, s obzirom da njene metode ‐resize” i ‐push_back” (odnosno ‐Redimenzioniraj” i ‐DodajNaKraj”) omogućavaju promjenu veličine same strukture podataka. Međutim, mehanizam promjene veličine ovih struktura podataka zasniva se na *realokaciji* i kopiranju čitavog sadržaja strukture podataka prilikom svake realokacije, što je prilično neefikasno. U ovom poglavlju ćemo razmotriti fleksibilnije načine za realiziranje dinamičkih struktura podataka, koji nisu zasnovani na realokaciji.

Principle na kojima se zasnivaju dinamičke strukture podataka najlakše je objasniti na primjeru dinamičke implementacije apstraktног tipа podataka poznatog pod nazivom *stek* ili *stog* (engl. *stack*), o kojem smo već govorili. Podsjetimo se da je stek kontejnerska struktura podataka zasnovana na LIFO (Last In First Out) principu, odnosno podatak koji se posljednji stavlja na stek ujedno je i podatak koji se prvi skida sa steka. U osnovi implementacije gotovo svih dinamičkih struktura podataka leže specijalne pomoćne strukture podataka nazvane *čvorovi* (engl. *nodes*) o kojima smo već govorili ranije. To su strukture podataka tipa strukture ili klase koja obavezno među svojim atributima sadrži jedan ili više pokazivača (ili referenci) na neku instancu iste strukture ili klase. Na primjer, deklaracija jednog čvora mogla bi izgledati recimo ovako (atribut ‐element” čuva vrijednost pohranjenu u čvoru, dok atribut ‐vezा” predstavlja pokazivač na neki drugi čvor, odnosno predstavlja ‐vezу” sa drugim čvorom):

```

struct Cvor {
    int element;
    Cvor *veza;
};

```

Razmotrimo sada kako nam čvorovi mogu pomoći za dinamičku realizaciju steka. Osnovna ideja sastoji se u tome da svaki put kada želimo da dodamo novi element na stek, dinamički kreiramo novi čvor, u njegov atribut namijenjen za čuvanje vrijednosti smjestimo željeni element, a atribut za vezu postavimo tako da pokazuje na prethodno kreirani čvor (tj. na čvor koji čuva prethodni element na steku). Da bismo znali gdje se nalazi prethodno kreirani čvor, u svakom trenutku moramo pamtiti pokazivač na posljednji kreirani čvor, tj. pokazivač na vrh steka (npr. u atributu klase steka koji ćemo ponovo nazvati “`gdje_je_vrh`”, ali će on ovaj put biti pokazivač, a ne cijelobrojni indeks, kao što smo imali prilikom staticke implementacije steka). U slučaju da je kreirani čvor prvi čvor (tj. kada dodajemo element na prazan stek), njegov atribut veze postavljamo na 0, čime označavamo da on nema svog prethodnika. Također, atribut “`gdje_je_vrh`” će sadržavati 0 kada je stek prazan, tj. kada nema niti jednog kreiranog čvora. Sljedeća slika prikazuje kako će izgledati stanje u memoriji nakon smještanja 5 elemenata na stek (npr. elemenata sa vrijednostima 5, 2, 7, 6, i 3):

Vidimo da se zaista zauzeće memorije povećava kako dodajemo nove elemente na stek, jer se pri tome kreiraju novi čvorovi. Uz ovaku organizaciju, skidanje elemenata sa steka je također jednostavno. Naime, dovoljno je prosto izbrisati iz memorije čvor na koji pokazuje pokazivač “`gdje_je_vrh`”, a pokazivač “`gdje_je_vrh`” preusmjeriti na adresu koju sadrži atribut veze čvora na koji je prethodno pokazivač “`gdje_je_vrh`” pokazivao (tj. čvora koji je upravo obrisan).

Nakon ovog teoretskog uvoda, možemo preći na samu dinamičku implementaciju generičke klase “`Stek`”. S obzirom da nam čvorovi trebaju samo za *internu implementaciju* ove klase, najprirodnije je čvor deklarirati kao “ugniježdenu” strukturu unutar privatne sekcije klase “`Stek`”. Čvor ćemo realizirati kao *strukturu sa konstruktorom*, jer je na taj način moguće inicijalizaciju sadržaja čvora obaviti odmah po njegovom kreiranju, što pojednostavljuje implementaciju (takva struktura je, u suštini, klasa čiji su svi elementi javni, što nam ne smeta, s obzirom da je čitav čvor “zapakovan” unutar privatne sekcije klase “`Stek`”, i nije vidljiv izvan nje). Sada bi deklaracija generičke klase “`Stek`” mogla izgledati recimo ovako:

```

template <typename Tip>
class Stek {
    struct Cvor {

```

```

    Tip element;
    Cvor *veza;
    Cvor(const Tip &element, Cvor *veza)
        : element(element), veza(veza) {}
    };
    Cvor *gdje_je_vrh;
    void Unisti();
    void Kopiraj(const Stek &s);
public:
    Stek() : gdje_je_vrh(0) {}
    Stek(const Stek &s) { Kopiraj(s); }
    ~Stek() { Unisti(); }
    Stek &operator=(const Stek &s);
    bool Prazan() const { return gdje_je_vrh == 0; }
    void Stavi(const Tip &element) {
        gdje_je_vrh = new Cvor(element, gdje_je_vrh);
    }
    Tip Skini();
};

```

Radi kompletnosti, u ovoj klasi smo predvidjeli i destruktur, konstruktor kopije i preklopljeni operator dodjele, s obzirom da se radi o klasi koja dinamički alocira memoriju. Destruktor odnosno konstruktor kopije samo pozivaju privatne metode “Unisti” odnosno “Kopiraj” respektivno, koje su definirane kao posebne metode zbog činjenice da će isti postupci biti korišteni i unutar operatorske funkcije koja implementira preklopljeni operator dodjele. Implementacija metoda “Prazan” i “Stavi” je posve trivijalna, tako da smo njihovu implementaciju izveli odmah unutar deklaracije klase. Nešto je složenija implementacija metode “Skini”:

```

template <typename Tip>
Tip Stek<Tip>::Skini() {
    if(gdje_je_vrh == 0) throw "Stek je prazan!\n";
    Tip element = gdje_je_vrh->element;
    Cvor *prethodni = gdje_je_vrh->veza;
    delete gdje_je_vrh;
    gdje_je_vrh = prethodni;
    return element;
}

```

Implementacija ove metode traži izvjesnu pažnju, jer nakon što obrišemo čvor pozivom operatora “**delete**”, pokazivač na njega postaje viseći pokazivač, i bilo kakav pristup sadržaju memorije na koji on pokazuje veoma je rizičan (*i zabranjen*, po standardu). Zbog toga je, prije nego što obrišemo sam čvor, potrebno pokupiti u pomoćne promjenljive sve informacije koje on sadrži a koje su nam potrebne i nakon njegovog brisanja (u navedenom primjeru, to su pomoćne promjenljive “element” i “prethodni”).

Nakon što je implementirana metoda “Skini” implementacija destruktora (bolje rečeno, funkcije “Unisti” koja se poziva iz destruktora) postaje veoma jednostavna. Naime, da bismo

obrisali stek, dovoljno je u petlji pozivati metodu “*Skini*” sve dok stek ne postane prazan (pri tome, povratni rezultat koji vraća metoda “*Skini*” prosto ignoriramo):

```
template <typename Tip>
void Stek<Tip>::Unisti() {
    while(!Prazan()) Skini();
}
```

Da nismo implementirali destruktor, korisnik klase “*Stek*” bi bio dužan da uvijek sam *isprazni* stek prije nego što odgovarajuća instanca klase “*Stek*” prestane postojati. U suprotnom bi došlo do curenja memorije.

Funkcija “*Kopiraj*”, koja je iskorištena za realizaciju konstruktora kopije i prekopljenog operatora dodjele, relativno je složena, i zahtijeva izvjesne dosjetke. Naime, ova funkcija treba da kreira doslovnu kopiju sadržaja steka, čiji elementi nisu kompaktno zapisani unutar jedne cjeline (npr. unutar nekog niza) nego su razbacani svuda po memoriji, i međusobno uvezani pokazivačima. Pri tome je otežavajuća okolnost činjenica da zbog načina kako su usmjerene veze između čvorova, čvorovima steka koji se kopira možemo pristupiti samo u obrnutom redoslijedu u odnosu na redoslijed njihovog kreiranja. Stoga nam ne preostaje ništa drugo nego da čvorove kopije steka kreiramo upravo u onom redoslijedu kojim možemo pristupati čvorovima steka koji se kopira, a da veze ostvarene pokazivačima usmjerimo tako da se zadrži logička struktura veze između čvorova. Drugim riječima, kopija steka iz prethodnog primjera (sa elementima 5, 2, 7, 6, i 3) stvorena konstruktorom kopije u memoriji će izgledati ovako (uz prepostavku da se čvorovi smještaju u memoriju saglasno redoslijedu kreiranja):

Primijetimo da je memorjska slika steka kreiranog pozivom metode “*Kopiraj*” reflektirana kao slika u ogledalu memorjske slike izvornog steka, ali to ništa ne mijenja njegovu funkcionalnost. U skladu sa opisanim objašnjenjem, konstruktor kopije klase “*Stek*” mogao bi izgledati ovako:

```
template <typename Tip>
void Stek<Tip>::Kopiraj(const Stek &s) {
    Cvor *prethodni, *tekuci = s.gdje_je_vrh;
    gdje_je_vrh = 0;

    while(tekuci != 0) {
        Cvor *novi = new Cvor(tekuci->element, 0);
        if(gdje_je_vrh == 0) gdje_je_vrh = novi;
        else prethodni->veza = novi;
        tekuci = tekuci->veza;
        prethodni = novi;
    }
}
```

Analiza ove metode zahtijeva izvjesnu koncentraciju, s obzirom da se u svakom trenutku koriste tri pokazivača “*tekuci*”, “*novi*” i “*prethodni*”, koji redom pokazuju na čvor izvornog steka koji se upravo kopira, novokreirani čvor kopije steka i prethodno kreirani čvor kopije steka. Da bi se bolje razumio rad ove metode, potrebno je na konkretnom primjeru proći kroz njeno tijelo, i crtati kakvo je stanje memorije u svakom trenutku. Veoma je važno da čitatelj odnosno čitateljica samostalno prođu kroz ovaj postupak, jer njegovo razumijevanje predstavlja ključ za razumijevanje svih ostalih dinamičkih struktura podataka. Kao interesantnu ilustraciju činjenice da fizički raspored čvorova u memoriji uopće nije važan za funkcioniranje struktura podataka zasnovanih na čvorovima, navedimo i to da u slučaju da na kopirani stek dodamo novi element (npr. 8), njegova slika u memoriji će vrlo vjerovatno (uz pretpostavku da se novokreirani čvor smješta iza prethodno kreiranih čvorova) izgledati ovako:

Preklopjeni operator dodjele za klasu “*Stek*” treba obaviti još složeniji zadatak nego konstruktor kopije, jer se dodjela uvijek vrši nad objektom koji *već postoji*. Efikasna izvedba operadora dodjele trebala bi iskopirati elemente izvornog steka u odredišni stek, pri čemu bi u odredišni stek trebala dodati nove čvorove u slučaju da odredišni stek sadrži manje elemenata od izvornog steka, odnosno ukloniti suvišne čvorove u slučaju da odredišni stek sadrži više elemenata od izvornog steka. Ovako izvedena realizacija preklopjenog operadora dodjele bila bi dosta složena (čitatelj odnosno čitateljica mogu pokušati izvesti ovakvu realizaciju kao veoma korisnu vježbu). Znatno jednostavnije (ali i manje efikasno) rješenje je prosto *uništiti* čitav odredišni stek, a zatim ga iznova *rekreirati* kao kopiju izvornog steka, na isti način kao u konstruktoru kopije. Kako je dodjela međusobna dodjela objekta tipa “*Stek*” operacija koja se vjerovatno neće vršiti često, ovakvo manje efikasno rješenje može posve zadovoljiti. Stoga implementacija operatorske funkcije za operator dodjele za klasu “*Stek*” može izgledati ovako:

```
template <typename Tip>
Stek<Tip> &Stek<Tip>::operator =(const Stek<Tip> &s) {
    if(&s == this) return *this;
    Unisti(); Kopiraj(s);
    return *this;
}
```

Obratimo pažnju na prvu liniju tijela ove funkcije. Podsjetimo se da je svrha testiranja “*&s == this*” sprečavanje problema koji bi mogli nastati ukoliko bi se neki objekat klase “*Stek*” pokušao dodijeliti sam sebi (bilo direktno, bilo indirektno putem referenci).

Kako su realizacije konstruktora kopije i preklopjenog operadora dodjele za klasu “*Stek*” relativno složene, često se u literaturi i praksi susreću razne implementacije klase “*Stek*” u kojima su konstruktor kopije i preklopjeni operator dodjele samo deklarirani ali ne i

implementirani, čime se eksplisitno zabranjuje kopiranje i međusobno dodjeljivanje primjeraka ove klase. Ipak, na taj način onemogućavamo prenošenje objekata tipa “*Stek*” po vrijednosti u funkcije, i što je još gore, vraćanje objekata tipa “*Stek*” kao rezultata iz funkcije. Međutim, ukoliko nam to nije potrebno, takvo polurješenje može sasvim zadovoljiti. U svakom slučaju, i takvo polurješenje je mnogo bolje od rješenja koje se zasniva na tome da konstruktor kopije i preklopljeni operator dodjele *ne deklariramo nikako*. U tom slučaju bi se koristili podrazumijevani konstruktor kopije i podrazumijevani operator dodjele, a već smo detaljno govorili o tome kakve probleme ovo uzrokuje kad god klasa sadrži pokazivače na dinamički alocirane objekte (pogotovo u kombinaciji sa destruktorm).

Treba još napomenuti da su prethodno opisane realizacije konstruktora kopije i preklopljenog operatara dodjele dosta neefikasne, s obzirom da se vrši kopiranje čitave interne strukture steka. Moguće je napraviti mnogo efikasnija rješenja, zasnovana na brojanju referenciranja, o čemu smo već govorili. Naravno, brojanje referenciranja samo po sebi dovodi do plitkih kopija, koje možemo izbjegći kreiranjem duboke kopije kada se ustanovi da je to neophodno. Na primjer, duboku kopiju možemo kreirati unutar metoda “*stavi*” ili “*skini*” ukoliko primjetimo da više primjeraka klase “*Stek*” dijeli istu skupinu čvorova. Dalje, moguće je da svaki čvor posjeduje svoj vlastiti brojač referenciranja, tako da je moguće da više različitih primjeraka klase “*Stek*” dijele neke zajedničke čvorove u memoriji. Na taj način možemo dobiti vrlo efikasne implementacije konstruktora kopije i preklopljenog operatara dodjele, kod kojih je broj kopiranja zaista sveden na najnužniji minimum, i to samo u slučaju izričite potrebe. Ovdje smo samo izložili osnovne ideje, dok su same realizacije ovih ideja prilično složene, i na ovom mjestu ih nećemo izlagati.

Stek je izuzetno važna struktura podataka za praktične primjene, s obzirom da se mnogi važni algoritmi za rješavanje praktičnih problema zasnivaju na upotrebi steka. Stoga, standardna biblioteka “*stack*” jezika C++ sadrži istoimenu generičku klasu, koja je veoma slična po funkcionalnosti klasi “*Stek*” koju smo upravo razvili. Generička klasa “*stack*” sadrži metode “*empty*” i “*push*”, koje su po funkcionalnosti identične kao metode “*Prazna*” i “*Stavi*” klase “*Stek*”. Umjesto metode “*skini*”, klasa “*stack*” koristi dvije metode bez parametara nazvane “*top*” i “*pop*”. Metoda “*top*” vraća kao rezultat element koji se nalazi na vrhu steka, ali ga *ne skida sa steka*, dok metoda “*pop*” skida element sa vrha steka i *ne vraća nikakav rezultat*. Stoga bi se primjer koji stavlja na stek brojeve 5, 2, 7, 6 i 3 a zatim ih skida sa steka i ispisuje (čime dobijamo ispis u obrnutom poretku) korištenjem standardne generičke klase “*stek*” mogao napisati ovako:

```
stack<int> s;
s.push(5);
s.push(2);
s.push(7);
s.push(6);
s.push(3);
```

```

while (!s.empty()) {
    cout << s.top() << endl;
    s.pop();
}

```

Metoda “`top`” zapravo kao rezultat vraća *referencu na objekat koji se nalazi na vrhu steka*, što omogućava izmjenu objekta na vrhu steka konstrukcijama poput “`s.top() = 4`” bez potrebe da prvo skidamo element sa steka, a da zatim stavljamo novi element na stek. Međutim, to je ujedno i pomalo opasno. Naime, ukoliko deklariramo referencu koju vežemo za rezultat vraćen iz metode “`top`”, takva referenca će postati ilegalna (divlja) ukoliko nakon toga pozivom metode “`pop`” uklonimo taj element sa steka. Vjerovatnoća da učinimo ovako nešto nije prevelika, ali ipak treba ukazati na moguće izvore eventualnih problema.

Većina dinamičkih struktura podataka realizira se na sličnom principu kao i upravo razvijena klasa “`Stek`”. Zbog toga ćemo, prije nego što razmotrimo realizacije drugih dinamičkih struktura podataka, razmotriti još neke aspekte realizacije klase “`Stek`”, koji će kasnije biti primjenljivi i na sve ostale dinamičke strukture podataka. Na prvom mjestu, primijetimo da smo u prikazanom primjeru razvili stek čiji čvorovi pamte *kopije* objekata koji se stavljaju na stek. Ti objekti također mogu biti primjerici neke klase (npr. moguće je napraviti stek čiji su elementi tipa “`Student`”). Međutim, čuvanje kopija čitavih instanci klasa unutar čvorova često nije poželjno. Naime, instance složenijih klasa mogu zauzimati mnogo memorijskog prostora, pa bi se njihovim kopiranjem u čvorove neracionalno trošio memorijski prostor, jer bi se isti podaci čuvali na dva mjesta (u samoj instanci klase, i u njenoj kopiji unutar čvora). Racionalnije rješenje je u čvoru čuvati samo *pokazivač na instancu klase*. Na primjer, prepostavimo da imamo deklaraciju poput

```
Stek<Student> s;
```

gdje je “`Student`” neka klasa čije instance čuvaju podatke o studentima (alternativno, umjesto klase “`Stek`” koju smo sami razvili, možemo koristiti i standardnu klasu “`stack`”, bez bitnijeg utjecaja na stvari na koje želimo ukazati), i neka je “`neki_student`” neka instanca klase “`Student`”. U ovakovom slučaju bi, nakon naredbe poput

```
s.Stavi(neki_student);
```

informacija o istom studentu bila pohranjena na dva mjesta: u promjenljivoj “`neki_student`” i u novostvorenom čvoru steka “`s`”. Međutim, nije nikakav problem deklarirati stek čiji će čvorovi čuvati *adrese objekata tipa* “`Student`”, a ne same objekte tog tipa:

```
Stek<Student*> s;
```

U ovom slučaju bismo stavljanje podataka o studentu na stek ostvarili putem naredbe

```
s.Stavi(&neki_student);
```

Adresni operator u navedenom primjeru je neophodan, s obzirom da na steku zapravo čuvamo adresu promjenljive "neki_student". Također, ne smijemo zaboraviti da će metoda "Skini" umjesto samog objekta tipa "Student" kao rezultat vratiti pokazivač, koji treba ili dereferencirati, ili na vraćeni rezultat primijeniti operator indirektnog pristupa "->". U svakom slučaju, ovako se mnogo racionalnije troši memorija.

Moguće je napraviti kontejnersku klasu koja se sintaksno koristi kao da čuva *objekte*, a zapravo čuva *pokazivače na njih*. Razmotrimo, na primjer, sljedeću klasu nazvanu "IndirektniStek":

```
template <typename Tip>
class Stek {
    struct Cvor {
        Tip *pok;
        Cvor *veza;
        Cvor(Tip *pok, Cvor *veza) : pok(pok), veza(veza) {}
    };
    Cvor *gdje_je_vrh;
public:
    IndirektniStek() : gdje_je_vrh(0) {}
    bool Prazan() const { return gdje_je_vrh == 0; }
    void Stavi(Tip &element) {
        gdje_je_vrh = new Cvor(&element, gdje_je_vrh);
    }
    const Tip &Skini();
};

template <typename Tip>
const Tip &IndirektniStek<Tip>::Skini() {
    if(gdje_je_vrh == 0) throw "Stek je prazan!\n";
    Tip *pok = gdje_je_vrh->pok;
    Cvor *prethodni = gdje_je_vrh->veza;
    delete gdje_je_vrh;
    gdje_je_vrh = prethodni;
    return *pok;
}
```

Sa ovako napisanom klasom mogli bismo imati konstrukcije poput

```
IndirektniStek<Student> s;
...
s.Stavi(neki_student);
```

iako se na steku ne bi čuvala kopija sadržaja promjenljive “neki _student”, već samo njena adresa. Također, metoda “Skini” bi kao rezultat dala sadržaj promjenljive stavljene na stek, a ne pokazivač, bez obzira što se na steku čuva samo pokazivač. Ovo je ostvareno uzimanjem adrese i dereferenciranjem unutar samih izvedbi metoda “Stavi” i “Skini”, tako da se o tome ne mora brinuti korisnik klase “IndirektniStek”, već o tome vodi računa sama klasa.

U prikazanoj implementaciji, radi jednostavnosti prikaza i uštede u prostoru, nismo definirali destruktor, konstruktor kopije i operator dodjele (mada treba voditi računa da su “jednostavnost” i “ušteda u prostoru” često samo dobra isprika za pravi argument – lijnost). Međutim, ovdje je važno da uočimo izvjesne detalje. Prvo, formalni parametar “element” u metodi “Stavi” deklariran je kao *referenca*, i to na *nekonstantni objekat*. Prilično je razumljivo zbog čega je korištena referenca. Da nismo koristili referencu, odnosno da se koristi prenos parametra *po vrijednosti*, formalni parametar “element” bio bi *kopija* stvarnog parametra proslijedenog u metodu, tako da bi adresni operator “&” uzeo adresu te kopije, a ne objekta koji je prenesen u metodu (u slučaju prenosa po referenci, formalni parametar “element” i stvarni parametar faktički predstavljaju *isti objekat*). Međutim, razmotrimo zbog čega nismo koristili referencu na konstantni objekat, kao što obično radimo. Na prvi pogled, mogli bismo koristiti referencu na konstantni objekat, s obzirom da nigdje ne mijenjamo sadržaj referiranog objekta. Prvi, i manje bitan razlog, je što se pokazivačima na nekonstantne objekte (kakav je pokazivač “*pok*” deklariran unutar čvora) ne smiju dodjeljivati adrese konstantnih objekata, jer bi se onda putem takvog pokazivača mogao promijeniti sadržaj konstantnog objekta (inače, skup pravila koja određuju koje su dodjele zabranjene zbog činjenice da bi njihova primjena mogla dovesti do promjene sadržaja konstantnih objekata poznata su pod nazivom *pravila o konzistenciji konstantnosti*). Ovo bi se moglo lako riješiti tako što bismo sam pokazivač “*pok*” deklarirali kao pokazivač na konstantan objekat (što sasvim ima smisla, jer stek *nikada* ne mijenja sadžaj objekta kojeg čuva). Drugi, mnogo bitniji razlog zbog kojeg formalni parametar “element” nije referenca na konstantni objekat leži u činjenici da na taj način stvarni parametar mora biti l-vrijednost (npr. neka promjenljiva) a ne proizvoljan izraz, s obzirom da se referenca na nekonstantni objekat ne može vezati za privremene objekte koji nastaju kao rezultat izračunavanja izraza. Međutim, zbog čega smo uveli ovo ograničenje? Prepostavimo da smo dozvolili da se kao stvarni argument upotrijebi proizvoljan izraz. Tada bi se na steku pohranila adresa privremenog objekta koji sadrži izračunati izraz. Međutim, ovaj privremeni objekat se automatski uništava čim se završi kompletno izvršavanje izraza unutar kojeg je stvoren, tako da će se nakon toga na steku čuvati viseći pokazivač! Ilustrirajmo ovo na konkretnom primjeru. Prepostavimo da konstruktor klase “Student” prima kao parametre ime i prezime studenta, kao i broj indeksa. Tada bi, ukoliko bi parametar metode “Stavi” bila referenca na konstantni objekat, sljedeća naredba bila sasvim sintaksno ispravna:

```
s.Stavi(Student("Pero Perić", 1234));
```

Posljedice ove naredbe mogle bi biti fatalne. Poziv “Student("Pero Perić", 1234)” kreira privremeni propisno inicijalizirani bezimeni objekat tipa “Student” koji se dalje proslijedi metodi “Stavi”, koja uzima njegovu adresu i upisuje je na stek. Međutim, taj privremeni

objekat prestaje postojati odmah po završetku prikazane naredbe, nakon čega će se na steku nalaziti viseći pokazivač. Deklariranjem formalnog parametra metode “*Stavi*” kao reference na nekonstantni objekat, ovakve konstrukcije nisu dozvoljene. Naime, metodi “*Stavi*” će se kao parametri moći proslijediti samo već postojeći objekti, koji će nastaviti svoje postojanje i nakon poziva metode “*Stavi*”. Iz ovoga treba izvući sljedeću pouku: *nikada i ni po koju cijenu ne smijemo u kontejnerskim strukturama podataka čuvati adrese privremenih objekata!*

Obratimo pažnju na još jedan detalj. Metoda “*Skini*” prije nego što vrati rezultat, vrši dereferenciranje pokazivača (da bi zaista vratila objekat na koji pokazivač pokazuje), i vraća kao rezultat *konstantnu referencu na taj objekat*. Na taj način izbjegavamo suvišno kreiranje privremenog objekta koji će biti vraćen kao rezultat iz funkcije i kopiranje dereferenciranog pokazivača u privremeni objekat. Naime, pošto pokazivač koji dereferenciramo pokazuje na konkretan objekat *koji najvjerovaljnije postoji* (jer takvog smo ga stavili na stek, osim ako ga nismo u međuvremenu uništili, što svakako ne bismo trebali raditi), prosto vratiti kao rezultat *referencu na njega*. Kvalifikator “**const**” obezbjeđuje da se tako vraćena vrijednost neće moći koristiti sa lijeve strane operatora dodjele.

U jeziku C++ se savjetuje da se umjesto “zločestih” pokazivača koriste “manje zločeste” reference gdje god je to moguće. Time se, pored neznatno drugačije sintakse (poneka mrska zvjezdica manje), smanjuje mogućnost grešaka u implementaciji, jer je sa referencama teže “zabrljati” nego sa pokazivačima. Stoga bismo, u skladu sa modernim trendovima u C++ programiranju, u čvorovima klase “*IndirektniStek*” umjesto pokazivača na objekte trebali čuvati *reference na objekte*. Ovakva izvedba klase “*IndirektniStek*” izgledala bi ovako:

```
template <typename Tip>
class Stek {
    struct Cvor {
        Tip &ref;
        Cvor *veza;
        Cvor(Tip &ref, Cvor *veza) : ref(ref), veza(veza) {}
    };
    Cvor *gdje_je_vrh;
public:
    IndirektniStek() : gdje_je_vrh(0) {}
    bool Prazan() const { return gdje_je_vrh == 0; }
    void Stavi(Tip &element) {
        gdje_je_vrh = new Cvor(element, gdje_je_vrh);
    }
    const Tip &Skini();
};

template <typename Tip>
const Tip &IndirektniStek<Tip>::Skini() {
    if(gdje_je_vrh == 0) throw "Stek je prazan!\n";
    Tip &ref = gdje_je_vrh->ref;
    Cvor *prethodni = gdje_je_vrh->veza;
    delete gdje_je_vrh;
    gdje_je_vrh = prethodni;
    return ref;
}
```

Verzije klase “`IndirektniStek`” koje čuvaju pokazivače odnosno reference na objekte funkcionalno su potpuno identične, i na programeru je da izabere stil koji mu se više sviđa. Međutim, ove dvije verzije ipak nisu funkcionalno identične sa klasom “`Stek`” čiji čvorovi čuvaju čitave kopije objekata koji se stavljuju na stek. Naime, pretpostavimo da smo “gurnuli nekog studenta na stek” naredbom poput

```
s.Stavi(neki_student);
```

i da smo nakon toga, a prije “skidanja studenta sa steka”, promijenili sadržaj promjenljive “`neki_student`”. Prilikom skidanja sa steka, u slučaju kada stek čuva čitavu kopiju instance klase “`Student`”, kao rezultat će biti vraćena vrijednost objekta “`neki_student`” kakva je bila u trenutku smještanja na stek (jer stek čuva kopiju svih informacija iz objekta “`neki_student`” kakve su bile u tom trenutku). Međutim, u slučaju kad stek čuva samo pokazivače ili reference na studente koji se smještaju na stek, prilikom skidanja sa steka biće vraćena *modificirana* vrijednost promjenljive “`neki_student`”, jer se na steku nije ni čuval njen sadržaj, već samo njena *adresa* (upakovana u formu pokazivača ili reference). Još gora situacija nastaje ukoliko iz bilo kojeg razloga promjenljiva “`neki_student`” prestane postojati prije trenutka skidanja odgovarajućeg elementa sa steka, recimo zbog izlaska iz vidokruga unutar kojeg je promjenljiva “`neki_student`” definirana. U tom slučaju, kao rezultat će biti vraćen fantomski objekat, odnosno dereferencirani višeći pokazivač! Ovo je još jedan od primjera problema vlasništva, na koji smo u više navrata ukazivali (naime, ovakav stek nije vlasnik objekata koji su na njemu smješteni, pa ne može utjecati na to što se sa njima dešava za vrijeme dok se formalno “čuvaju” na steku). U brojnim primjenama se sadržaj promjenljivih poput “`neki_student`” neće mijenjati između trenutka smještanja na stek i trenutka skidanja sa steka, tako da korištenje steka koji čuva samo pokazivače ili reference može zadovoljiti. Vidimo da je potrebno dobro razmisiliti prije nego što donešemo odluku da li ćemo koristiti stek koji čuva samo pokazivače (ili reference) na objekte, ili čitave kopije objekata. Ukoliko se ipak odlučimo za korištenje pokazivača, možda je najbolje uopće ne koristiti klase kao što je klasa “`IndirektniStek`”, već koristiti klasu “`Stek`” (ili, još bolje, standardnu klasu “`stack`”) kojoj ćemo eksplicitno kao tip podataka koji se smještaju na stek deklarirati pokazivački tip (kao što smo na početku ilustrirali), i eksplicitno koristiti adresni operator i dereferenciranje. Na taj način, program jasnije odražava samu namjeru programera, i smanjuje se mogućnost pogrešne interpretacije.

Pored steka, red predstavlja izvrsnog kandidata za dinamičku implementaciju. Podsjetimo se da za razliku od steka koji predstavlja apstraktну strukturu podataka zasnovanu na LIFO principu, red predstavlja apstraktну strukturu podataka zasnovanu na FIFO (First In First Out) principu, odnosno podatak koji je prvi ušao u red, prvi izlazi iz reda. Čitatelju ili čitateljki koji su shvatili način na koji radi dinamička implementacija reda, ne bi trebalo da predstavlja nikakav problem da samostalno kreiraju generičku klasu “`Red`” koja realizira dinamičku implementaciju reda. Zbog činjenice da se u red podaci dodaju na jedan kraj, a skidaju sa drugog kraja (tj. početka), vezu između čvorova je bolje ostvariti tako da veze pokazuju na *sljedeći* čvor a ne na prethodni. Kako se ovakvo vezivanje može ostvariti, najbolje je ilustrirano u konstruktoru kopije za klasu “`Stek`”. Naravno, pri tome je potrebno da klasa

“`Red`” čuva pokazivač na *prvi čvor* u lancu. Na taj način, vađenje elemenata iz reda principijelno je ekvivalentno skidanju elemenata sa steka. Da bi se olakšalo ubacivanje elemenata u red, pametno je čuvati i pokazivač na *posljednji čvor* (u suprotnom bismo morali petljom proći kroz čitav lanac počev od prvog čvora da nađemo poziciju posljednjeg čvora kojeg trebamo povezati sa novokreiranim čvorom koji sadrži element koji ubacujemo). Interesantno je da se konstruktor kopije i operator dodjele za klasu “`Red`” mogu izvesti znatno jednostavnije nego za klasu “`Stek`”, s obzirom da kod kreiranja kopije čvorovima izvornog reda možemo pristupati tačno onim redoslijedom kojim trebamo kreirati čvorove kopije reda, a ne u obrnutom poretku (kao kod steka). Na ovom mjestu nećemo davati dinamičku implementaciju klase `Red`, nego ćemo to ostaviti čitatelju ili čitateljici kao korisnu vježbu. Kao što je već rečeno, ovo ne bi trebao da bude nikakav problem, pogotovo onima koji su u potpunosti shvatili kako radi konstruktor kopije klase “`Stek`”. Također, oni kojima implementacija ove klase bude predstavljala problem, mnoge ideje mogu preuzeti iz implementacije klase “`Lista`”, koja će biti objašnjena nešto kasnije u ovom poglavlju.

S obzirom da je red također veoma korisna struktura podataka, on je također implementiran kao sastavni dio standardne biblioteke jezika C++. Odgovarajuća generička klasa koja implementira red zove se “`queue`”, a nalazi se u istoimenoj biblioteci. Njene metode se zovu istovjetno kao i kod klase “`stack`”, odnosno “`empty`”, “`push`”, “`top`” i “`pop`”, čija je funkcionalnost ista kao kod klase “`stack`”, samo što se vrh reda nalazi sa suprotne strane u odnosu na stek. Stoga će sljedeća sekvenca instrukcija

```
queue<int> q;
q.push(5);
q.push(2);
q.push(7);
q.push(6);
q.push(3);
while (!q.empty()) {
    cout << q.top() << endl;
    q.pop();
}
```

ispisati na ekran slijed brojeva 5, 2, 7, 6 i 3 (svaki u novom redu). Naime, metoda “`top`” dohvata element koji je *prvi* stavljen u red (a ne *posljednji*, kao u slučaju steka). Također, metoda “`pop`” uklanja element koji je *prvi* stavljen u red.

Stek i red spadaju u veoma važne strukture podataka, koje se mnogo koriste u raznim algoritmima. Međutim, obje ove strukture podataka su prilično ograničene, s obzirom da je u njima dodavanje elemenata moguće samo na kraj, a uzimanje podataka je također moguće samo sa kraja (jednog ili drugog, zavisno od toga radi li se o steku ili redu). U primjenama u kojima se koriste stekovi ili redovi, oni se koriste upravo na takav način, tako da to ne predstavlja bitno ograničenje (npr. ukoliko nam treba da pristupamo elementu koji *nije* posljednji element koji smo stavili u neku strukturu podataka, to zapravo govori da nam ustvari i nije potreban stek, nego neka druga struktura podataka koja to omogućava). S druge

strane, često su nam potrebne i fleksibilnije dinamičke strukture podataka. Stek i red, na način kako su implementirani ovdje, predstavljaju specijalan slučaj dinamičkih struktura podataka koje se nazivaju *jednostruko povezane liste* (engl. *single linked lists*), s obzirom da se sastoje od niza čvorova od kojih je svaki čvor povezan sa jednim susjednim čvorom. Moguće je napraviti i znatno fleksibilniju jednostruko povezanu listu, koja omogućava dodavanje novih elemenata na *proizvoljno mjesto* u listi, kao i pristup elementima na proizvoljnoj poziciji u listi. U nastavku ćemo razviti jednu verziju generičke klase koja pruža upravo ovakve mogućnosti. Deklaracija ove klase, koju ćemo nazvati prosto “Lista”, izgleda ovako:

```
template <typename Tip>
class Lista {
    struct Cvor {
        Tip element;
        Cvor *veza;
        Cvor(const Tip &element, Cvor *veza)
            : element(element), veza(veza) {}
    };
    Cvor *pocetak, *kraj, *aktuuelni;
    int velicina, aktuelni_indeks;
public:
    Lista() : pocetak(0), kraj(0), velicina(0) {}
    Lista(const Lista &lista);
    ~Lista();
    Lista &operator =(const Lista &lista);
    bool Prazna() const { return velicina == 0; }
    int Duzina() const { return velicina; }
    void DodajNaPocetak(const Tip &element);
    void DodajNaKraj(const Tip &element);
    void Umetni(int indeks, const Tip &element);
    void Izbaci(int indeks);
    Tip &operator [](int indeks);
    void Ponavljam(void(*akcija) (Tip &));
};
```

Ostavimo za sada implementacione detalje po strani, i pogledajmo samo šta sadrži interfejs klase. Na prvom mjestu, tu su konstruktor, konstruktor kopije, destruktur i prekopljeni operator dodjele, koje treba da posjeduje svaka pristojna klasa koja koristi dinamičku alokaciju memorije. Dalje, tu su metode “Prazna” i “Duzina”, koje respektivno ispituju da li je lista prazna, odnosno koliko lista sadrži elemenata (klase “Stek” i “Red” nisu posjedovale metodu “Duzina”, zbog činjenice da u primjenama u kojima se koriste stekovi i redovi ne treba znati koliko u nekom trenutku stek odnosno red sadrži elemenata). Metode “DodajNaPocetak” i “DodajNaKraj” dodaju novi element na početak odnosno kraj liste, dok metoda “Umetni” umeće element u listu na proizvoljnu poziciju. Metoda “Izbaci” izbacuje iz liste element na navedenoj poziciji. Tu je i prekopljeni operator indeksiranja “[]” koji će omogućiti da proizvoljnim elementima liste pristupamo na isti način kao da se radi o elementima niza. Konačno, predviđena je i interesantna metoda “Ponavljam” koja obavlja akciju definiranu pokazivačem na funkciju koji joj je prenesen kao parametar nad svakim elementom liste. Način upotrebe ove metode biće objašnjen kasnije.

Predimo sada na implementacione detalje. Rad klase opisuju atributi “pocetak”, “kraj” i “velicina” koji respektivno sadrže pokazivače na početak odnosno kraj liste, kao i broj elemenata u listi, te attribute “aktuuelni” i “aktuuelni_indeks”, čija će uloga biti uskoro razjašnjena, a pomoću kojih se postižu izvjesni trikovi koji bitno povećavaju efikasnost klase

(vidjećemo da se u načelu sve moglo postići samo pomoću atributa "pocetak", ali uz gubitak efikasnosti). Veze između čvorova ćemo realizirati tako da svaki čvor sadrži pokazivač na sljedeći čvor. Stoga bi se dodavanje elementa na početak liste principijelno moglo realizirati na isti način kao i stavljanje elemenata na stek. Tako bismo i radili da je atribut "pocetak" jedini atribut koji opisuje rad liste. Međutim, ukoliko dodajemo čvor na početak prazne liste, kreirani čvor je ujedno i prvi i posljednji, tako da je u tom slučaju potrebno također inicijalizirati pokazivač "kraj" da pokazuje na isti element (istom prilikom je potrebno inicijalizirati i atribute "aktuuelni" i "aktuuelni_indeks", čija će uloga postati jasna kasnije). Također je pri svakom dodavanju novog čvora potrebno povećati atribut "velicina" za 1. Stoga bi metoda "DodajNaPocetak" mogla izgledati ovako:

```
template <typename Tip>
void Lista<Tip>::DodajNaPocetak(const Tip &element) {
    pocetak = new Cvor(element, pocetak);
    if(velicina == 0) {
        aktuelni = kraj = pocetak;
        aktuelni_indeks = 0;
    }
    else aktuelni_indeks++;
    velicina++;
}
```

S obzirom da u atributu "velicina" vodimo evidenciju o broju čvorova u listi, implementacija metode "Duzina" je trivijalna, i izvedena je unutar deklaracije klase. Interesantno je da smo metodu "Duzina" mogli napisati i bez uvođenja atributa "velicina", tako što bismo prosto krenuli od pokazivača "pocetak" i prebrojali koliko čvorova ima u listi, prateći veze sve dok ne dođemo do čvora iza kojeg ne slijedi niti jedan čvor. Konkretnije, metodu "Duzina" mogli smo napisati ovako:

```
template <typename Tip>
int Lista<Tip>::Duzina() {
    int brojac(0);
    for(Cvor *pok = pocetak; pok != 0; pok = pok->Veza) brojac++;
    return Brojac;
}
```

Međutim, nije ni potrebno govoriti da je mnogo efikasnije voditi informaciju o broju čvorova u nekom atributu nego ih svaki put brojati. Implementacija metode "Prazna" je također trivijalna, a bila bi trivijalna i da nismo uveli atribut "velicina" (lista je prazna ako je pokazivač "pocetak" jednak nuli).

Dodavanje novog elementa na kraj liste u slučaju prazne liste istovjetno je dodavanju elementa na njen početak. U svim ostalim slučajevima, potrebno je kreirati novi čvor, povezati ga sa posljednjim čvorom u listi, i proglašiti novokreirani čvor za posljednji. Na taj način, implementacija metode "DodajNaKraj" mogla bi izgledati ovako:

```
template <typename Tip>
void Lista<Tip>::DodajNaKraj(const Tip &element) {
    if(velicina == 0) DodajNaPocetak(element);
    else {
        kraj = kraj->veza = new Cvor(element, 0);
        velicina++;
    }
}
```

```
}
```

Po cijenu smanjene efikasnosti, mogli smo proći i bez pokazivača “*kraj*”, jer je kraj liste moguće pronaći od početka i prateći veze između čvorova. U tom slučaju, implementacija metode “*DodajNaKraj*” mogla bi izgledati ovako:

```
template <typename Tip>
void Lista::DodajNaKraj(const Tip &element) {
    if(velicina == 0) DodajNaPocetak(element);
    else {
        Cvor *pok;
        for(pok = pocetak; pok->veza != 0; pok = pok->veza);
        pok->veza = new Cvor(element, 0);
        velicina++;
    }
}
```

Uz prepostavku da imamo metodu “*Izbaci*” koja uklanja element iz liste, destruktor je trivijalan. Dovoljno je izbacivati jedan po jedan element iz liste, dok se lista ne isprazni. Najlakše je stalno izbacivati prvi element (tj. element sa indeksom 0 ukoliko usvojimo konvenciju da indeksiranje elemenata počinje od nule, kao u slučaju običnih nizova):

```
template <typename Tip>
Lista<Tip>::~Lista() {
    while(!Prazna()) Izbaci(0);
}
```

S obzirom da imamo napisanu metodu “*DodajNaKraj*”, konstruktor kopije i izvedba preklopljenog operatora dodjele su također jednostavnji. Potrebno je proći kroz cijelu izvornu listu i svaki čvor dodati na kraj odredišne liste (u slučaju dodjele, prethodno je potrebno obrisati postojeću izvornu listu):

```
template <typename Tip>
Lista<Tip>::Lista(const Lista &lista) : pocetak(0), kraj(0),
    velicina(0) {
    for(Cvor *pok = lista.pocetak; pok != 0; pok = pok->veza)
        DodajNaKraj(pok->element);
}

template <typename Tip>
Lista<Tip> &Lista<Tip>::operator =(const Lista<Tip> &lista) {
    if(&lista == this) return *this;
    while(!Prazna()) Izbaci(0);
    for(Cvor *pok = lista.pocetak; pok != 0; pok = pok->veza)
        DodajNaKraj(pok->element);
    return *this;
}
```

Implementaciju operatorske funkcije za preklapanje operatatora indeksiranja “[]” ćemo objasniti prije implementacije metoda “*Umetni*” i “*Izbaci*”, jer će se ove dvije metode oslanjati na implementaciju ove operatorske funkcije. Najjednostavniji način da se realizira ova operatorska funkcija je da krećući od početka liste prateći veze pronađemo čvor sa zadanim rednim brojem, i da iz njega očitamo traženi element. Primijetimo da je potrebno

vratiti kao rezultat *referencu na nadjeni element*, jer će jedino tako biti moguće da se rezultat indeksiranja nađe sa lijeve strane operatora dodjele, kao što je moguće u slučaju običnih nizova:

```
template <typename Tip>
Tip &Lista<Tip>::operator [](int indeks) {
    if(indeks < 0 || indeks >= velicina) throw "Indeks izvan opsega!\n";
    Cvor *pok = pocetak;
    for(int i = 0; i < indeks; i++) pok = pok->veza;
    return pok->element;
}
```

Međutim, moguće je postići mnogo veću efikasnost. Zamislimo, na primjer, da smo prvo trebali da pristupimo 50-tom elementu liste, a nakon toga 52-gom elementu. Uz prethodnu implementaciju, prilikom pristupa 52-gom elementu ponovo bismo potragu za čvorom koji sadrži ovaj element krenuli od početka liste, iako se ovaj čvor nalazi svega dva čvora ispred 50-tog čvora kojem smo maločas pristupali! Ovim se nameće ideja da je pri svakom pristupu nekom čvoru pametno čuvati njegovu adresu i redni broj u nekim atributima. U slučaju da je indeks čvora kojem želimo da pristupimo veći od indeksa posljednjeg čvora kojem smo pristupali (tj. ako se traženi čvor nalazi *ispred* posljednjeg čvora kojem smo pristupali), potragu za njegovom lokacijom možemo započeti upravo od posljednjeg čvora kojem smo pristupali, a ne od početka liste. Međutim, u slučaju da je indeks čvora kojem želimo da pristupamo manji od indeksa posljednjeg čvora kojem smo pristupali, potraga se mora započeti od početka liste (jer nema načina da na osnovu adrese čvora saznamo adresu njegovog prethodnika). Stoga su uvedeni već pomenuti atributi “*aktuuelni*” i “*aktuuelni_indeks*” koji redom čuvaju adresu i redni broj posljednjeg čvora kojem se pristupalo. Uvođenjem ovih atributa, implementacija operatorske funkcije za operator “[]” mogla bi izgledati ovako:

```
template <typename Tip>
Tip &Lista<Tip>::operator [](int indeks) {
    if(indeks < 0 || indeks >= velicina) throw "Indeks izvan opsega!\n";
    if(indeks < aktuelni_indeks) {
        aktuelni_indeks = 0; aktuelni = pocetak;
    }
    for(; aktuelni_indeks < indeks; aktuelni_indeks++)
        aktuelni = aktuelni->veza;
    return aktuelni->element;
}
```

Ovim trikom se zaista može mnogo dobiti na efikasnosti, naročito u situacijama kada elementima liste često pristupamo u rastućem redoslijedu indeksa.

Umetanje novih elemenata u listu je naročito interesantno. Umetanje novog elementa na početak odnosno na kraj liste svodi se na već napisane metode “*DodajNaPocetak*” odnosno “*DodajNaKraj*”. Međutim, posebno je interesantno razmotriti umetanje elemenata na proizvoljno mjesto unutar liste koje nije niti početak liste niti njen kraj. Poznato je da je ubacivanje elemenata usred niza veoma neefikasan postupak, jer je prethodno sve elemente niza koji slijede iza pozicije na koju želimo da ubacimo novi element potrebno pomjeriti za jedno mjesto naviše, da bi se stvorilo prazno mjesto za element koji želimo da ubacimo. Ovim se troši mnogo vremena, pogotovo ukoliko je potrebno pomjeriti mnogo elemenata niza.

Međutim, ubacivanje elemenata unutar liste može se izvesti znatno efikasnije, uz mnogo manje trošenje vremena. Naime, dovoljno je kreirati novi čvor koji sadrži element koji umećemo, a zatim izvršiti uvezivanje pokazivača tako da novokreirani čvor logički dođe na svoje mjesto. Sve ovo se može izvesti veoma efikasno. Posmatrajmo, na primjer, listu koja sadrži brojeve 3, 6, 7, 2 i 5, u tom poretku. Raspored čvorova u ovoj listi možemo prikazati sljedećom slikom:

Prepostavimo dalje da je između drugog i trećeg elementa potrebno ubaciti novi element, čija je vrijednost 8. Nakon kreiranja novog čvora, dovoljno je povezati pokazivače tako da dobijemo situaciju kao na sljedećoj slici:

Vidimo da nikakva premještanja elemenata u memoriji nisu potrebna (zahvaljujući činjenici da je redoslijed elemenata definiran vezama između čvorova, a ne njihovim fizičkim rasporedom). Na osnovu ove ideje, implementacija metode `Umetni` mogla bi se izvesti ovako (parametar `Indeks` predstavlja redni broj elementa *ispred* kojeg ubacujemo novi element):

```
template <typename Tip>
void Lista<Tip>::Umetni(int indeks, const Tip &element) {
    if(indeks == 0) DodajNaPocetak(element);
    else if(indeks == velicina) DodajNaKraj(element);
    else {
        operator [](indeks - 1);
        aktuelni->veza = new Cvor(element, aktuelni->veza);
        velicina++;
    }
}
```

Ovdje smo eksplisitno pozvali operatorsku funkciju za operator “[]” sa ciljem da nam pronađe adresu čvora koji prethodi čvoru koji umećemo (nađena adresa biće zapamćena u atributu “`aktuelni`”).

Brisanje elemenata se u slučaju listi može također izvesti mnogo efikasnije nego u slučaju nizova. Da bismo uklonili neki element iz niza, potrebno je sve elemente niza koji se nalaze iza elementa koji izbacujemo pomjeriti za jedno mjesto unazad. Međutim, u slučaju liste, dovoljna je mala igra sa pokazivačima koji povezuju čvorove. Tako, na primjer, da bismo iz liste čiji su elementi 3, 6, 7, 2 i 5 izbacili četvrti element, dovoljno je prepraviti pokazivače tako da se dobije sljedeća memorijska slika:

Četvrti čvor na ovaj način više nije unutar “lanca”, pa ga slobodno možemo obrisati, da ne troši memoriju. Vidimo da ni ovdje nije potrebno vršiti nikakvo premještanje elemenata u memoriji, tako da je i operacija brisanja veoma efikasna. Ova ideja iskorištena je u realizaciji metode “Izbaci”, koja kao parametar zahtijeva redni broj čvora koji se uklanja iz liste. Implementacija je nešto složenija, zbog činjenice da treba razlikovati tri slučaja. Naime, nije teško zaključiti da se postupak brisanja čvora koji se nalazi na samom početku odnosno na samom kraju liste razlikuje od postupka brisanja čvora koji nije granični čvor (tj. nije niti prvi niti posljednji čvor). Bez obzira na sve, nije teško shvatiti kako ova metoda radi:

```
template <typename Tip>
void Lista<Tip>::Izbaci(int indeks) {
    if(velicina == 0) throw "Lista je prazna!\n";
    velicina--;
    Cvor *za_brisanje;
    if(indeks == 0) {
        za_brisanje = pocetak;
        aktuelni = pocetak = za_brisanje->veza;
        aktuelni_indeks = 0;
    }
    else {
        operator [](indeks - 1);
        za_brisanje = aktuelni->veza;
        aktuelni->veza = za_brisanje->veza;
        if(indeks == velicina) kraj = aktuelni;
    }
    delete za_brisanje;
}
```

Ostala je još implementacija metode “Ponavljam”. Ova metoda prosto prolazi kroz čitavu listu, i na svaki element primjenjuje funkciju zadalu parametrom “Akcija”:

```
template <typename Tip>
void Lista<Tip>::Ponavljam(void(*akcija)(double &)) {
    for(Cvor *pok = pocetak; pok != 0; pok = pok->veza)
        akcija(pok->element);
}
```

Da bismo vidjeli kako se ova metoda može korisno upotrijebiti. Često je potrebno obaviti istu akciju nad svim elementima liste zaredom. Pretpostavimo, na primjer, da želimo da ispišemo sve elemente u listi realnih brojeva “lista” razdvojene razmacima. Naravno, jedna mogućnost je da koristimo for petlju i preopterećeni operator indeksiranja “[]” kao u slučaju da koristimo niz:

```
for(int i = 0; i < lista.Duzina(); i++) cout << lista[i] << " ";
```

Međutim, druga mogućnost je da definiramo pomoćnu funkciju (nazovimo je npr. “Ispisi”) koja ispisuje vrijednost svog parametra, na primjer

```
void Ispisi(double &element) {
    cout << element << " ";
```

a zatim primijenimo metodu “Ponavljam” proslijedući joj funkciju “Ispisi” kao parametar:

```
lista.Ponavljam(Ispisi);
```

Ovakav pristup je mnogo efektniji, jer se akcija izvodi neposredno nad elementima liste kojima se pristupa sekvensijalno, prateći pokazivače, bez potrebe za pozivanjem operatorske funkcije za operator “[]”, što svakako troši vrijeme, bez obzira što je ova operatorska funkcija znatno optimizirana pamćenjem pozicije i indeksa posljednjeg elementa kojem smo pristupali. Pored toga, mnogo stvari sa listom bi se moglo raditi pomoću metode “Ponavljam” čak i da u listi nismo uopće implementirali operator “[]”. Može se postaviti pitanje zbog čega funkcija koja se prenosi kao parametar u metodu “Ponavljam” svoj parametar prihvata po referenci (i to na nekonstantni objekat). Ovo smo uradili da bismo omogućili da metoda “Ponavljam” eventualno obavi *izmjene* nad elementima liste. Zamislimo, na primjer, da želimo da udvostručimo sve elemente u listi. Jedno moguće rješenje je svakako korištenje petlje:

```
for(int i = 0; i < lista.Duzina(); i++) lista[i] *= 2;
```

Međutim, alternativno rješenje je da definiramo pomoćnu funkciju (nazovimo je npr. “Dupliraj”) koja udvostručava vrijednost svog parametra, na primjer

```
void Dupliraj(double &element) {
    element *= 2;
}
```

a zatim primijenimo metodu “Ponavljam” prosljeđujući joj funkciju “Dupliraj” kao parametar:

```
lista.Ponavljam(Dupliraj);
```

Metode poput “Ponavljam” koje prolaze kroz sve elemente neke strukture obavljajući neku akciju nad njima obično se nazivaju *iteratorske metode* ili *metode iteratori*.

Pored osobine da ne moramo imati ikakve apriorne informacije o broju elemenata koje treba smjestiti, lijepa osobina liste kao strukture podataka je činjenica da je umetanje elemenata na proizvoljno mjesto kao i izbacivanje elemenata iz liste sa proizvoljnog mesta mnogo efikasnije nego pri korištenju nizova. Radi ilustracije čemo prikazati primjer sortiranja brojeva koji se unose sa tastature “u hodu”, odnosno njihovim smještanjem u listu na pravo mjesto odmah po njihovom unosu. Prikazani isječak programa zahtijeva unos skupine brojeva, pri čemu unos nule prekida unos. Nakon unosa svakog broja, traži se mjesto u listi na koji bi on trebao da bude ubačen, nakon čega se poziva metoda “Umetni” da se broj ubaci na pravo mjesto. Ovaj postupak konceptualno je sličan načinu kako bi čovjek sortirao spisak podataka od kojih je svaki podatak zapisan na posebnom listu papira (u suštini, ovo je zapravo varijanta *sortiranja umetanjem* o kojem smo govorili u poglavljju o sortiranju):

```
Lista<double> lista;
for(;;) {
    double broj;
    cout << "Unesi broj (0 za kraj): ";
    cin >> broj;
    if(broj == 0) break;
    for(int i = 0; i < lista.Duzina() && lista[i] < broj; i++);
    lista.Umetni(i, broj);
}
cout << "Sortirani spisak brojeva glasi: ";
lista.Ponavljam(Ispisi);
```

Lista je jedna od najkorisnijih i najviše korištenih struktura podataka u brojnim programerskim problemima. Kako je lista mnogo općenitija struktura podataka i od steka i od reda (koji se, kao što smo vidjeli, mogu smatrati kao njene podvarijante), klase "Stek" i "Red" je, u slučaju da nam problem koji rješavamo zahtijeva njihovu upotrebu, moguće implementirati veoma jednostavno pomoću klase "Lista". Jedna od mogućih implementacija mogla bi izgledati ovako:

```
template <typename Tip>
class Stek {
    Lista<Tip> lista;
public:
    bool Prazan() { return lista.Prazna(); }
    void Stavi(const Tip &element) { lista.DodajNaPocetak(element); }
    Tip Skini() {
        double element = lista[0]; lista.Izbaci(0);
        return element;
    }
};

template <typename Tip>
class Red {
    Lista<Tip> lista;
public:
    bool Prazan() { return lista.Prazna(); }
    void Ubaci(const Tip &element) { lista.DodajNaKraj(element); }
    Tip Izvadi() {
        double element = lista[0]; lista.Izbaci(0);
        return element;
    }
};
```

Ove implementacije su zaista u toj mjeri jednostavne da ne traže nikakva dopunska objašnjenja.

Najveći nedostatak jednostrukog povezane liste je činjenica da je veza između čvorova ostvarena samo u jednom smjeru (od prethodnom ka sljedećem čvoru). Posljedica je da pristup elementima liste može biti veoma neefikasan ukoliko se elementima ne pristupa u rastućem poretku indeksa. Na primjer, ukoliko prvo trebamo da pristupimo 100-tom elementu, a nakon toga 99-tom elementu, da bismo pronašli 99-ti element moramo krenuti od početka liste, iako on neposredno prethodi 100-tom elementu kojem smo upravo pristupali! Ovaj nedostatak

može se izbjjeći tako što ćemo proširiti čvorove liste tako da sadrže *dva pokazivača*, na *prethodni* i na *sljedeći* čvor. Dinamička struktura podataka zasnovana na ovakvim čvorovima naziva se *dvostruko povezana lista* ili *dvostruko spregnuta lista* (engl. *double linked list*). Sljedeća slika ilustrira kako bi mogla izgledati organizacija čvorova u dvostruko spregnutoj listi:

Dvostruko povezana lista je mnogo efikasnija dinamička struktura podataka od jednostruko povezane liste. Čitaocima koji su shvatili implementaciju jednostruko povezane liste ne bi trebao da bude nikakav problem da sastave implementaciju dvostruko povezane liste.