Predavanje 1

Ovi materijali namijenjeni su onima koji su odslušali kurs "Osnove računarstva" na Elektrotehničkom fakultetu u Sarajevu po novom nastavnom programu, kao i svima onima koji već poznaju osnove programiranja u programskom jeziku C, a žele da pređu na programski jezik C++. S obzirom da je programski jezik C++ isprva bio zamišljen kao nadgradnja jezika C, gotovo svi programi pisani u jeziku C ujedno predstavljaju ispravne C++ programe. Ipak, kompatibilnost između C-a i C++-a nije stoprocentna, tako da postoje neke konstrukcije koje su ispravne u jeziku C, a nisu u jeziku C++. Takvi primjeri će biti posebno istaknuti. S druge strane, bitno je napomenuti da mnoge konstrukcije iz jezika C, mada rade u jeziku C++, treba izrazito izbjegavati u jeziku C++, s obzirom da u jeziku C++ postoje mnogo bolji, efikasniji i sigurniji načini da se ostvari isti efekat. Treba shvatiti da je C++ danas ipak posve novi programski jezik, bitno drugačiji od C-a, mada u principu visoko kompatibilan sa njim. Rješenja koja su dobra (a ponekad i jedina moguća) u C-u, u C++-u mogu biti veoma loša. Stoga biti vrlo dobar programer u C-u najčešće ne znači biti dobar C++ programer. Vrijedi čak obrnuto: ispočetka su nabolji C programeri obično katastrofalno loši C++ programeri, sve dok se ne naviknu na filozofiju novog jezika i na novi način razmišljanja. Stoga će uvodna poglavlja uglavnom biti posvećena onim aspektima jezika C koje nije preporučljivo koristiti u C++-u, dok će specifičnosti načina razmišljanja u C++-u doći kasnije.

Počnimo od prvog C programa iz većine udžbenika o C-u – tzv. "Hello world" / "Vozdra raja" programa:

```
#include <stdio.h>

main() {
  printf("Vozdra raja!\n");
  return 0;
}
```

Čak ni ovaj posve jednostavan C program nije ispravan u C++-u prema posljednjem standardu ISO 98 jezika C++, kojeg ćemo se u ovim materijalima pridržavati (bez obzira što će mnogi C++ kompajleri ipak prevesti i izvršiti ovaj program). Naime, u ovom primjeru, funkcija "`main`" nema povratni tip. U C-u se tada podrazumijeva da je povratni tip "**`int`**", dok se u C++-u povratni tip *uvijek mora navoditi*. Dakle, u C++-u bi ispravna bila sljedeća varijanta:

```
#include <stdio.h>

int main() {
  printf("Vozdra raja!\n");
  return 0;
}
```

Ovo je, bar za sada, ispravno C++ rješenje. Međutim, već pojavom sljedećeg C++ standarda, postoji velika mogućnost da se ni ovo rješenje neće smatrati ispravnim. Naime, standardna biblioteka funkcija prilagođena jeziku C, kojoj između ostalog pripada i zaglavlje "`stdio.h`", ne uklapa se dobro u neke moderne koncepte jezika C++. Stoga će većina kompajlera na gornji program dati upozorenje da je upotreba zaglavlja "`stdio.h`" izrazito nepreporučljiva (engl. *deprecated*) u jeziku C++, i da vjerovatno u budućnosti neće biti podržana. Umjesto toga, po ISO 98 C++ standardu, napravljene su nove verzije zaglavlja standardne biblioteke za one elemente jezika C++ koji su naslijeđeni iz jezika C, a koje su napravljene tako da se bolje uklapaju u koncepte jezika C. Stoga, ukoliko je "`pqr.h`" naziv nekog zaglavlja standardne biblioteke jezika C, u C++-u umjesto njega treba koristiti zaglavlje imena "`cpqr`", dakle, bez sufiksa "`.h`" i sa prefiksom "`c`" (npr. "cstdio" umjesto "`stdio.h`", "`cmath`" umjesto "`math.h`", "`cctype`" umjesto "`ctype.h`", itd.). Stoga, potpuno ispravna verzija gornjeg programa u C++-u glasi ovako:

```
#include <cstdio>

int main() {
  std::printf("Vozdra raja!\n");
  return 0;
}
```

Ovdje vidimo još jednu novinu: kvalifikator "`std::`" ispred imena funkcije "`printf`". Ovo je jedna od novina zbog koje su uopće i uvedena nova zaglavlja – tzv. imenici. Imenici su osobina jezika C++ koja je uvedena tek nedavno. Naime, pojavom velikog broja nestandardnih biblioteka različitih proizvođača bilo je nemoguće spriječiti konflikte u imenima koje mogu nastati kada dvije različite biblioteke upotrijebe isto ime za dva različita objekta ili dvije različite funkcije. Na primjer, biblioteka za rad sa bazama podataka može imati funkciju nazvanu "`update`" (za ažuriranje podataka u bazi), dok neki autor biblioteke za rad sa grafikom može odabrati isto ime "`update`" za ažuriranje grafičkog prikaza na ekranu. Zbog toga je odlučeno da se imena (identifikatori) svih objekata i funkcija mogu po potrebi razvrstavati u *imenike* (kao što se datoteke mogu razvrstavati po folderima). Dva različita objekta ili funkcije mogu imati isto ime, pod uvjetom da su definirani u različitim imenicima. Identifikatoru "*ime*" koji se nalazi u imeniku "*imenik*" pristupa se pomoću konstrukcije "*imenik*`::`*ime*". Dalje, standard propisuje da se svi objekti i sve funkcije iz standardne biblioteke jezika C++ moraju nalaziti u imeniku "`std`". Odatle potiče kvalifikacija "`std::printf`". Sličnu kvalifikaciju bismo morali dodati ispred svih drugih objekata i funkcija iz standardne biblioteke (npr. "`std::sqrt`" za funkciju "`sqrt`" iz zaglavlja "`cmath`", koja računa kvadratni korijen).

Stalno navođenje imena imenika može biti naporno, stoga je moguće pomoću ključne riječi "**`using`**" navesti da će se podrazumijevati da se neki identifikator uzima iz navedenog imenika, ukoliko se ime imenika izostavi. Sljedeći primjer pokazuje takav pristup:

```
#include <cstdio>

using std::printf;

int main() {
  printf("Vozdra raja!\n");
  return 0;
}
```

Ovim smo naveli da se podrazumijeva da identifikator "`printf`" uzimamo iz imenika "`std`", ako se drugačije ne kaže eksplicitnim navođenjem imenika ispred identifikatora. Možemo također reći da se neki imenik (u našem primjeru "`std`") podrazumijeva ispred *bilo kojeg identifikatora* ispred kojeg nije eksplicitno naveden neki drugi imenik, na način kao u sljedećem primjeru:

```
#include <cstdio>

using namespace std;

int main() {
  printf("Vozdra raja!\n");
  return 0;
}
```

Ovakvo rješenje ćemo pretežno koristiti u svim primjerima tokom kursa radi jednostavnosti, mada se ovakvo rješenje ne preporučuje, jer na taj način u program "uvozimo" cijeli imenik (ovo je logički ekvivalentno pozicioniranju u neki folder pri radu sa datotekama), čime narušavamo osnovni razlog zbog kojeg su imenici uopće uvedeni. Ipak, razvrstavanje identifikatora po različitim imenicima je tehnika koja se koristi uglavnom pri razvoju vrlo velikih projekata.

Mada su prethodni primjeri dotjerani tako da budu u potpunosti po standardu jezika C++, oni još uvijek nisu "u duhu" jezika C++, zbog korištenja biblioteke "`cstdio`" i njene funkcije "`printf`", koje su isuviše "niskog nivoa" sa aspekta jezika C++, koji je konceptualno jezik višeg nivoa nego C (zapravo,

C++ je "hibridni jezik", u kojem se može programirati na "niskom nivou", na "visokom nivou", u proceduralnom, neobjektnom stilu, kao i u objektno-orijentiranom stilu, pa čak i u nekim apstraktnim stilovima kao što su generički i funkcionalni stil programiranja, za razliku od jezika kao što je Java, koji forsira isključivo objektno-orijentirani stil, i kad treba, i kad ne treba). Slijedi primjer koji radi isto što i svi prethodni primjeri, samo pisan u duhu jezika C++:

```
#include <iostream>

using namespace std;

int main() {
  cout << "Vozdra raja!\n";
  return 0;
}
```

U ovom primjeru je umjesto funkcije "`printf`" upotrijebljen tzv. *objekat izlaznog toka podataka* (engl. *output stream*) "`cout`", koji je povezan sa standardnim izlaznim uređajem (obično ekranom). Ovaj objekat definiran je u biblioteci "`iostream`", i kao i svi objekti iz standardnih biblioteka, također se nalazi u imeniku "`std`" (da nismo "uvezli" imenik "`std`", morali bismo pisati "`std::cout`", inače bi nam kompajler javio da ne poznaje objekat "`cout`", što je vrlo česta greška kod početnika). Znak "<<" predstavlja *operator umetanja* (engl. *insertion*) u izlazni tok, koji pojednostavljeno možemo čitati kao "šalji na". Stoga konstrukcija

```
cout << "Vozdra raja!\n";
```

"umeće" niz znakova (string) `"Vozdra raja\n"` (podsjetimo se da je "\n" znak za prelaz u novi red) u izlazni tok, odnosno "šalje ga" na standardni izlazni uređaj (ekran).

Prvi argument operatora "<<" je obično neki objekat izlaznog toka (postoje i drugi objekti izlaznog toka osim "`cout`", npr. objekti izlaznih tokova vezani sa datotekama), dok drugi argument može biti proizvoljni ispisivi izraz. Na primjer, sljedeće konstrukcije su posve legalne (uz pretpostavku da je uključena biblioteka "`cmath`", u kojoj je deklarirana funkcija "`sqrt`" za računanje kvadratnog korijena):

```
cout << 32 – (17 + 14 * 7) / 3;
cout << 2.56 - 1.8 * sqrt(3.12);
```

Primijetimo da objekat izlaznog toka "`cout`" sam "vodi računa" o tipu podataka koji se "umeću" u tok, tako da nije potrebno eksplicitno specificirati tip podatka koji se ispisuje, kao što bismo morali prilikom korištenja funkcije "`printf`", gdje bismo morali pisati nešto poput:

```
printf("%d", 32 – (17 + 14 * 7) / 3);
printf("%f", 2.56 - 1.8 * sqrt(3.12));
```

Važno je napomenuti da konstrukcija poput "`cout << 2 + 3`" u jeziku C++ *također predstavlja izraz*, kao što je izraz i sama konstrukcija "`2 + 3`". Šta je rezultat ovog izraza? U C++-u rezultat operatora umetanja primjenjen na neki objekat izlaznog toka kao rezultat daje ponovo isti objekat izlaznog toka (odnosno "`cout`" u našem primjeru), što omogućava *nadovezivanje* operatora "<<", kao u sljedećem primjeru:

```
cout << "2 + 3 = " << 2 + 3 << "\n";
```

Naime, ovaj izraz se interpretira kao

```
((cout << "2 + 3 = ") << 2 + 3) << "\n";
```

Drugim riječima, u izlazni tok se prvo umeće string `"2 + 3 = "`. Kao rezultat tog umetanja dobijamo ponovo objekat "`cout`" kojem se šalje vrijednost izraza "`2 + 3`" (odnosno 5). Rezultat tog umetanja je

ponovo objekat "cout", kojem se šalje znak za novi red. Krajnji rezultat je ponovo objekat "cout", ali tu činjenicu možemo ignorirati, s obzirom da jezik C++, slično jeziku C, dozvoljava da se krajnji rezultat ma kakvog izraza ignorira (recimo, i funkcija "printf" daje kao rezultat broj ispisanih znakova, ali se taj rezultat gotovo uvijek ignorira). U duhu jezika C, prethodna naredba napisala bi se ovako:

```
printf("2 + 3 = %d\n", 2 + 3);
```

U zaglavlju "iostream" je definiran i objekat za prelazak u novi red "endl". U logičkom smislu, on je ekvivalentan stringu "\n", mada u izvedbenom smislu postoje značajne razlike (u efektivnom smislu, razlika je što upotreba objekta "endl" uvijek dovodi do pražnjenja spremnika izlaznog toka, što u većini primjena nije bitno). Stoga smo mogli pisati i:

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

Na ovom mjestu nije loše napomenuti da standard ISO 98 jezika C++ predviđa da standardna biblioteka jezika C++ obavezno mora sadržavati biblioteke sa sljedećim zaglavljima:

```
algorithm   bitset      cassert     cctype      cerrno      cfloat
ciso64      climits     clocale     cmath       complex     csetjmp
csignal     cstdarg     cstddef     cstdio      cstdlib     cstring
ctime       cwchar      cwtype      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
```

Pored navedenih 50 standardnih zaglavlja iz standardne biblioteke, mnogi kompajleri 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. Zaglavlja nestandardnih biblioteka gotovo uvijek imaju i dalje nastavak ".h" na imenu, da bi se razlikovala od standardnih biblioteka.

Slično objektu "cout", biblioteka "iostream" definira i *objekat ulaznog toka podataka* (engl. *input stream*) "cin", koji je povezan sa standardnim uređajem za unos (tipično tastaturom). Ovaj objekat se obično koristi zajedno sa *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". Slijedi primjer programa koji zahtijeva unos dva cijela broja sa tastature, a koji zatim ispisuje njihov zbir:

```
#include <iostream>

using namespace std;

int main() {
  int broj_1, broj_2;
  cout << "Unesite prvi broj: ";
  cin >> broj_1;
  cout << "Unesite drugi broj: ";
  cin >> broj_2;
  cout << "Zbir brojeva" << broj_1 << " i " << broj_2
    << " glasi " << broj_1 + broj_2 << endl;
  return 0;
}
```

Isti program, napisan u duhu C-a, mogao bi izgledati recimo ovako:

```
#include <cstdio>

using namespace std;

int main() {
  int broj_1, broj_2;
  printf("Unesite prvi broj: ");
  scanf("%d", &broj_1);
  printf("Unesite drugi broj: ");
  scanf("%d", &broj_2);
  printf("Zbir brojeva %d i %d glasi %d\n", broj_1, broj_2,
    broj_1 + broj_2);
  return 0;
}
```

Primijetimo da nas objekat ulaznog toka također oslobađa potrebe da vodimo računa o tipu podataka koje unosimo, i da eksplicitno prenosimo *adresu odredišne promjenljive* pomoću operatora "&", što moramo raditi pri upotrebi funkcije "`scanf`". Za razliku od operatora umetanja, desni operand kod operatora izdvajanja ne može biti proizvoljan izraz, već samo promjenljiva, ili neki objekat iza kojeg postoji rezervirani prostor u memoriji, kao što je npr. element niza, dereferencirani pokazivač. itd. (takvi objekti nazivaju se *l-vrijednosti*, engl. *l-values*).

Promjenljive se u jeziku C++ deklariraju na isti način kao u jeziku C, i za njihova imenovanja vrijede ista pravila kao u jeziku C, uz iznimku da C++ posjeduje više ključnih riječi od jezika C, tako da postoji veći broj riječi koje ne smiju biti imena promjenljivih (tako, npr. "`friend`" i "`this`" ne mogu biti imena promjenljivih u C++-u, a mogu u C-u, jer si u C++-u "`friend`" i "`this`" ključne riječi). Prema standardu ISO 98 C++-a, ključne riječi u jeziku C++ su sljedeće:

```
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.

Slično kao kod operatora umetanja, rezultat operatora izdvajanja primijenjen nad objektom ulaznog toka daje kao rezultat ponovo sam objekat ulaznog toka, što omogućava ulančavanje i ovog operatora. Recimo, konstrukcija poput

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

ima isto dejstvo kao da smo napisali

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

Ovo je iskorišteno u sljedećem primjeru, u kojem se sa tastature učitavaju tri cijela broja, a zatim ispisuju na ekran, svaki u novom redu:

```
#include <iostream>

using namespace std;

int main() {
  int a, b, c;
  cin >> a >> b >> c;
  cout << a << endl << b << endl << c << endl;
  return 0;
}
```

Na ovom mjestu je bitno ukazati na jednu čestu početničku grešku. Mada nekome može izgledati logična konstrukcija poput

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

ona *neće dati očekivani rezultat* (iz ulaznog toka će biti učitana samo vrijednost promjenljive "a"). Što je najgore, kompajler *neće prijaviti nikakvu grešku* s obzirom da je ova konstrukcija *sintaksno ispravna* u skladu sa općim sintaksnim pravilima jezika C i C++. Ovo je neželjeni propratni efekat načina na koji se u jezicima C i C++ interpretira znak zareza u općem slučaju (u detalje ovdje nećemo ulaziti).

Razmotrimo šta se detaljnije dešava prilikom upotrebe ulaznog toka. Svi znakovi koje korisnik unese sve do pritiska na taster ENTER čuvaju se u spremniku (baferu) ulaznog toka. Međutim, operator izdvajanja vrši izdvajanje iz ulaznog toka samo do prvog razmaka ili prvog znaka koji ne odgovara tipu podataka koji se izdvaja (npr. do prvog znaka koji nije cifra u slučaju kada izdvajamo cjelobrojni podatak). Preostali znakovi i dalje su pohranjeni u spremniku ulaznog toka. Sljedeća upotreba operatora izdvajanja će nastaviti izdvajanje iz niza znakova zapamćenog u spremniku. Tek kada se *ulazni tok isprazni*, odnosno kada se *istroše svi znakovi* pohranjeni u spremniku biće zatražen novi unos sa ulaznog uređaja. Sljedeća slika prikazuje nekoliko mogućih scenarija prilikom izvršavanja prethodnog programa (kurzivom su prikazani podaci koje unosi korisnik nakon što se program pokrene).

```
10  20  30  40(ENTER)
10
20
30
```

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

```
10(ENTER)
20(ENTER)
30  40(ENTER)
10
20
30
```

Ovakvo ponašanje je u nekim slučajevima povoljno, ali u nekim nije. Nekada želimo da smo uvijek sigurni da će operator izdvajanja pročitati "svježe unesene" podatke, a ne neke podatke koji su od ranije preostali u spremniku ulaznog toka. Zbog toga je moguće pozivom funkcije "ignore" nad objektom ulaznog toka "cin" isprazniti ulazni tok (vidjećemo kasnije da u jeziku C++ postoje funkcije koje se ne pozivaju samostalno, nego uvijek nad nekim objektom, pri čemu je sintaksa takvog poziva "*objekat*.*funkcija*(*argumenti*)". Ova funkcija ima dva argumenta, od kojij je prvi cjelobrojnog tipa, a drugi znakovnog tipa (tipa "**char**"). Ona 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 broj_1, broj_2;
cout << "Unesite prvi broj: ";
cin >> broj_1;
cin.ignore(10000, '\n');
cout << "Unesite drugi broj: ";
cin >> broj_2;
```

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.

U slučaju unosa pogrešnih podataka (npr. ukoliko se očekuje broj, a već prvi uneseni znak nije cifra), ulazni tok će dospjeti u tzv. *neispravno stanje*. Da je tok u neispravnom stanju, možemo provjeriti primjenom operatora negacije "`!`" nad objektom toka. Rezultat je tačan ako i samo ako je tok u neispravnom stanju, što možemo iskoristiti kao uvjet unutar naredbe "**`if`**", "**`while`**" ili "**`for`**". Ovo ilustrira sljedeći fragment:

```
int broj;
cout << "Unesite neki broj: ";
cin >> broj;
if(!cin) cout << "Niste unijeli broj!\n";
else cout << "Zaista ste unijeli broj!\n";
```

Kao uvjet unutar naredbi "**`if`**", "**`while`**" i "**`for`**", može se iskoristiti i sam objekat toka, koji se tada interpretira kao tačan ako i samo ako je u ispravnom stanju. Tako se prethodni fragment mogao napisati i ovako:

```
int broj;
cout << "Unesite neki broj: ";
cin >> broj;
if(cin) cout << "Zaista ste unijeli broj!\n";
else cout << "Niste unijeli broj!\n";
```

Kada ulazni tok jednom dospije u neispravnom stanju, on u takvom stanju ostaje i nadalje, i svaki sljedeći pokušaj izdvajanja iz ulaznog toka biće ignoriran. Tok možemo ponovo vratiti u ispravno stanje pozivom funkcije "`clear`" bez parametara nad objektom ulaznog toka. U narednom primjeru ćemo iskoristiti ovu funkciju i "**`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 izbjeći? 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 potvrdan, 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 *propratni efekat uvjeta petlje*! Naime, konstrukcija "`cin >> broj`" sama po sebi predstavlja izraz (sa propratnim efektom unošenja vrijednosti u promjenljivu "`broj`"), 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 tačan ili netačan 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;
```

Znatno je čistije sljedeće rješenje, u kojem je prvo upotrijebljena konstrukcija "**`for`**`(;;)`" koja predstavlja petlju bez prirodnog izlaza, a koja se nasilno prekida pomoću naredbe "**`break`**" u slučaju ispravnog unosa:

```
int broj;
cout << "Unesite broj: ";
for(;;) {
  cin >> broj;
  if(cin) break;
  cout << "Nemojte se šaliti, unesite broj!\n";
  cin.clear();
  cin.ignore(10000, '\n');
}
cout << "U redu, unijeli ste broj " << broj << endl;
```

Već smo rekli da je korištenje objekata ulaznog i izlaznog toka "`cin`" i "`cout`" mnogo jednostavnije i fleksibilnije (kao što ćemo uskoro vidjeti) od korištenja funkcija iz biblioteke "`cstdio`", kao što su "`printf`" i "`scanf`". To ipak ne znači da biblioteku "`cstdio`" treba potpuno zaboraviti. Dva su slučaja u kojima treba koristiti funkcije iz ove biblioteke. Prvo, objekti poput "`cin`" i "`cout`" su, zbog svoje velike fleksibilnosti, veoma masivni, i njihova upotreba osjetno produžuje dužinu generiranog izvršnog koda. Stoga, samo uključenje biblioteke "`iostream`" u program, koja deklarira i definira ove objekte, tipično *produžava dužinu generisanog izvršnog koda za nekoliko desetina do stotina kilobajta* (zavisno od kompajlera). U slučaju da želimo *generiranje kratkog izvršnog koda*, treba zaobići ovu biblioteku i koristiti funkcije iz biblioteke "`cstdio`". Drugo, funkcije iz biblioteke "`cstdio`" su *brže* od ekvivalentnih operacija sa tokovima, tako da je u slučaju kada iz ulaznog toka treba pročitati više desetina hiljada podataka (ili kada treba ispisati više desetina hiljada podataka na izlazni tok) za kratko vrijeme, preporučuje se ne koristiti objekte tokova, nego funkcije iz biblioteke "`cstdio`". U svim ostalim slučajevima, upotreba biblioteke "`cstdio`" se u jeziku C++ ne preporučuje.

Jezik C++ omogućava mnogo veću slobodu u deklaraciji promjenljivih u odnosu na jezik C. Na primjer, razmotrimo sljedeći program, koji nalazi rješenja kvadratne jednačine, vodeći računa o tome da li su ona realna ili kompleksna (pri tome se kompleksna rješenja ispisuju u obliku "*re* + *im*`*i`" odnosno "*re* − *im*`*i`"). U ovom primjeru, deklaracija promjenljivih je izvedena "u duhu" jezika C (podsjetimo se da funkcija "`fabs`" vraća apsolutnu vrijednost realnog broja zadanog kao argument):

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
  double a, b, c, d, x1, x2, re, im;
  cout << "Unesi koeficijente a, b i c kvadratne jednačine:";
  cin >> a >> b >> c;
  d = b * b - 4 * a * c;
  if(d >= 0) {                                   /* realna rješenja */
    x1 = (-b - sqrt(d)) / (2 * a);
    x2 = (-b + sqrt(d)) / (2 * a);
    cout << "x1 = " << x1 << "\nx2 = " << x2 << endl;
  }
  else {                                         /* kompleksna rješenja */
    re = -b / (2 * a);
    im = fabs(sqrt(-d) / (2 * a));
    cout << "x1 = " << re << "+" << im << "*i\n"
      << "x2 = " << re << "-" << im << "*i\n";
  }
  return 0;
}
```

Naime, u jeziku C se sve promjenljive unutar nekog bloka moraju deklarirati isključivo *prije prve izvršne naredbe bloka*. Ovo ograničenje ne postoji u jeziku C++. Čak naprotiv, u C++-u se smatra veoma lošom praksom deklarirati promjenljivu prije nego što se za njom zaista pojavi potreba u programu. Dalje, u C-u se promjenljive prilikom deklaracije mogu inicijalizirati *isključivo konstantnim vrijednostima*, dok se u C++-u za inicijalizaciju mogu upotrijebiti proizvoljni izrazi ispravnog tipa. Slijedi ponovo isti primjer, ovaj put napisan u skladu sa C++ stilom:

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
  double a, b, c;
  cout << "Unesi koeficijente a, b i c kvadratne jednačine:";
  cin >> a >> b >> c;
  double d = b * b - 4 * a * c;
  if(d >= 0) {                                   // realna rješenja
    double x1 = (-b - sqrt(d)) / (2 * a);
    double x2 = (-b + sqrt(d)) / (2 * a);
    cout << "x1 = " << x1 << "\nx2 = " << x2 << endl;
  }
  else {                                         // kompleksna rješenja
    double re = -b / (2 * a);
    double im = fabs(sqrt(-d) / (2 * a));
    cout << "x1 = " << re << "+" << im << "*i\n"
      << "x2 = " << re << "-" << im << "*i\n";
  }
  return 0;
}
```

Treba napomenuti da neki C kompajleri (npr. GNU C) također dozvoljavaju ovakve slobodnije deklaracije promjenljivih, ali to nije po C standardu (doduše, ta mogućnost je ušla u najnoviji standard jezika C, poznat kao C99). U ovom programu, pored slobodnije deklaracije promjenljivih vidimo i novi način pisanja komentara koji je uveden u jeziku C++. Naime, pored načina za pisanje komentara naslijeđenog iz jezika C, po kojem se komentari omeđuju znakovima "`/*`" i "`*/`", u C++-u je komentar moguće započeti znakom "`//`" pri čemu se smatra da je komentar sve iza tog znaka pa do kraja reda u kojem je on napisan. Ovakav način pisanja je jako praktičan za kratke jednolinijske komentare, dok se za dugačke komentare, koji se protežu u nekoliko redova, i dalje može koristiti način naslijeđen iz C-a.

Slično kao u C-u, opseg važenja (vidokrug) svake deklarirane promjenljive, završava se na kraju bloka unutar kojeg je promjenljiva deklarirana. U prethodnom primjeru, to na primjer znači da promjenjive "`x1`" i "`x2`" vrijede samo unutar bloka koji slijedi iza "**`if`**" naredbe, dok je opseg važenja promjenljivih "`re`" i "`im`" ograničen samo na blok koji slijedi iza "**`else`**". Pravilo dobre upotrebe promjenljivih u jeziku C++ nalaže da svaka promjenljiva treba da ima vidokrug koji ne treba da bude veći od opsega u kojem se ta promjenljiva zaista i koristi (tj. da joj vidokrug bude najmanji mogući). Na taj način se sprečava mogućnost da se neka promjenljiva greškom upotrijebi na mjestu gdje to nije bilo planirano.

Radi politike sužavanja opsega važenja promjenljivih na najmanju moguću mjeru, u jeziku C++ je omogućeno da se promjenljive mogu deklarirati unutar *prvog argumenta naredbe* "**`for`**", što u jeziku C naravno nije dozvoljeno. Opseg važenja takvih promjenljivih ograničen je samo na tijelo pripadne "**`for`**" petlje, odnosno svaka takva promjenljiva prestaje postojati čim se petlja završi (tako da je istoimenu promjenljivu moguće kasnije ponovo deklarirati i upotrijebiti za nešto drugo). Na primjer, pogledajmo sljedeći C++ fragment, koji izračunava i ispisuje zbir recipročnih vrijednosti svih brojeva od 1 do 1000:

```
double suma = 0;
for(int i = 1; i <= 1000; i++) suma += 1. / i;
cout << suma;
```

U ovom primjeru, promjenljiva "`i`", deklarirana unutar "**`for`**" petlje, postoji samo dok se petlja ne završi. U C++-u je preporuka da se brojačke promjenljive koje upravljaju radom "**`for`**" petlji deklariraju isključivo na ovakav način. Na taj način se sprečava mogućnost da se brojačka promjenljiva nehotice neispravno upotrijebi ili čak zloupotrijebi izvan tijela petlje.

Obratimo pažnju da smo u prethodnom primjeru pisali "`1.`" umjesto "`1`". Da to nismo uradili, imali bismo pogrešan rezultat. Naime, kako je "`1`" cjelobrojnog tipa, i kako je promjenljiva "`i`" također cjelobrojna, operator "`/`" u izrazu poput "`1 / i`" interpretirao bi se kao *cjelobrojno djeljenje*. Ovako, kako je "`1.`" realnog tipa, operator "`/`" se interpretira kao *klasično dijeljenje*. Ovo je dobar povod da kažemo nešto o *konverziji tipova* u C++-u. C++ podržava skoro sve automatske konverzije tipova koji se dešavaju i u jeziku C (uz rijetke izuzetke koji će biti posebno istaknuti). Na primjer, cjelobrojni izrazi se automatski konvertiraju u realne ukoliko se upotrijebe u kontekstu u kojem se očekuje realan izraz, npr. ukoliko želimo dodijeliti neki cjelobrojni izraz realnoj promjenljivoj. Ovakvu automatsku konverziju, pri kojoj se "uži" tip konvertira u "širi" obično zovemo *promocija*. S druge strane, realan izraz često možemo upotrijebiti u kontekstu u kojem se očekuje cjelobrojni izraz, npr. kao argument neke funkcije koja očekuje cjelobrojni argument, ili pri dodjeli realnog izraza cjelobrojnoj promjenljivoj. Takve konverzije praćene su *gubitkom informacija* (u konkretnom primjeru *odsjecanjem decimala*), i obično ih nazivamo *degradacija*. C++ kompajleri imaju pravo da nailaskom na degradirajuću automatsku konverziju emitiraju *poruku upozorenja*, ali ne smiju odbiti da prevedu program. Dalje, podsjetimo se da u jeziku C možemo izvršiti *eksplicitnu konverziju* proizvoljnog izraza *izraz* u neki tip *tip*, pomoću tzv. *operatora konverzije tipa* (engl. *type-cast*) koji ima sintaksu

(*tip*) *izraz*

Ovaj operator se može koristiti i u jeziku C++. Pogledajmo, na primjer, sljedeći programski isječak koji ispisuje količnik dvije cjelobrojne vrijednosti unesene sa tastature:

```
int broj_1, broj_2;
cin >> broj_1 >> broj_2;
cout << (double)broj_1 / broj_2;
```

U ovom isječku, pomoću eksplicitne konverzije privremeno smo promijenili tip promjenljive "`broj_1`" u realni tip "**`double`**" da bismo izbjegli da operator "`/`" bude interpretiran kao cjelobrojno dijeljenje. Mada ovakva sintaksa i dalje radi u jeziku C++, ona se više *ne preporučuje*. C++ uvodi dvije nove sintakse za konverziju tipova. Prva je takozvana *funkcijska* ili *konstruktorska* notacija, koja izgleda ovako:

tip (*izraz*)

Dakle, u ovoj sintaksi, ime tipa se tretira kao *funkcija koja svoj argument prevodi u rezultat iste vrijednosti*, *ali odgovarajućeg tipa*. Prethodni primjer bi se, u skladu sa ovom sintaksom, napisao ovako:

```
int broj_1, broj_2;
cin >> broj_1 >> broj_2;
cout << double(broj_1) / broj_2;
```

Ova sintaksa može biti mnogo praktičnija kod složenijih izraza. Naime, zbog vrlo visokog prioriteta operatora konverzije tipa u C stilu, često su svakako potrebne dodatne zagrade. Na primjer, neka želimo izračunati i ispisati cijeli dio produkta "`2.541 * 3.17`". Koristeći C notaciju, to bismo uradili ovako:

```
cout << (int)(2.541 * 3.17);
```

Dodatne zagrade su neophodne, jer bi se u suprotnom operator konverzije "`(int)`" odnosio samo na podatak "`2.541`" (odnosno, kao rezultat bismo dobili vrijednost produkta "`2 * 3.17`"). Korištenjem funkcijske notacije, istu stvar možemo uraditi ovako:

```
cout << int(2.541 * 3.17);
```

Na ovaj način, ime tipa "**`int`**" se ujedno može interpretirati i kao *funkcija* koja vraća kao rezultat cijeli dio svog argumenta.

Funkcijska notacija posjeduje dva ograničenje – ime tipa u koji se vrši konverzija može se sastojati *samo od jedne riječi*, i nije podržana konverzija u tzv. *anonimne tipove* (poput "**`int`** `*`", što predstavlja tip pokazivača na cijele brojeve). Na primjer, ukoliko želimo konvertirati promjenljivu "`a`" u tip "**`unsigned long int`**", to *ne možemo* uraditi ovako:

```
unsigned long int(a)
```

ali možemo ovako (u C stilu):

```
(unsigned long int)a
```

Druga sintaksa za konverziju tipa uvedena u jeziku C++, koja ne posjeduje navedene nedostatke, koristi novu ključnu riječ "**`static_cast`**", a izgleda ovako:

`static_cast`<*tip*> (*izraz*)

U skladu sa ovom sintaksom, prethodno napisani primjeri izgledali bi ovako:

```
int broj_1, broj_2;
cin >> broj_1 >> broj_2;
cout << static_cast<double>(broj_1) / broj_2;
```

odnosno

```
cout << static_cast<int>(2.541 * 3.17);
```

Ova sintaksa je dosta rogobatna, ali ona ima svoje prednosti. Prvo, mjesta gdje se vrše konverzije tipa su potencijalno opasna mjesta, koja često prave probleme kada se program treba prenijeti sa jednog tipa računara na drugi, ili sa jednog na drugi operativni sistem. Korištenjem ove sintakse, takva mjesta u programu je lako locirati prostim traženjem ključne riječi "**static_cast**". Drugo, konverzija tipa se za različite tipove u C++-u često vrši na suštinski različite načine, na šta programer ranije nije imao nikakvog utjecaja. Sada su u C++-u uvedene četiri ključne riječi za konverziju tipova (pored već spomenute "**static_cast**", imamo i "**const_cast**", "**reinterpret_cast**" i "**dynamic_cast**"), čime programer može eksplicitnije da iskaže tačnu namjeru kakvu konverziju želi. U detalje na ovom mjestu nećemo ulaziti.

U vezi sa tipovima i konverzijama tipova, vrlo je bitno istaći jednu minornu nekompatibilnost jezika C++ sa jezikom C, koja je nastala nakon uvođenja ISO 98 C++ standarda, a koja se može uočiti pri radu sa kompajlerima koje se striktno drže ovog standarda. Ona se tiče matematičkih funkcija iz biblioteke "cmath", odnosno funkcija poput "sqrt", "sin", "log", itd. Naime, u C-u su izrazi poput "sqrt(3)" ili "sqrt(a)" gdje je "a" neka cjelobrojna promjenljiva savršeno legalni, s obzirom da se u tim slučajevima automatski vrši promocija cjelobrojnih vrijednosti "3" odnosno "a" u realne vrijednosti koje funkcija "sqrt" očekuje kao argument. Međutim, ovakve konstrukcije u C++-u pod ISO 98 standardu prijavljuju grešku (mada kod mnogih kompajlera, koji donekle izlaze izvan okvira ISO 98 standarda, neće doći ni do kakve prijave greške)! Šta se dešava? Zar automatska promocija nije podržana i u jeziku C++? Naravno da jeste, ali je odgovor na ovo pitanje mnogo suptilniji.

Da bismo vidjeli šta se ovdje zapravo dešava, podsjetimo se da u jeziku C, kao i u C++-u, postoje *tri tipa* realnih promjenljivih, nazvani "**float**", "**double**" i "**long double**", koji se međusobno razlikuju po dozvoljenom opsegu i preciznosti (broju tačnih cifara koje se mogu zapamtiti). U C-u, svaka matematička funkcija iz biblioteke "math.h" uvijek očekuje argument tipa "**double**", i daje rezultat tipa "**double**". To znači da ukoliko je argument tipa "**float**", vrši se njegova promocija u tip "**double**", a ukoliko je argument tipa "**long double**", vrši se njegova degradacija u tip "**double**". Ovo nije posve dobra ideja. Naime, na taj način se ne može iskoristiti činjenica da se matematičke operacije sa tipom "**float**" mogu obavljati brže (s obzirom da su promjenljive tipa "**float**" manje tačnosti i zauzimaju duplo manji prostor u memoriji), niti iskoristiti povećana tačnost koju pružaju promjenljive tipa "**long double**" (jer će doći do njihove degradacije u tip "**double**" čim se proslijede kao argument nekoj funkciji). Da bi se ovaj problem riješio, standard ISO 98 jezika C++ predvidio je da se svaka matematička funkcija iz biblioteke "cmath" izvede 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.

Sad smo u stanju da objasnimo zbog čega nastaju problemi ako kao argument neke od funkcija iz biblioteke "cmath" upotrijebimo cjelobrojnu vrijednost. Naime, jasno je da se ova cjelobrojna vrijednost treba pretvoriti u realnu vrijednost prije prosljeđivanja funkciji, ali je nejasno u *kakvu* realnu vrijednost (tj. da li tipa "**float**", "**double**" ili "**long double**"), s obzirom da od tipa odabrane pretvorbe ovisi i tačnost sa kojim će vrijednost funkcije biti izračunata. Srećom, ovaj problem se vrlo lako 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);
```

dovešće do prijave greške da je poziv nejasan odnosno *dvosmislen* (engl. *ambiguity*). Ovaj problem rješavamo eksplicitnom pretvorbom. Drugim riječima, zavisno od željene tačnosti računanja, upotrijebićemo neku od sljedeće tri konstrukcije:

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

U trećem slučaju, ključnu riječ "**static_cast**" smo morali upotrijebiti, 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 cjelobrojna 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). Problemi ove vrste bi se mogli izbjeći ukoliko se koristi staro zaglavlje "math.h", ali takvo rješenje je vrlo loše, s obzirom da stara verzija matematičke biblioteke nije dobro prilagođena konceptima jezika C++, tako da bismo, na duge staze, upotrebom stare biblioteke prije ili kasnije naišli na neočekivane probleme druge prirode. Stoga će nas kompajler upozoriti da zaglavlje "math.h" ne treba koristiti u C++ programima.

Već je rečeno da se opisani problem ne pojavljuje sa svim C++ kompajlerima. Naime, zbog činjenice da se ovaj problem pokazao prilično frustrirajući za većinu korisnika jezika C++ (naročito onih manje iskusnih), mnogi autori kompajlera za C++ u biblioteku "cmath" dodaju i verzije matematičkih funkcija koje primaju cjelobrojne argumente, koje se onda automatski preusmjeravaju na verzije koje primaju argument tipa "**double**", kao što je bilo u staroj verziji matematičke biblioteke sa zaglavljem "math.h". Na ovaj način su opisani problemi izbjegnuti. Ipak, ovakvo ponašanje nije predviđeno standardom ISO C++ 98, tako da ga ne možemo očekivati kod kompajlera koji striktno poštuju ovaj standard. Stoga se dobro navikavati na korištenje eksplicitne pretvorbe tipa, čak i ukoliko kompajler to ne traži, jer ćemo time izbjeći eventualne probleme ukoliko isti program kasnije trebamo prevesti pomoću nekog kompajlera koji striktno poštuje standard. Dodatna prednost je što eksplicitnom konverzijom tipa možemo neposredno utjecati na tačnost računanja.

Za ispravno razumijevanje nekih složenijih koncepata jezika C++, bitno je na samom početku dobro razumjeti suštinsku razliku između *inicijalizacije* (engl. *initialization*) i *dodjele* (engl. *assignment*). Naime, treba dobro razlikovati konstrukciju poput

```
int a = 3;
```

u kojoj se promjenljiva "a" *inicijalizira* na vrijednost "3", i konstrukcije poput

```
int a;
a = 3;
```

u kojoj se promjenljivoj "a" *dodjeljuje* vrijednost "3". Naime, u prvoj konstrukciji, u memoriji se stvara promjenljiva "a" koja se odmah pri stvaranju inicijalizira na vrijednost "3". U drugom slučaju, prvo se stvara *neinicijalizirana* promjenljiva "a" (slučajnog i nepredvidljivog sadržaja), kojoj se kasnije *dodjeljuje* vrijednost "3". Za razliku od inicijalizacije, koja se vrši nad objektom koji prije toga *nije postojao*, u procesu dodjele 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*.

Početnicima ukazivanje na ovu razliku može djelovati kao nepotrebno sitničarenje. I zaista, u ovako jednostavnom primjeru, kada se radi o jednostavnim tipovima poput "**int**", razlika između inicijalizacije i dodjele je zaista minorna, tako da mnogi na nju neće obraćati pažnju. Ta razlika također nije mnogo bitna kod svih tipova podataka naslijeđenih iz jezika C. Međutim, kod rada sa složenijim objektima, koji ne postoje u C-u, razlike postaju mnogo izražajnije. U slučaju složenijih objekata ćemo vidjeti da dodjela može biti znatno neefikasnija od inicijalizacije. Pored toga, postoje i takvi objekti nad kojima se uopće ne može vršiti dodjela, već samo inicijalizacija, odnosno dodjela se ne može ni primijeniti. 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 objektima mnogo složenije strukture nego što su cjelobrojne promjenljive, odnosno složenijim korisnički definiranim tipovima podataka.

Da bi se jasnije istakla razlika između inicijalizacije i dodjele (na koju se u jeziku C++ treba dobro navići), u C++-u je uvedena i alternativna sintaksa za inicijalizaciju, u kojoj se ne koristi znak dodjele "=", nego se početna vrijednost promjenljive navodi unutar zagrada. Ova sintaksa je u jeziku C++ preporučena, dok je stara sintaksa inicijalizacijie (sa znakom "=") zadržana pretežno radi kompatibilnosti sa jezikom C. Na primjer, umjesto deklaracije

```
int a = 3, b = 2;
```

u C++-u možemo (i trebamo) koristiti deklaraciju

```
int a(3), b(2);
```

Ova nova sintaksa uvedena je iz nekoliko razloga. Prvo, ona nas odmah podstiče da se navikavamo na razliku između inicijalizacije i dodjele. Drugo, C++ je uveo i neke složenije 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). Treće, postoje izvjesni tipovi objekata u jeziku C++, koje ćemo kasnije upoznati, kod kojih bi upotreba znaka "=" za inicijalizaciju bila vrlo zbunjujuća, te je stoga i zabranjena. Vidjećemo da prilikom dizajniranja vlastitih korisničkih tipova podataka, programer može odlučiti da li će inicijalizacija pomoću znaka dodjele "=" uopće biti dozvoljena ili ne.

Recimo sada nešto o tretmanu konstantnih vrijednosti u jeziku C++. U jeziku C, konstantne vrijednosti se obično definiraju pomoću preprocesorske direktive "#define", na primjer kao

```
#define BrojStudenata 50
#define PI 3.141592654
```

Ovaj način posjeduje mnoge nedostatke. Najveći nedostatak je što preprocesorski elementi ne podliježu sintaksnoj kontroli i tipizaciji jezika, tako da eventualna greška u njihovoj definiciji često ostaje neotkrivena, ili bude otkrivena tek prilikom njihove upotrebe. Stoga, u jeziku C++ ovaj način treba izbjegavati. Generalno, preprocesor "nije u duhu" C++-a, tako da njegovu upotrebu u C++-u treba smanjiti na najnužniji minimum (po mogućnosti, samo na direktivu "#include"). Umjesto toga, u C++-u konstante deklariramo kao i promjenljive, uz dodatak ključne riječi "**const**":

```
const int BrojStudenata = 50;
const double PI = 3.141592654;
```

S obzirom da se kod konstanti može govoriti samo o *inicijalizaciji* (a ne o *dodjeli*), uputnije je koristiti sintaksu koja ne koristi znak dodjele, nego zagrade:

```
const int BrojStudenata(50);
const double PI(3.141592654);
```

Konstante moraju biti inicijalizirane, tako da deklaracija poput

```
const int broj;
```

nije dozvoljena.

Interesantno je napomenuti da ključna riječ "**const**" postoji i u jeziku C, ali u njemu ima slabiju težinu nego u C++-u. Naime, u C-u ključna riječ "**const**" i dalje deklarira *promjenljive* (koje imaju svoju *adresu* i zauzimaju *prostor u memoriji*), ali čiji se sadržaj nakon inicijalizacije više ne može mijenjati. Ovakve kvazi-konstante obično se zovu nesretnim imenima poput *konstantne promjenljive* (iako je, sa lingvističkog aspekta, jasno da je ovaj termin klasični oksimoron) ili, još gore, *neizmjenljive promjenljive* (engl. *unmodifable variables*). Možda je bolje koristiti termin *neprave konstante*. One se ne mogu koristiti u svim kontekstima u kojima i prave konstante. Na primjer, u standardnom C-u nije dozvoljena sljedeća konstrukcija (bez obzira što će je neki kompajleri prihvatiti):

```
const int BrojStudenata = 50;
int ocjene[BrojStudenata];
```

dok sljedeća konstrukcija jeste:

```
#define BrojStudenata 50
int ocjene[BrojStudenata];
```

S druge strane, u C++-u su obje konstrukcije ispravne, a samo je prva preporučena. Naime, u C++-u je sa "**const**" moguće deklarirati *prave konstante* (engl. *true constants*), pod uvjetom da se njihova inicijalizacija izvrši ili *brojem* ili *konstantnim izrazom* (engl. *constant expression*), koji se definira kao izraz koji se sastoji isključivo od brojeva i drugih pravih konstanti. Tako, je u prethodnom primjeru "`BrojStudenata`" prava konstanta.

Sljedeći primjer ilustrira *neprave konstante*. U njemu je definirana neprava konstanta "`starost`", koja zbog načina inicijalizacije, očito nije prava konstanta:

```
int godina_rodjenja, tekuca_godina;
cout << "Unesite godinu rođenja: ";
cin >> godina_rodjenja;
cout << "Unesite tekuću godinu: ";
cin >> tekuca_godina;
const int starost = tekuca_godina - godina_rodjenja;
```

U ovom primjeru se od korisnika prvo traže podaci o godini rođenja i tekućoj godini, a zatim se na osnovu ovih podataka računa starost, koja se koristi za inicijalizaciju (neprave) konstante "`starost`". Ovdje je upotrijebljena (neprava) 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, bilo pravih, bilo nepravih, je da omoguće kompajleru prijavu greške u slučaju da kasnije u programu nehotično probamo promijeniti 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. Stoga ona ne može biti prava konstanta.

Na kraju, završimo ovo izlaganje napomenama o formatiranom ispisu podataka. Poznato je da funkcija "printf" naslijeđena iz jezika C posjeduje obilje mogućnosti za formatiranje ispisa. Slične mogućnosti mogu se ostvariti pomoću objekata izlaznog toka, samo na drugi način. Funkcija "width" primijenjena nad objektom "cout" postavlja željenu širinu ispisa podataka (širina se zadaje kao argument), dok funkcija "precision" postavlja željenu preciznost ispisa (tj. broj tačnih cifara) prilikom ispisa realnih podataka. Tako, na primjer, ukoliko izvršimo slijedeću sekvencu naredbi:

```
cout << "10/7 = ";
cout.width(20);
cout.precision(10);
cout << 10./7;
```

ispis će izgledati ovako:

```
1/7 =          1.428571429
```

Ovdje treba obratiti pažnju da "20" nije *broj razmaka* koji se ispisuju ispred podatka, nego *broj mjesta koje će zauzeti podatak*. Kako u našem slučaju podatak zauzima ukupno 11 mjesta (10 cifara i decimalna tačka), on se dopunjuje sa 9 dodatnih razmaka ispred, tako da će ukupna širina ispisa biti 20 mjesta. Za slučaj kada je zadana širina ispisa manja od minimalne neophodne širine potrebne da se ispiše rezultat, funkcija "width" se ignorira.

Nažalost, funkcija "width" djeluje samo na prvi sljedeći ispis, nakon čega se opet sve vraća na normalu. Pretpostavimo da želimo uljepšati ispis iz nekog programa koji računa i prikazuje podatke o prihodima, oporezovanom prihodu, porezu i čistom prihodu. Ovo uljepšavanje bi moralo rezultirati dosadnim ponavljanjem konstrukcija "cout.width", kao u sljedećem isječku:

```
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 može proizvesti ispis poput sljedećeg (konkretan izgled zavisi od vrijednosti odgovarajućih promjenljivih):

```
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 funkcije "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 upotrebom, 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;
```

Naravno, za tu svrhu treba pomoću direktive "`#include`" uključiti zaglavlje biblioteke "`iomanip`" u program.

Primijetimo da smo u svim navedenim primjerima unutar navodnika umetali razmake, bez obzira na to što ćemo kasnije dodatno podešavati širinu ispisa brojčanih rezultata upotrebom funkcije "`width`" ili 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.

Pored manipulatora "`setw`" postoje i brojni drugi manipulatori. Pomenućemo samo još manipulator "`setprecision`" kojim se postiže isti efekat kao i primjenom funkcije "`precision`" nad objektom izlaznog toka, ali bez potrebe za prekidanjem toka. Tako bismo umjesto naporne konstrukcije

```
cout << "10/7 = ";
cout.width(20);
cout.precision(10);
cout << 10./7;
```

mogli prosto pisati

```
cout << "10/7 = " << setw(20) << setprecision(10) << 10./7;
```

bez potrebe za prekidanjem toka. Jedina cijena koju za to trebamo "platiti" je uključivanje zaglavlja biblioteke "`iomanip`" u program.

Predavanje 2

Jezik C++ poznaje znatno više tipova podataka u odnosu na C. Neki od novododanih tipova podataka ugrađeni su u samo jezgro jezika C++ kao nove ključne riječi, poput logičkog tipa "**bool**", dok su drugi definirani kao izvedeni tipovi podataka u standardnoj biblioteci jezika C++, poput tipova "complex", "vector", "string" itd. čija upotreba zahtijeva uključivanje zaglavlja odgovarajuće biblioteke u program.

Recimo nekoliko stvari o tipu "**bool**". 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 98 C++ standarda) ista konvencija je vrijedila i u jeziku C++. Međutim standard ISO 98 C++ 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**". Na 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, ostvarena je praktično stoprocentna kompatibilnost između tretmana logičkih izraza u C-u i C++-u, bez obzira na činjenicu da su oni različitih tipova u ova dva jezika (cjelobrojnog odnosno logičkog tipa respektivno). Na primjer, u C++-u je i dalje moguće kombinirati aritmetičke i logičke operatore u istom izrazu, zahvaljujući automatskim pretvorbama. Recimo, ukoliko se kao neki od operanada operatora sabiranja "+" upotrijebi logička vrijednost (ili logički izraz), ona će biti

konvertirana u cjelobronu vrijednost, s obzirom da operator 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 primijeni operacija množenja.

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 prethodnog 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 nejasnijim. Stoga, ukoliko je neka promjenljiva zamišljena da čuva samo logičke vrijednosti, nju treba deklarirati upravo kao takvu.

Radi planiranja memorijskih resursa, korisno je znati da promjenljive tipa "**`bool`**" zauzimaju *jedan bajt* a ne *jedan bit* kako bi neko mogao pomisliti. Ovo je urađeno iz razloga efikasnosti, s obzirom da procesori ne mogu dovoljno efikasno pristupati individualnim bitima u memoriji, tako da bi rad sa promjenljivim tipa "**`bool`**" bio isuviše spor kada bi se one pakovale tako da zauzimaju svega jedan bit u memoriji. Naravno, razlika između utroška jednog bita i jednog bajta za individualne promjenljive nije toliko bitna, ali treba imati na umu da niz od 10000 elemenata čiji su elementi tipa "**`bool`**" zauzima 10000 bajta memorije, a ne 1250 bajta, koliko bi bilo da se za jedan element tipa "**`bool`**" troši samo jedan bit.

Na ovom mjestu dobro je reći nešto o *enumeracijama*, odnosno *pobrojanim* (engl. *enumerated*) *tipovima*. Pobrojani tipovi postoje i u jeziku C, ali oni u njemu predstavljaju samo manje ili više prerušene cjelobrojne tipove. Njihov tretman u jeziku C++ je kompletno izmijenjen, pri čemu su, nažalost, morale nastati izvjesne nekompatibilnosti sa jezikom C. U jeziku C++ pobrojani 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 deklaracije "**`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, Utorak, Srijeda, Cetvrtak, Petak,
  Subota, Nedjelja};
enum Rezultat {Poraz, Nerijeseno, Pobjeda};
```

definiramo dva nova *tipa* nazvana "`Dani`" i "`Rezultat`". Promjenljive pobrojanog tipa možemo deklarirati na uobičajeni način, na primjer:

```
Rezultat danasnji_rezultat;
Dani danas, sutra;
```

Za razliku od jezika C++, u jeziku C se ključna riječ "**`enum`**" mora ponavljati prilikom svake upotrebe pobrojanog tipa. Na primjer:

```
enum Rezultat danasnji_rezultat;
enum Dani danas, sutra;
```

Ova sintaksa i dalje radi u C++-u, ali se smatra nepreporučljivom.

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++ (ovo vrijedi i za 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 "`Dani`" i "`Rezultat`" i promjenljivih "`danasnji_rezultat`", "`danas`" i "`sutra`" mogle su se pisati skupa, na sljedeći način:

```
enum Dani {Ponedjeljak, Utorak, 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* (pogotovo u C++-u).

Promjenljive ovako definiranog tipa "`Dani`" mogu uzimati samo vrijednosti "`Ponedjeljak`", "`Utorak`", "`Srijeda`", "`Cetvrtak`", "`Petak`", "`Subota`" i "`Nedjelja`" i ništa drugo. Ove vrijednosti nazivamo *pobrojane konstante tipa* "`Dani`" (u slučaju potrebe, treba znati da pobrojane konstante spadaju u *prave konstante*). Slično, promjenljive tipa "`Rezultat`" mogu uzimati samo vrijednosti "`Poraz`", "`Nerijeseno`" i "`Pobjeda`" (pobrojane konstante tipa "`Rezultat`"). Slijede primjeri legalnih dodjeljivanja sa pobrojanim tipovima:

```
danas = 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 kada budemo govorili 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 propratne 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 "`danas`" dodijeljena pobrojana konstanta "`Srijeda`". Iz istog razloga će naredba

```
cout << danas;
```

ispisati na ekran vrijednost "2", a ne tekst "Srijeda", kako bi neko mogao brzopleto 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
```

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". Pobrojane konstante kojima nije eksplicitno pridružena odgovarajuća vrijednost konvertiraju se 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 prerušene 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 prerušene cjelobrojne konstante, dok su promjenljive pobrojanog tipa samo prerušene 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 *strogo zabranjena* dodjela cjelobrojnih vrijednosti promjenljivim pobrojanog tipa, *bez eksplicitnog navođenja konverzije tipa*. Drugim riječima, sljedeća naredba nije legalna (ovdje pretpostavljamo da je "danas" promjenljiva tipa "Dani"):

```
danas = 5;
```

Ukoliko je baš neophodna dodjela poput prethodne, možemo koristiti eksplicitnu konverziju tipa:

```
danas = Dani(5);
```

S druge strane, u jeziku C je dozvoljena dodjela cjelobrojne vrijednosti promjenljivoj pobrojanog tipa, bez eksplicitnog navođenja konverzije tipa, što predstavlja jednu od malobrojnih nekompatibilnosti jezika C++ sa C-om. Osnovni razlog zbog kojeg su autori jezika C++ zabranili ovakvu dodjelu je *sigurnost* (ova zabrana je uvedena relativno kasno, pa je neki stariji kompajleri ne poštuju). Naime, u protivnom, bile bi moguće opasne dodjele poput dodjele

```
danas = 17;
```

nakon koje bi promjenljiva "`danas`" 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 "`danas`" tipa "`Dani`":

```
danas = Poraz;
```

Naime, imenovana konstanta "`Poraz`" konvertirala bi se u cjelobrojnu vrijednost, koja bi mogla biti legalno dodijeljena promjenljivoj "`danas`". Komitet za standardizaciju jezika C++ odlučio je da ovakve konstrukcije zabrani. S druge strane, posve je legalno (mada ne uvijek i previše smisleno) dodijeliti pobrojanu konstantu ili promjenljivu pobrojanog 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 pobrojanog tipa vrijednost druge promjenljive nekog drugog pobrojanog 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 pobrojanih tipova (što je također dozvoljeno u jeziku C), kao i za zabranu čitanja promjenljivih pobrojanih tipova sa tastature pomoću operatora "`>>`". Naime, ukoliko bi ovakve operacije bile dozvoljene, postavlja se pitanje šta bi trebala da bude vrijednost promjenljive "`danas`" nakon izvršavanja izraza "`danas++`" 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 "`danas`" bila "`Srijeda`", nakon "`danas++`" nova vrijednost postaje "`Cetvrtak`"), bez obzira da li novodobijena cjelobrojna vrijednost zaista odgovara nekoj pobrojanoj 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 *sljedeću* 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++. U tom slučaju, kompajler za C++ će prijaviti grešku u prevođenju, 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 "`danas++`" pisati nešto poput

```
danas = Dani(danas + 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 kada budemo govorili o preklapanju operatora.

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 pojmova 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;
```

Kompleksni brojevi predstavljaju korisno obogaćenje jezika C++, koje je ušlo u standard ISO 98 jezika C++. Kompleksni tip spada u tzv. *izvedene tipove* (engl. *derived types*) i predstavljen je riječju "`complex`". Kako ovaj tip nije ugrađen u samo jezgro C++ nego je definiran u standardnoj biblioteci jezika C++, za njegovo korištenje potrebno je uključiti u program zaglavlje biblioteke koje se također zove "`complex`". U matematici je kompleksni broj formalno definiran kao par realnih brojeva, ali kako 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 promjenljivih 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 cjelobrojnih tipova. Na primjer,

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

deklarira kompleksnu promjenljivu "`gauss`" čiji realni i imaginarni brojevi mogu biti samo cjelobrojne 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`").

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 "$2 + 3.5\,i$". Pored toga, moguće je inicijalizirati kompleksnu promjenljivu realnim izrazom odgovarajućeg realnog tipa iz kojeg je razmatrani kompleksni tip izveden (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 "=" mada, generalno, treba izbjegavati inicijalizaciju korištenjem sintakse koja podsjeća na dodjelu. Na primjer:

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

Treba naglasiti da tipovi poput "complex<**double**>", "complex<**float**>", "complex<**int**>" itd. predstavljaju bitno različite tipove i automatske konverzije između njih su prilično ograničene. Stoga je izrazito nepreporučljivo u istom izrazu miješati kompleksne promjenljive izvedene iz različitih osnovnih tipova. Mada je takvo miješanje moguće uz poštovanje izvjesnih pravila koja moraju biti zadovoljena, najbolje je takve "vratolomije" u potpunosti izbjeći. U daljem, isključivo ćemo koristiti kompleksni tip izveden iz realnog tipa "**double**".

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* , *imaginarni_dio*)

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);
```

Kompleksnim promjenljivim je moguće dodjeljivati realne vrijednosti, pa čak i cjelobrojne vrijednosti, pri čemu dolazi do automatske konverzije tipa (promocije).

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*. 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)", zahvaljujući interpretaciji tzv. *zarez-operatora* (engl. *comma operator*) koja je naslijeđena još iz jezika C, 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)"! U suštini, upravo opisani razlog je ujedno i razlog zbog kojeg se izraz poput "cin >> a, b, c" prihvata sintaksno, ali ne interpretira onako kako bi neiskusan programer mogao pomisliti na prvi pogled.

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:

real(*z*)	Realni dio kompleksnog broja *z*; rezultat je *realan* broj (odnosno, realnog je tipa)
imag(*z*)	Imaginarni dio kompleksnog broja *z*; rezultat je *realan* broj
abs(*z*)	Apsolutna vrijednost (modul) kompleksnog broja *z*; rezultat je *realan* broj
arg(*z*)	Argument kompleksnog broja *z* (u *radijanima*); rezultat je *realan* broj
conj(*z*)	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 uobičajenijoj algebarskoj notaciji "$x + y\,i$". 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 pretpostavku 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 onog tipa iz kojeg je odgovarajući kompleksni tip izveden. Recimo, moguće je miješati operande tipa "complex<**double**>" i tipa "**double**". Stoga, ukoliko su, pored gore navedenih promjenljivih, deklarirane i promjenljive "x" i "y" tipa "**double**", legalni su i izrazi poput sljedećih:

```
a + b * y
a * real(c) – b * imag(c)
sin(x) + c – complex<double>(x + y, x – y)
(a + b / x) * (c – complex<double>(y, 3)) / real(a)
```

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 djelimično miješanje tipova u izrazima, zbog izvjesnih tehničkih razloga nije dozvoljeno da se u istom izrazu miješaju operand kompleksnog tipa i operand nekog tipa *različitog od tipa iz kojeg je posmatrani kompleksni tip izveden*. Na primjer, ukoliko je "a" promjenljiva tipa "complex<**double**>" a "b" promjenljiva tipa "**int**", izraz poput "a + b" *nije legalan*. Problem se lako rješava eksplicitnom konverzijom tipa. Naime, možemo izvršiti eksplicitnu konverziju tipa promjenljive "b" u tip "**double**", ili čak direktno u kompleksni tip "complex<**double**>". Stoga će bilo koji od sljedeća dva izraza raditi korektno:

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

Zapravo, nisu legalni čak ni tako banalni izrazi poput "`a + 1`" ili "`1 / a`" (s obzirom da je broj "`1`" također tipa "**`int`**"), ali se u ovim slučajevima problem lako rješava pisanjem "`a + 1.`" odnosno "`1. / a`" (po konvenciji su brojevi koji sadrže decimalnu tačku tipa "**`double`**"). Naravno, radili bi i izrazi poput "`a + double(1)`" ili "`double(1) / a`". Napomenimo da bismo u slučaju da je "`a`" tipa "`complex<float>`" a ne "`complex<double>`", umjesto "`1.`" morali pisati "`1.F`", jer je "`1.`" tipa "**`double`**", a "`1.F`" tipa "**`float`**" (alternativno bismo mogli koristiti konstrukcije poput "`float(1)`").

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 onog kompleksnog tipa koji odgovara tipu njenih argumenata. Tako, ako se kao argumenti zadaju vrijednosti tipa "**`double`**", funkcija "`polar`" daje rezultat tipa "`complex<double>`". Ova informacija je korisna zbog prethodno navedenih ograničenja vezanih za miješanje tipova prilikom inicijalizacija i upotrebe aritmetičkih operatora.

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 << "Unesi koeficijente:\n";
  cin >> a >> b >> 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> x1((-b – sqrt(complex<double>(d))) / (2 * a));
    complex<double> x2((-b + sqrt(complex<double>(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):

```
Unesi koeficijente:
3 6 15
x1 = (-1,-2)
x2 = (-1,2)
```

Čak i u ovako jednostavnom programu ima mnogo stvari koje treba da se pojasne. Prvo, primijetimo da su promjenljive "x1" i "x2" deklarirane kao realne u slučaju kada su rješenja realna (diskriminanta nenegativna), a kao kompleksne samo u slučaju kada rješenja nisu realna. Zbog čega smo uopće provjeravali znak diskriminante, odnosno, zbog čega nismo uvijek ove promjenljive posmatrali kao kompleksne? Nezgoda je u tome što se kompleksne promjenljive uvijek ispisuju kao *uređeni parovi*, čak i kad je imaginarni dio jednak nuli. Drugim riječima, da smo koristili isključivo kompleksne promjenljive (bez ispitivanja znaka diskriminante, odnosno bez razdvajanja slučajeva kada su rješenja realna odnosno kompleksna), mogli bismo imati scenario poput sljedećeg, što nije prijatno:

```
Unesi koeficijente:
2 10 12
x1 = (-3,0)
x2 = (2,0)
```

Alternativni način da riješimo ovaj problem je da uvijek koristimo kompleksne promjenljive "x1" i "x2", ali da u slučaju kada su rješenja realna, ispisujemo samo realni dio ovih promjenljivih (koji možemo dobiti pomoću funkcije "real". Ova ideja prikazana je u sljedećem programu:

```
#include <iostream>
#include <cmath>
#include <complex>

using namespace std;

int main() {
  double a, b, c;
  cout << "Unesi koeficijente:\n";
  cin >> a >> b >> c;
  double d(b * b – 4 * a * c);
  complex<double> x1((-b – sqrt(complex<double>(d))) / (2 * a));
  complex<double> x2((-b + sqrt(complex<double>(d))) / (2 * a));
  if(d >= 0)
    cout << "x1 = " << real(x1) << "\nx2 = " << real(x2) << endl;
  else
    cout << "x1 = " << x1 << "\nx2 = " << x2 << endl;
  return 0;
}
```

Druga stvar na koju treba obratiti pažnju u ovom programu je prilično rogobatna konstrukcija "`sqrt(complex<double>(d))`". Zbog čega ne možemo samo pisati "`sqrt(d)`"? Problem je u tome što 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 krajnji rezultat smješta u promjenljivu kompleksnog tipa). Stoga nam je trebala eksplicitna pretvorba tipa "`complex<double>(d)`" da obezbijedimo da argument funkcije bude kompleksan. 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 *kodomenom* (npr. funkcija $x \to x^2$ definirana za operande x iz skupa $\mathbb{N}$ i funkcija $x \to x^2$ definirana za operande x iz skupa $\mathbb{R}$ *nisu ista funkcija*). Vidjeli smo da 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 mogućnosti ćemo detaljnije govoriti kasnije.

Konstrukcija "`sqrt(d)`" bi svakako dala kompleksne rezultate kada bi sama promjenljiva "`d`" bila kompleksna. Međutim, ukoliko uradimo tako nešto, nećemo moći izvršiti test poput "`d >= 0`", s obzirom da se kompleksni brojevi ne mogu porediti po veličini. Doduse, da li su rješenja realna u tom slučaju bismo mogli testirati na drugi način (npr. testom poput "`imag(x1) == 0`"), ali 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 (poput poređenja) nemaju smisla za kompleksne brojeve.

Interesantnu mogućnost dobijamo ukoliko u prethodnom programu sve promjenljive, uključujući "`a`", "`b`" i "`c`", deklariramo kao kompleksne promjenljive. To će nam omogućiti da možemo rješavati i kvadratne jednačine čiji su koeficijenti kompleksni brojevi (ne zaboravimo da kompleksne brojeve unosimo sa tastature kao uređene parove). Probajte sami izvršiti izmjene u prethodnom programu koje će omogućiti i rješavanje takvih jednačina. Drugim riječima, program nakon izmjene treba da prihvati i scenario poput sljedećeg, u kojem je prikazano rješavanje jednačine $(2 + 3i)\ x^2 - 5\ x + 7i = 0$ (probajte riješiti ovu jednačinu "pješke", vidjećete da nije baš jednostavno):

```
Unesi koeficijente:
(2,3) -5 (0,7)
x1 = (0.912023,-2.01861)
x2 = (-0.142792,0.864761)
```

Razumije se da ukoliko u tako modificiranom programu želite omogućiti da se realna rješenja ne ispisuju kao uređeni parovi, ne možete koristiti uvjet poput "`d >= 0`" (s obzirom da je "`d`" kompleksno) nego uvjet "`imag(x1) == 0`" ili neki srodan uvjet.

Poznato je da se u jeziku C niti jedan ozbiljniji problem ne može riješiti bez upotrebe nizova. Nizovi zaista jesu veoma korisne strukture podataka, koje se sasvim regularno mogu koristiti i u jeziku C++. Međutim, C je jezik *mnogo nižeg nivoa* od jezika C++ (u smislu da mu je filozofija razmišljanja više orijentirana ka načinu rada samog računara). Posljedica toga je da mnogi koncepti nizova iz jezika C nisu u skladu sa naprednim konceptima jezika C++. Posmatrano sa aspekta C++-a, nizovi preuzeti iz jezika C posjeduju brojne nedostatke i nedoslijednosti. Možemo reći da rad sa C-ovskim nizovima, mada sasvim legalan, nije "u duhu" jezika C++. Da bi se ovi problemi izbjegli, u standard ISO 98 jezika C++ uveden je novi tip podataka, nazvan "`vector`", koji je definiran u istoimenom zaglavlju standardne biblioteke jezika C++ (tako da za korištenje ovog tipa podataka moramo uključiti u program zaglavlje biblioteke "`vector`"). Ovaj tip podataka (zovimo ga prosto *vektor*) zadržava većinu svojstava koji posjeduju standardni nizovi, ali ispravlja neke njihove nedostatke. Promjenljive tipa "`vector`" mogu se deklarirati na nekoliko načina, od kojih su najčešći sljedeći:

`vector<` *tip_elemenata* `>` *ime_promjenljive* `;`
`vector<` *tip_elemenata* `>` *ime_promjenljive* `(` *broj_elemenata* `) ;`
`vector<` *tip_elemenata* `>` *ime_promjenljive* `(` *broj_elemenata*, *inicijalna_vrijednost* `) ;`

Na primjer, vektor "`ocjene`", čiji su elementi cjelobrojni, možemo deklarirati na jedan od sljedećih načina:

```
vector<int> ocjene;
vector<int> ocjene(10);
vector<int> ocjene(10, 5);
```

Prva deklaracija deklarira vektor "ocjene", koji je inicijalno prazan, odnosno ne sadrži niti jedan element (vidjećemo kasnije kako možemo naknadno dodavati elemente u njega). Druga deklaracija (koja se najčešće koristi) deklarira vektor "ocjene", koji inicijalno sadrži 10 elemenata, a koji su automatski inicijalizirani na vrijednost 0. Treća deklaracija deklarira vektor "ocjene", koji inicijalno sadrži 10 elemenata, a koji su automatski inicijalizirani na vrijednost 5 (tužan početak, zar ne?).

Primijetimo da "vector" *nije ključna riječ*, s obzirom da tip vektor nije tip koji je *ugrađen u samo jezgro jezika* C++, već izvedeni tip (slično kao i npr. tip "complex"), definiran u istoimenoj biblioteci "vector". Promjenljive tipa *vektor* mogu se u gotovo svim slučajevima koristiti poput običnih nizovnih promjenljivih (za pristup individualnim elementima vektora, kao i kod običnih nizova, koriste se uglaste zagrade). S druge strane, rad sa ovakvim promjenljivim je znatno fleksibilniji nego sa običnim nizovima, mada postoje i neki njihovi minorni nedostaci u odnosu na klasične nizove. U nastavku će biti opisane najbitnije razlike između običnih nizovnih promjenljivih i promjenljivih tipa *vektor*.

Na prvom mjestu, broj elemenata neke vektorske promjenljive nije *svojstvo samog tipa* koji određuje promjenljivu, nego njeno *individualno*, *dinamičko svojstvo* koje se može mijenjati tokom života vektorske promjenljive. Vektorska promjenljiva u toku svog života može povećavati i smanjivati broj svojih elemenata (samim tim, zauzeće memorije koje zauzima neka vektorska promjenljiva može se dinamički mijenjati, dok nizovi od trenutka svoje deklaracije do kraja postojanja zauzimaju uvijek istu količinu memorije). Broj elemenata pri deklaraciji vektora navodi se u *običnim*, a ne u uglastim zagradama, čime je istaknuta razlika između zadavanja broja elemenata i navođenja indeksa za pristup elementima. Na primjer, vektor "ocjene" od 10 elemenata deklarirali bismo kao

```
vector<int> ocjene(10);
```

dok bismo "obični" niz "ocjene" od 10 elemenata deklarirali kao:

```
int ocjene[10];
```

U slučaju običnih nizova, indeks unutar uglaste zagrade ima jedno značenje pri deklaraciji niza (ukupan broj elemenata niza), a sasvim drugo značenje pri upotrebi elemenata niza, kada predstavlja indeks elementa kojem se pristupa (npr. "ocjene[3]" je element niza "ocjene" sa indeksom 3). Kod vektora ova dva značenja su jasno razdvojena: obične zagrade se koriste pri *zadavanju broja elemenata*, a uglaste pri *izboru željenog elementa*. Bitno je napomenuti da ukoliko greškom zadamo deklaraciju sa uglastim zagradama poput

```
vector<int> ocjene[10];
```

nećemo dobiti grešku, nego ćemo na taj način deklarirati *niz* (*obični*) *od 10 elemenata*, *čiji su elementi* (*inicijalno prazni*) *vektori*. Vidjećemo kasnije da su ovakve konstrukcije sasvim moguće, legalne i često korisne, ali u ovom trenutku to nije ono što smo vjerovatno željeli.

Poznato je da prilikom deklaracije nizova, broj elemenata niza mora biti (prava) konstanta. Drugim riječima, konstrukcije poput sljedeće nisu legalne u C++-u (niti su bile legalne u C-u):

```
int broj_ocjena;
cout << "Koliko ima ocjena: ";
cin >> broj_ocjena;
int ocjene[broj_ocjena];
```

Neki kompajleri, poput kompajlera iz GNU porodice kompajlera će doduše prihvatiti ovakve deklaracije, ali treba znati da je to *nestandardno proširenje* (*ekstenzija*) *podržano od konkretnog kompajlera* a ne dio standarda jezika C++, i vjerovatno *neće raditi pod drugim kompajlerima* (interesantno je, međutim, da je takva mogućnost dodana u ISO 99 standard jezika C, koji je nastao nakon posljednjeg standarda ISO 98 jezika C++, pa je vrlo vjerovatno da će biti dodana u sljedeći standard jezika C++). S druge strane, kod

vektora željeni broj elemenata može biti *proizvoljan izraz*, a ne samo konstantna vrijednost ili konstantan izraz, tako da ne mora biti apriori poznat. Stoga je sljedeći programski isječak posve legalan:

```
int broj_ocjena;
cout << "Koliko ima ocjena: ";
cin >> broj_ocjena;
vector<int> ocjene(broj_ocjena);
```

Sljedeća interesantna osobina vektora je što se svi njihovi elementi *automatski inicijaliziraju na podrazumijevanu vrijednost za tip elemenata vektora* (ta vrijednost je *nula* za sve brojčane tipove), a po želji možemo pri deklaraciji zadati i drugu vrijednost koja će biti iskorištena za inicijalizaciju. Ovo je bitno drugačije 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, što možemo shvatiti kao minorni nedostatak vektora u odnosu na obične nizove. Naime, dok je kod običnih nizova moguće zadati deklaraciju poput

```
int dani_u_mjesecu[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

nešto slično nije moguće u slučaju vektora. S obzirom na brojne druge prednosti vektora u odnosu na nizove, ovo nije toliko veliki nedostatak, tim prije što postoje funkcije koje mogu čitav sadržaj niza prosto iskopirati u vektor (usput, planirano je da ovaj nedostatak bude otklonjen u sljedećoj reviziji jezika C++, koja je trenutno poznata pod radnim nazivom ISO 0X C++).

Rekli smo već da kod vektora postoji mogućnost naknadnog mijenjanja broja njihovih elemenata, što se postiže primjenom funkcije "`resize`" nad vektorom, pri čemu se kao parametar zadaje novi broj elemenata (koji može biti veći ili manji od aktuelnog broja elemenata). Na primjer, ukoliko je vektor "`ocjene`" deklariran tako da prima 10 cijelih brojeva (ocjena), a naknadno se utvrdi da je potrebno zapisati 15 ocjena, moguće je izvršiti naredbu

```
ocjene.resize(15);
```

nakon koje će broj elemenata vektora "`ocjene`" biti povećan sa 10 na 15. Postojeći elementi zadržavaju svoje vrijednosti, a novododani elementi inicijaliziraju se na podrazumijevane vrijednosti.

U jeziku C postoji tijesna veza između nizova i pokazivača iskazana kroz dvije činjenice. Prva je da se ime niza upotrijebljeno samo za sebe bez uglastih zagrada *automatski konvertira u pokazivač na prvi element niza* (uz jedini izuzetak kada je ime niza upotrijebljeno kao argument operatora "**`sizeof`**" ili "&", kada ta pretvorba ne vrijedi). Stoga, ukoliko je "`ocjene`" neki niz, tada se ime "`ocjene`" upotrijebljeno samo za sebe (tj. bez indeksiranja) interpretira kao "`&ocjene[0]`". Ova osobina se u jeziku C, zajedno sa pokazivačkom aritmetikom, često koristi da se neke operacije izvedu na efektniji način. Svi ovi trikovi rade i u jeziku C++. Na primjer, ukoliko je "`ocjene`" niz od 10 cijelih brojeva, njegove elemente možemo ispisati i sljedećim programskim isječkom, u kojem se malo "igramo" sa pokazivačkom aritmetikom (analizirajte pažljivo ovaj isječak, da se podsjetite pokazivačke aritmetike):

```
for(int *p = ocjene; p < ocjene + 10; p++) cout << *p << endl;
```

Druga činjenica koja iskazuje vezu između pokazivača i nizova iskazuje se u činjenici da se indeksiranje može primijeniti i na pokazivače, pri čemu se u slučaju da je "`p`" pokazivač a "`i`" cijeli broj, izraz oblika "`p[i]`" interpretira kao "`*(p+i)`", ali ta činjenica nije bitna za ono o čemu ovdje želimo govoriti. Bitno je to da se imena vektora upotrijebljena sama za sebe, bez indeksa, za razliku od nizova *ne konvertiraju automatski* u pokazivač na prvi element vektora. Zbog toga, prethodni isječak *neće biti sintaksno ispravan* u slučaju da "`ocjene`" nije niz nego vektor. To ne znači da se pokazivačka aritmetika ne može koristiti sa vektorima. Za elemente vektora se *garantira* da su u memoriji smješteni jedan iza drugog, u rastućem poretku indeksa, tako da je pokazivačka aritmetika valjana i sa elementima vektora. Jedino je adresu elemenata vektora neophodno uzeti *eksplicitno*, zbog nepostojanja automatske konverzije. Stoga bismo prethodni isječak mogli prepraviti ovako da radi sa vektorima (tako prepravljeni isječak radiće i sa vektorima, i sa nizovima):

```
for(int *p = &ocjene[0]; p < &ocjene[10]; p++) cout << *p << endl;
```

Umjesto "&ocjene[10]" mogli smo pisati i "&ocjene[0] + 10", s obzirom da kad jednom uzmemo adresu elementa vektora, dobijamo klasični pokazivač na koji se pokazivačka aritmetika normalno primjenjuje. Mada nekome nepostojanje automatske konverzije vektora u pokazivač na prvi element vektora može djelovati kao nedostatak a ne kao prednost, treba napomenuti da su na taj način eliminirane brojne mis-interpretacije, kao što ćemo uskoro vidjeti.

Sljedeća interesantna osobina vektora je mogućnost dodjeljivanja jednog vektora drugom pomoću operatora dodjele "=" (pri čemu se kopiraju *svi elementi*, odnosno vektor *kao cjelina*), što nije moguće sa nizovima (ni u C-u, ni u C++-u). Na primjer, ukoliko su "niz1" i "niz2" dva niza istog tipa, nije moguće izvršiti dodjelu poput "niz1 = niz2" sa ciljem kopiranja svih elemenata niza "niz2" u niz "niz1", nego je neophodno vršiti kopiranje element po element uz pomoć petlji, na primjer ovako (uz pretpostavku da je "broj_elemenata" zajednički broj elemenata oba niza):

```
for(int i = 0; i < broj_elemenata; i++) niz1[i] = niz2[i];
```

Alternativno je moguće koristiti neku bibliotečku funkciju (poput "memcpy") koja u načelu radi istu stvar. Međutim, ukoliko su "v1" i "v2" vektori čiji su elementi istog tipa, dodjela poput "v1 = v2" je sasvim legalna, pri čemu dolazi do kopiranja svih elemenata vektora "v2" u vektor "v1" (na vrlo brz i efikasan način). Vektori "v1" i "v2" pri tom čak ne moraju imati ni isti broj elemenata: veličina vektora "v1" se automatski modificira tako da nakon obavljenog kopiranja vektor "v1" ima isti broj elemenata kao i vektor "v2".

Razmotrimo sada problem poređenja dva niza odnosno vektora. Ukoliko želimo ispitati da li su dva niza jednaka odnosno različita (pri čemu smatramo da su nizovi jednaki ukoliko su im jednaki elementi sa istim indeksima), to ne možemo uraditi uz pomoć operatora "==" odnosno "!=". Što je najgore, ukoliko su "niz1" i "niz2" dva niza istog tipa, izrazi poput "niz1 == niz2" i "niz1 != niz2" su savršeno legalni, ali ne rade ono što se očekuje. Naime, zbog automatske konverzije nizova u pokazivače, u ovakvim izrazima se zapravo upoređuju *adrese prvih elemenata nizova*, a ne sami nizovi. Ovo je tipičan primjer situacije u kojoj automatska konverzija nizova u pokazivače može dovesti do zabune. Za poređenje nizova moramo ponovo koristiti konstrukcije koje koriste petlje, kao u sljedećem primjeru:

```
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!";
```

S druge strane, ukoliko su "v1" i "v2" vektori, njihovo poređenje može se slobodno izvršiti uz pomoć operatora "==" i "!=". Vektore čak možemo porediti i pomoću operatora "<", "<=", ">" i ">=", pri čemu se tada poređenje vrši po *leksikografskom kriteriju*. U slučaju nizova, ponovo bi se poredile *adrese*, što teško da može biti od osobite koristi.

Veoma interesantna mogućnost vektora je mogućnost dodavanja novih elemenata *na kraj* vektora, pri čemu se broj elemenata vektora pri takvom dodavanju *povećava za jedan*. To se postiže izvršavanjem funkcije "push_back" nad vektorom, pri čemu se kao parametar zadaje element koji se dodaje. Na primjer, naredba

```
ocjene.push_back(9);
```

povećava broj elemenata vektora "ocjene" za 1, i novododanom elementu (koji se nalazi *na kraju* vektora) dodjeljuje vrijednost "9". Kao što je već rečeno, sasvim je moguće deklarirati *prazan vektor*,

odnosno vektor koji ne sadrži *niti jedan element*, 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). Trenutno aktuelni broj elemenata u svakom trenutku možemo saznati pozivom funkcije "size" bez parametara nad vektorom. Razmotrimo, na primjer, sljedeći programski isječak:

```
vector<int> brojevi;
cout << "Unesi slijed brojeva, pri čemu 0 označava kraj unosa: ";
int broj;
do {
  cin >> broj;
  if(broj != 0) brojevi.push_back(broj);
} while(broj != 0);
cout << "Ovaj niz brojeva u obrnutom poretku glasi:\n";
for(int i = brojevi.size() – 1; i >= 0; i--) cout << brojevi[i] << " ";
```

U ovom primjeru, unosimo slijed brojeva u vektor, sve dok se ne unese nula. Pri tome je odgovarajući vektor na početku prazan, a svaki novouneseni broj (osim terminalne nule) dodajemo na kraj vektora pomoću funkcije "push_back". Na kraju, unesene brojeve ispisujemo u obrnutom poretku, pri čemu broj unesenih elemenata saznajemo pomoću funkcije "size".

Treba napomenuti da operator "**sizeof**" primijenjen na promjenljive vektorskog tipa daje rezultat koji *nije u skladu sa intuicijom*. Zbog toga, razni prljavi trikovi sa "**sizeof**" operatorom, koji se u izvjesnim situacijama mogu iskoristiti za određivanje broja elemenata nekog niza, a koji se mogu susresti u nekim C programima, *ne daju ispravan rezultat* ukoliko se primijeni na vektore. Stoga, za određivanje broja elemenata vektora uvijek treba koristiti funkciju "size".

Veliki nedostatak nizova je u tome što se oni ne mogu prenositi kao parametri u funkcije. Doduše, zahvaljujući automatskoj pretvorbi nizova u pokazivače izgleda *kao da se nizovi mogu prenositi kao parametri u funkcije*, ali se pri tome u funkciju ne prenosi niz, već samo *pokazivač na prvi element niza*, a funkcija se ponaša *kao da operira sa nizom* zahvaljujući činjenici da se na pokazivače može primjenjivati indeksiranje. Ipak, funkcija ni na kakav način ne može saznati broj elemenata niza koji joj je tobože "prenesen" kao parametar, nego se broj elemenata niza mora prenositi u funkciju kao dodatni parametar. Neka je, na primjer, potrebno napisati funkciju "Prosjek" koja kao rezultat vraća aritmetičku sredinu niza cijelih brojeva koji joj je "prenesen" kao parametar. Pošto nema načina da funkcija sazna koliko niz ima elemenata, broj elemenata se mora prenijeti kao dodatni parametar. Takva funkcija bi mogla izgledati recimo ovako:

```
double Prosjek(int niz[], int broj_elemenata) {
  double suma(0);
  for(int i = 0; i < broj_elemenata; i++) suma += niz[i];
  return suma / broj_elemenata;
}
```

Ukoliko bismo ovu funkciju htjeli upotrijebiti da nađemo prosjek svih ocjena iz niza "ocjene", uz pretpostavku da ih ima 10, to bismo mogli uraditi ovako:

```
cout << "Prosjek ocjena iznosi " << Prosjek(ocjene, 10);
```

Vidimo da moramo eksplicitno prenositi broj ocjena kao parametar. Pored toga, formalni parametar "niz" unutar funkcije "Prosjek" *uopće nije niz već pokazivač* (tako da bismo identičan efekat dobili kada bismo deklaraciju "**int** niz[]" zamijenili sa "**int** *niz", što se u C programima često radi), a ponaša se slično poput niza isključivo zahvaljujući činjenici da se na pokazivače može primijeniti indeksiranje. S druge strane vektori se *zaista* mogu prenositi kao parametri u funkcije (bez konverzije u pokazivače), pri čemu je broj elemenata prenesenog vektora moguće saznati pomoću funkcije "size". Razmotrimo sljedeću funkciju, koja je analogna prethodnoj, ali radi sa vektorima:

```
double Prosjek(vector<int> v) {
  double suma(0);
  for(int i = 0; i < v.size(); i++) suma += v[i];
  return suma / v.size();
}
```

Primijetimo kako se deklarira formalni parametar funkcije vektorskog tipa – bez ikakve posebne specifikacije broja elemenata. Ukoliko je sada "`ocjene`" vektor, prosječnu ocjenu možemo naći sljedećim pozivom, bez obzira na to koliko ih ima:

```
cout << "Prosjek ocjena iznosi " << Prosjek(ocjene);
```

Kada budemo govorili o referencama, vidjećemo da se, radi nekih razloga efikasnosti, formalni parametri funkcija vektorskog tipa obično deklariraju na nešto drugačiji način, ali to ne mijenja smisao onoga o čemu na ovom mjestu govorimo.

Nizovi se ne mogu vraćati kao rezultati iz funkcija, što je često veliko ograničenje, koje se može zaobići samo pomoću nekih dosta nezgrapnih konstrukcija, koje uključuju prenošenje adrese odredišnog niza koji bi trebao da prihvati rezultat kao dodatni parametar u funkciju. Sve ove zavrzlame nisu potrebne ukoliko koristimo vektore, jer se vektori najnormalnije mogu vraćati kao rezultati iz funkcije. Razmotrimo, na primjer, sljedeću funkciju, koja sabira dva cjelobrojna vektora koja su joj prenesena kao parametri, i vraća kao rezultat zbir ta dva vektora računat element po element. Radi jednostavnosti, pretpostavimo da vektori koji se prenose kao parametri u ovu funkciju imaju jednak broj elemenata:

```
vector<int> ZbirVektora(vector<int> a, vector<int> b) {
  vector<int> c(a.size());
  for(int i = 0; i < a.size(); i++) c[i] = a[i] + b[i];
  return c;
}
```

Istu stvar smo mogli uraditi i ovako, uz upotrebu funkcije "`push_back`":

```
vector<int> ZbirVektora(vector<int> a, vector<int> b) {
  vector<int> c;
  for(int i = 0; i < a.size(); i++) c.push_back(a[i] + b[i]);
  return c;
}
```

Bez obzira koje smo od ova dva rješenja primijenili, ukoliko su "`v1`", "`v2`" i "`v3`" tri cjelobrojna vektora, možemo koristiti naredbu poput

```
v3 = ZbirVektora(v1, v2);
```

da saberemo vektore "`v1`" i "`v2`" i rezultat smjestimo u vektor "`v3`". Ništa nalik ovome nije moguće ukoliko su "`v1`", "`v2`" i "`v3`" nizovi a ne vektori.

Sa vektorima je čak moguće otići i korak dalje. Izrazi poput "`v1 + v2`" nemaju smisla za nizove, a podrazumijevano nemaju smisla ni za vektore. Međutim, u slučaju vektora, moguće je definirati smisao svih operatora čije značenje nije inicijalno definirano. Na primjer, dovoljno je da u prethodnom primjeru funkciji "`ZbirVektora`" promijenimo ime u "**`operator`** `+`" pa da operator "+" postane definiran nad vektorima. Drugim riječima, ukoliko napišemo

```
vector<int> operator +(vector<int> a, vector<int> b) {
  vector<int> c;
  for(int i = 0; i < a.size(); i++) c.push_back(a[i] + b[i]);
  return c;
}
```

tada ćemo za sabiranje dva vektora moći koristiti sintaksu poput "`v3 = v1 + v2`". Ovaj primjer je naveden ovdje samo kao ilustracija mogućnosti koju posjeduju vektori i njihovoj povećanoj fleksibilnosti u odnosu na nizove. Za potpuno razumijevanje ovog primjera potrebno je upoznati se sa pojmom *operatorskih funkcija*, o čemu ćemo detaljno govoriti kada budemo govorili o preklapanju (preopterećivanju) operatora.

Kada koristimo nizove, nije jednostavno umetnuti novi element između dva proizvoljna elementa niza, niti izbrisati element koji se nalazi na proizvoljnoj poziciji u nizu. Naime, umetanje novog elementa između dva elementa zahtijeva pomjeranje svih elemenata niza koji slijede iza novoumetnutog elementa za jedno mjesto naviše (što možemo izvesti recimo "**`for`**" petljom), dok brisanje elementa na proizvoljnoj poziciji u nizu zahtijeva pomjeranje svih elemenata koji slijede iza elementa koji se briše za jedno mjesto naniže. U slučaju vektora, za umetanje odnosno brisanje elemenata na proizvoljnoj poziciji na raspolaganju su funkcije "`insert`" odnosno "`erase`". Ove funkcije nećemo opisivati, s obzirom da one kao svoje parametre primaju tzv. *iteratore* koji predstavljaju generalizaciju pokazivača, a koje u ovom kursu zbog nedostaka prostora nećemo obrađivati. Ipak, treba znati da funkcije "`insert`" i "`erase`" također premještaju elemente vektora u memoriji, tako da se za slučaj velikih vektora ne izvode efikasno.

Bitno je naglasiti da se prilikom korištenja indeksiranja niti kod nizova niti kod vektora ne provjerava da li se indeks prilikom pristupa nekom elementu nalazi u dozvoljenom opsegu. To, na primjer znači da ukoliko niz ili vektor "`a`" ima recimo 5 elemenata, sljedeća naredba će "izbombardovati" memoriju bez obzira da li je "`a`" niz ili vektor:

```
for(int i = 0; i < 30000; i++) a[i] = 1000;
```

Posljedice ovakvog "bombardiranja" su nepredvidljive. Još je najbolja varijanta ukoliko operativni sistem primijeti "zločesto" ponašanje programa pa ga "ubije" (uz prijavu poruke poput "This program performed illegal operation and will be shut down" ili neke slične). Međutim, postoji mogućnost da program "izbombarduje" sam sebe (ili svoje vlastite promjenljive), što operativni sistem ne može detektirati. 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. Da bi se omogućila bolja kontrola, kod vektorskih tipova je uvedena i funkcija "`at`" koja obavlja sličnu ulogu kao i uglaste zagrade, ali uz kontrolu ispravnosti indeksa. Preciznije, izrazi oblika "`a[i]`" i "`a.at(i)`" su u potpunosti identični, ali se u drugom slučaju provjerava da li je indeks u dozvoljenom opsegu. Ukoliko nije, dolazi do bacanja izuzetka. Šta su izuzeci, i na koji način se mogu "uhvatiti" bačeni izuzeci, govorićemo kasnije. Uglavnom, neuhvaćeni izuzetak trenutno dovodi do prekida programa. Tako, ukoliko je "`a`" vektor od 5 elemenata, prilikom izvršavanja naredbe

```
for(int i = 0; i < 30000; i++) a.at(i) = 1000;
```

doći će do bacanja izuzetka čim promjenljiva "`i`" dostigne vrijednost 5. Uz pretpostavku da izuzetak nije uhvaćen, to će odmah dovesti do prekida programa, prije nego što program uspije napraviti neku drugu štetu. Stoga, funkciju "`at`" treba koristiti kad god nam je sigurnost bitna. S druge strane, upotreba uglastih zagrada je efikasnija, s obzirom da se kod funkcije "`at`" gubi izvjesno vrijeme na kontrolu ispravnosti indeksa. U svakom slučaju, ostavljena nam je mogućnost izbora. Interesantno je napomenuti da se veoma jednostavno može prepraviti dejstvo uglastih zagrada kod vektora tako da i obična upotreba uglastih zagrada dovodi do provjere ispravnosti indeksa. O tome ćemo govoriti kasnije, kada budemo govorili o nasljeđivanju.

Elementi vektora mogu biti ma kojeg legalnog tipa (vidjećemo uskoro da elementi vektora mogu također biti vektori). Međutim, potrebno je ukazati na jedan detalj. Pretpostavimo da želimo deklarirati vektor čiji su elementi kompleksni brojevi, izvedeni iz tipa "**`double`**". Ispravna deklaracija glasi

```
vector <complex<double> > v;
```

a ne

```
vector <complex<double>> v;
```

Ovdje je ključno uočiti razmak između dva znaka ">" u deklaraciji. Ovaj razmak je *neophodan*, jer bi u suprotnom dva takva znaka jedan do drugog bili neispravno protumačeni kao operator ">>", što svakako ne želimo (inače, očekuje se da će u novoj reviziji jezika C++ sintaksni analizator biti malo inteligentniji što će eliminirati potrebu za ovim razmakom, ali za sada je on neophodan).

Ranije smo naveli da promjenljive tipa "**`bool`**", suprotno očekivanjima, zauzimaju jedan bajt a ne jedan bit u memoriji. Stoga niz od recimo 10000 elemenata tipa "**`bool`**" zauzima čitavih 10000 bajta. Međutim, pri upotrebi vektora izbjegnuto je ovo rasipanje memorije, tako da unutar vektora jedan element tipa "**`bool`**" zauzima *svega jedan bit*. Drugim riječima, vektor od 10000 elemenata tipa "**`bool`**" zauzima svega 10000 bita, osnosno 1250 bajta (8 puta manje). Doduše, ova kompresija plaćena je činjenicom da je pristup elementima vektora čiji su elementi tipa "**`bool`**" neznatno sporiji nego pristup elementima nizova čiji su elementi tipa "**`bool`**". Ovo je tipični primjer pojave koja je u računarstvu poznata kao *trgovanje između prostora i vremena* (engl. *space-time tradeoff*) prema kojoj se ušteda u zauzeću memorije često plaća gubitkom po pitanju vremena izvršavanja i obrnuto.

Izloženi argumenti sasvim su dovoljni da ilustriraju prednosti vektora u odnosu na obične nizove, tako da se pri programiranju u C++-u savjetuje da se vektori koriste umjesto nizova gdje je god to moguće. Zapravo, upotreba nizova umjesto vektora u jeziku C++ prakticira se isključivo u situacijama kada se unaprijed zna tačan broj elemenata koje je potrebno pohraniti i kada se taj broj ne mijenja tokom izvršavanja programa, kao recimo u primjeru niza od 12 elemenata koji čuva broj dana u pojedinim mjesecima godine (jasno je da takav niz mora imati tačno 12 elemenata).

Na kraju izlaganja o vektorima, treba napomenuti da C++ posjeduje još jedan veoma koristan tip podataka nazvan *dek* (engl. *deque*, što je akronim od *double ended queue*). Dek se deklarira uz pomoć riječi "`deque`", na identičan način kao i vektori, pri čemu je za njihovo korištenje potrebno u program uključiti zaglavlje istoimene biblioteke (tj. "`deque`"). Dekovi su funkcionalno veoma srodni vektorima, i gotovo sve konstrukcije koje rade sa vektorima rade i sa dekovima. Često je moguće u čitavom programu riječ "`vector`" prosto zamijeniti sa "`deque`", i da sve i dalje radi. U čemu je onda razlika? Jedina bitna razlika između vektora i dekova, sa aspekta korištenja, je što dekovi pored funkcije "`push_back`", koja omogućava efikasno dodavanje elemenata na *kraj*, podržavaju i funkciju "`push_front`" koja omogućava efikasno dodavanje elemenata na *početak*. Pri tome se ova operacija obavlja *bez premještanja elemenata deka u memoriji*, što znači vrlo efikasno. Cijena koja je pri tome "plaćena" je činjenica da elementi deka nisu nužno smješteni u memoriji jedan za drugim, što *zabranjuje primjenu pokazivačke aritmetike*. Na primjer, ukoliko "`p`" pokazuje na peti element deka, ne postoji nikakva garancija da će "`p + 1`" pokazivati na šesti element deka (moguće je da hoće, ali moguće je i da neće). S druge strane, takozvana *iteratorska aritmetika*, o kojoj ovdje ne možemo govoriti radi i za nizove, i za vektore (iteratori su poopćenje pokazivača uvedeno u C++-u). Još jedna nuspojava "razbacanosti" elemenata deka u memoriji je činjenica da je pristup elementima deka nešto sporiji nego u slučaju nizova odnosno vektora. Stoga dekove treba koristiti kada nam je bitna mogućnost efikasnog dodavanja novih podataka na oba kraja (tj. na početak odnosno kraj). U svakom slučaju, ostavljena nam je mogućnost izbora da odaberemo tip podataka koji najbolje odgovara problemu koji želimo riješiti.

Predavanje 3

Vektori, o kojima smo govorili na prethodnom predavanju, lijepo se mogu iskoristiti za kreiranje *dvodimenzionalnih struktura podataka*, poput *matrica*, i još općenitije, *višedimenzionalnih struktura podataka*. Tako se, na primjer, matrice mogu formirati kao *vektori čiji su elementi vektori*. Tako, recimo, matricu sa *m* redova i *n* kolona možemo formirati kao vektor od *m* elemenata čiji su elementi vektori od *n* elemenata. To je najlakše uraditi ovako:

```
vector<vector<tip_elemenata> > ime_matrice(m, vector<tip_elemenata>(n));
```

Na primjer, matricu "a" sa 10 redova i 5 kolona, čiji su elementi realni brojevi tipa "**double**", možemo deklarirati ovako:

```
vector<vector<double> > a(10, vector<double>(5));
```

Ova prividno rogobatna sintaksa zahtijeva pažljiviju analizu. Prvo, mi ovim zapravo deklariramo *vektor čiji su elementi tipa* "vector<**double**>", odnosno vektor vektora realnih brojeva, a to nam upravo i treba. Podsjetimo se na *neophodnost* razmaka između dva znaka ">" u ovoj deklaraciji, sa ciljem da izbjegnemo neispravno tumačenje dva takva znaka jedan do drugog kao operatora ">>". Drugo, razmotrimo kakvi su parametri dati u zagradi iza imena matrice "a". Prvi parametar, "10", kao što znamo, predstavlja dimenziju vektora. U našem slučaju, to će biti *broj redova matrice*, jer će svaki element takvog vektora biti po jedan red matrice. Drugi parametar, koji u našem slučaju glasi "vector<**double**>(5)" djeluje neobično. Ova konstrukcija predstavlja *bezimeni* (*anonimni*) *vektor* od 5 elemenata (vidjećemo kasnije da je postojanje bezimenih objekata jedno od bitnih svojstava jezika C++). S obzirom da znamo da drugi parametar predstavlja *vrijednost koja se koristi za inicijalizaciju elemenata vektora*, konačno dobijamo *vektor od 10 elemenata*, *pri čemu je svaki element vektor od 5 elemenata*, što nije ništa drugo nego tražena matrica formata 10×5. Nije na odmet napomenuti da bez obzira što će neki kompajleri prihvatiti i znatno jednostavniju i logičniju sintaksu poput

```
vector<vector<double> > a(10, 5);
```

takva sintaksa *nije u skladu sa standardom jezika C++* i *ne treba je koristiti*. Činjenica da je neki kompajleri prihvataju je slučajna nuspojava nekih internih tehničkih detalja vezanih za način kako je biblioteka "vector" implementirana na nekim kompajlerima.

Ako malo bolje razmotrimo pomenuto obrazloženje u kojem se pominje kreiranje bezimenih objekata, vidjećemo da je konstrukcija "vector<**double**>(5)", kojom se kreira bezimeni vektor realnih brojeva, u suštini u potpunosti u skladu sa konvencijom jezika C++ po kojoj se *ime tipa može upotrijebiti kao funkcija*, i pri tome se uvijek stvara bezimeni objekat odgovarajućeg tipa. Zaista, ranije smo vidjeli da se konstrukcijom poput "complex<**double**>(2.5, 3.12)" konstruira bezimeni (neimenovani) kompleksni broj čiji je realni dio 2.5 a imaginarni dio 3.12, dok se konstrukcijom poput "**double**(3)" kreira bezimeni realni objekat čija je vrijednost "3.0".

Elementi matrica kreiranih na opisani način (tj. kao *vektori vektora*) automatski su inicijalizirani na nulu. U slučaju da je potrebno deklarirati matricu čiji su elementi inicijalizirani na neku drugu vrijednost, recimo na "2.5", to možemo učiniti ovako:

```
vector<vector<double> > a(10, vector<double>(5, 2.5));
```

Smisao ove deklaracije je vjerovatno jasan nakon prethodnog obavještenja. Ovako deklarirana matrica može se koristiti isto kao matrica formirana kao dvodimenzionalni niz, odnosno elementu matrice "a" u "i"-tom redu i "j"-toj koloni možemo pristupati pomoću izraza "a[i][j]".

Treba napomenuti da, slično kao u slučaju prostih vektora, možemo kreirati inicijalno praznu matricu (tj. matricu bez elemenata) pomoću konstrukcije

```
vector<vector<double> > a;
```

kojoj kasnije možemo precizirati željene dimenzije (koje se, po potrebi, mogu čak i mijenjati tokom rada matrice). Na primjer, sljedeća sekvenca naredbi također kreira matricu "a" formata 10×5, na indirektan način. Prvo kreiramo prazan vektor vektora "a", kojem zatim pomoću funkcije "resize" širimo na 10 elemenata. Nakon toga, svaki od elemenata vektora "a", koji su i sami za sebe vektori, širimo funkcijom "resize" tako da mogu prihvatiti svaki po 5 elemenata. Konačni efekat je matrica formata 10×5:

```
vector<vector<double> > a;
a.resize(10);
for(int i = 0; i < 10; i++) a[i].resize(5);
```

Neko bi nakon svega mogao pomisliti da se ne vrijedi patiti sa matricama formiranim kao vektori vektora, kada se deklaracija matrice kao dvodimenzionalnog niza može ostvariti mnogo jednostavnije. Na primjer, matrica "a" formata 10×5 može se u formi dvodimenzionalnog niza deklarirati znatno jednostavnijom deklaracijom oblika

```
double a[10][5];
```

Međutim, prednosti kreiranja matrica kao vektora vektora posjeduje mnogostruke prednosti u odnosu na kreiranja matrica kao običnih dvodimenzionalnih nizova. Na prvom mjestu, postoje veliki problemi prilikom prenosa dvodimenzionalnih nizova kao parametara u funkcije, a njihovo vraćanje kao rezultata iz funkcija je nemoguće (kao i svih drugih nizova). Drugo, kreiranjem matrica kao vektora vektora dobijamo sve prednosti koje vektori nude u odnosu na matrice. Treće, kako nas niko ne prisiljava da pojedini vektori koji tvore vektor vektora moraju imati isti broj elemenata, moguće je kreirati dvodimenzionalne strukture slične matricama, u kojima pojedini redovi imaju različiti broj elemenata. Ovakve strukture nazivaju se *grbave matrice* (engl. *ragged arrays*). Na primjer, sljedeća konstrukcija kreira grbavu matricu sa 10 redova u kojoj prvi (odnosno nulti) red ima jedan element, sljedeći red dva elementa, sljedeći red tri elementa, i tako dalje sve do posljednjeg reda koji ima 10 elemenata:

```
vector<vector<double> > a(10);
for(int i = 0; i < 10; i++) a[i].resize(i + 1);
```

Princip na kojoj radi prikazana sekvenca naredbi trebao bi biti jasan nakon prethodno datih objašnjenja. Ovakva struktura može biti veoma korisna za memoriranje donjih trougaonih matrica, kod kojih su svi elementi iznad dijagonale jednaki nuli, ili simetričnih matrica, kod kojih nije potrebno pamtiti elemente iznad dijagonale, s obzirom da su identični odgovarajućim elementima ispod dijagonale. Kod velikih matrica na taj način možemo ostvariti primijetnu uštedu memorije.

Rad sa matricama kao vektorima vektora ilustrira sljedeći, prilično dugačak program (listing ovog programa može se skinuti pod imenom "matrica.cpp" sa web stranice kursa). Polaznicima se savjetuje pažljiva analiza ovog programa, što ne bi trebalo da bude preteško, s obzirom na prisutne komentare u programu.

```
#include <iostream>
#include <vector>

using namespace std;

// Kreira matricu sa zadanim brojem redova i kolona
vector<vector<double> > KreirajMatricu(int br_redova, int br_kolona) {
  vector<vector<double> > m(br_redova, vector<double>(br_kolona));
  return m;
}

// Vraca broj redova zadane matrice
int BrojRedova(vector<vector<double> > m) {
  return m.size();
}

// Vraca broj kolona zadane matrice
int BrojKolona(vector<vector<double> > m) {
  return m[0].size();
}
```

```
// Unosi sa tastature matricu sa zadanim brojem redova i kolona i
// vraca je kao rezultat
vector<vector<double> > UnesiMatricu(int br_redova, int br_kolona) {
  vector<vector<double> > m(KreirajMatricu(br_redova, br_kolona));
  for(int i = 0; i < br_redova; i++)
    for(int j = 0; j < br_kolona; j++) {
      cout << "Element (" << i + 1 << "," << j + 1 << "): ";
      cin >> m[i][j];
    }
  return m;
}

// Ispisuje zadanu matricu
void IspisiMatricu(vector<vector<double> > m) {
  for(int i = 0; i < BrojRedova(m); i++) {
    for(int j = 0; j < BrojKolona(m); j++) {
      cout.width(10);
      cout << m[i][j];
    }
    cout << endl;
  }
}

// Testira da li su dvije matrice saglasne za sabiranje
bool MoguSeSabirati(vector<vector<double> > m1,
  vector<vector<double> > m2) {
    return (BrojRedova(m1) == BrojRedova(m2))
      && (BrojKolona(m1) == BrojKolona(m2));
}

// Vraca zbir zadanih matrica, uz pretpostavku da su
// saglasne za sabiranje
vector<vector<double> > SaberiMatrice(vector<vector<double> > m1,
  vector<vector<double> > m2) {
     vector<vector<double> > m3(KreirajMatricu(BrojRedova(m1),
       BrojKolona(m1)));
     for(int i = 0; i < BrojRedova(m1); i++)
       for(int j = 0; j < BrojKolona(m1); j++)
         m3[i][j] = m1[i][j] + m2[i][j];
     return m3;
}

// Testira da li su dvije matrice saglasne za mnozenje
bool MoguSeMnoziti(vector<vector<double> > m1,
  vector<vector<double> > m2) {
    return BrojKolona(m1) == BrojRedova(m2);
}

// Vraca produkt zadanih matrica, uz pretpostavku da su
// saglasne za mnozenje
vector<vector<double> > PomnoziMatrice(vector<vector<double> > m1,
  vector<vector<double> > m2) {
    vector<vector<double> > m3(KreirajMatricu(BrojRedova(m1),
      BrojKolona(m2)));
    for(int i = 0; i < BrojRedova(m1); i++)
      for(int j = 0; j < BrojKolona(m2); j++) {
        double suma(0);
        for(int k = 0; k < BrojRedova(m2); k++)
          suma += m1[i][k] * m2[k][j];
        m3[i][j] = suma;
      }
    return m3;
}
```

```
// Glavni program
int main() {
  int m1, n1;
  cout << "Unesi dimenzije matrice A: ";
  cin >> m1 >> n1;
  cout << "Unesi elemente matrice A:\n";
  vector<vector<double> > A(UnesiMatricu(m1, n1));
  int m2, n2;
  cout << "Unesi dimenzije matrice B: ";
  cin >> m2 >> n2;
  cout << "Unesi elemente matrice B:\n";
  vector<vector<double> > B(UnesiMatricu(m2, n2));
  cout << "Matrica A:\n";
  IspisiMatricu(A);
  cout << "Matrica B:\n";
  IspisiMatricu(B);
  if(MoguSeSabirati(A, B)) {
    cout << "Matrica A+B:\n";
    IspisiMatricu(SaberiMatrice(A, B));
  }
  else cout << "Matrice nisu saglasne za sabiranje!\n";
  if(MoguSeMnoziti(A, B)) {
    cout << "Matrica A*B:\n";
    IspisiMatricu(PomnoziMatrice(A, B));
  }
  else cout << "Matrice nisu saglasne za mnozenje!\n";
  return 0;
}
```

Sada ćemo preći na tretman stringova u jeziku C++. Kao što od ranije znamo, u C-u se stringovi predstavljaju kao obični nizovi čiji su elementi *znakovi* (*karakteri*), tj. tipa "**char**", pri čemu se specijalni znak sa ASCII šifrom 0 koristi kao oznaka kraja stringa (zbog toga obično govorimo o *nul-terminiranim nizovima znakova*). Sve manipulacije sa stringovima posmatranim kao nul-terminiranim nizovima znakova rade i dalje bez ikakvih izmjena u C++-u. Međutim, u C++-u je uveden novi tip podataka, nazvan "string", koji nudi znatne olakšice u radu sa stringovima u odnosu na pristup naslijeđen iz C-a. Stoga se u C++-u preporučuje korištenje tipa "string" kad god se zahtijeva elegantna i fleksibilna manipulacija sa stringovima.

Prije nego što obradimo detaljnije tip "string", potrebno je objasniti kako se objekti ulaznog i izlaznog toka "cin" i "cout" ponašaju sa promjenljivim tipa "**char**" i nizovima čiji su elementi tipa "**char**", s obzirom da tokovi ne postoje u C-u. Što se tiče ulaznog toka, prilikom čitanja promjenljivih tipa "**char**" pomoću operatora ">>", iz ulaznog toka se izdvaja *samo jedan znak*, dok će preostali znakovi biti izdvojeni prilikom narednih čitanja. Također, promjenljive tipa "**char**" se automatski ispisuju kao znakovi, bez potrebe da to naglašavamo i na kakav način. Sljedeća sekvenca naredbi ilustrira ove činjenice:

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

Sljedeća slika prikazuje neke od mogućih scenarija izvršavanja ove sekvence:

```
abcdefg(ENTER)
abcd
```

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

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

Interesantno je napomenuti da operator izdvajanja podrazumijevano ignorira (ako se specijalnom naredbom takvo ponašanje ne isključi) sve tzv. *bjeline* (engl. *white spaces*), u koje spadaju oznaka kraja reda, obični razmak i tabulator. Dakle, operator izdvajanja podrazumijevano *ignorira sve razmake*, što se najbolje vidi iz scenarija prikazanog na sljedećoj slici:

```
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 znake tipa oznake novog reda '\n', itd.) možemo koristiti poziv funkcije "get" bez parametara nad objektom ulaznog toka. Ova funkcija izdvaja sljedeći znak iz ulaznog toka, ma kakav on bio, i vraća ga kao rezultat (ako je ulazni tok prazan, ova funkcija zahtijeva da se ulazni tok napuni svježim podacima). 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;
```

Dva moguća scenaria izvršavanja ovog programa prikazana su na sljedećoj slici:

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

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

U prvom scenariju, nakon izvršenog unosa, promjenljive "prvi" i "drugi" sadrže znakove "a" i "b", promjenljiva "treci" sadrži *oznaku za kraj reda*, dok promjenljiva "cetvrti" sadrži znak "c", koji slijedi iza oznake kraja reda. Sada je sasvim jasno zbog čega ispis ovih promjenljivih daje prikazani ispis. U drugom scenariju, nakon izvršenog unosa, promjenljive "prvi" i "drugi", "treci" i "cetvrti" sadrže respektivno znak "a", razmak, znak "b" i ponovo razmak.

Srodna funkciji "get" je funkcija "peek" koja daje kao rezultat *sljedeći znak koji bi bio izdvojen iz ulaznog toka*, ali *ne vrši njegovo izdvajanje*. Ova funkcija je korisna kada želimo znati šta se nalazi u ulaznom toku, npr. da ispitamo da li je nakon izdvajanja nekog podatka preostalo još nekih znakova u ulaznom toku. Međutim, 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 ASCII *šifru* (u vidu broja), s obzirom da je rezultat tipa "**int**". Ispis znaka bismo mogli dobiti jedino eksplicitnom pretvorbom tipa, npr. naredbom poput sljedeće:

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

Mogućnost iščitavanja znakova znak po znak pomoću funkcije "get" demonstriraćemo sljedećim primjerom, u kojem se znakovi iščitavaju jedan po jedan, i u slučaju da je znak malo slovo, pretvara ga u veliko slovo. Pretvorba se obavlja sitnim manipulacijama sa ASCII šiframa znakova. Petlja se izvršava sve dok se ne detektira oznaka kraja reda:

```
cout << "Unesite rečenicu: ";
char znak(cin.get());
while(znak != '\n') {
  if((znak >= 'a') && (znak <= 'z')) znak += 'A' – 'a';
  cout << znak;
  znak = cin.get();
}
cout << endl;
```

Mogući scenario izvršavanja ove sekvence naredbi je sljedeći:

```
Unesite rečenicu:
Ne sam, Safete!(ENTER)
NE SAM, SAFETE!
```

Alternativno, pretvorbu malih slova u velika možemo ostvariti pozivom funkcije "toupper" iz biblioteke "cctype" (za njeno korištenje, neophodno je uključiti zaglavlje ove biblioteke u program). Pošto ova funkcija, radi kompatibilnosti sa jezikom C, vraća rezultat tipa "**int**" a ne "**char**", potrebna je eksplicitna konverzija u tip "**char**" da bi se rezultat prikazao kao znak:

```
cout << "Unesite rečenicu: ";
char znak(cin.get());
while(znak != '\n') {
  cout << char(toupper(znak));
  znak = cin.get();
}
cout << endl;
```

Može se primijetiti da i u ovom primjeru, kao u jednom od primjera u ranijim predavanjima, imamo dupliranje: funkciju "get" pozivamo kako unutar tijela petlje, tako i prije ulaska u petlju. Slično kao u tom primjeru, i ovdje dupliranje možemo izbjeći tako što očitavanje znaka pozivom funkcije "get" izvedemo kao propratni efekat unutar samog uvjeta petlje. Na taj način dobijamo sljedeću optimiziranu sekvencu:

```
cout << "Unesite rečenicu: ";
char znak;
while((znak = cin.get()) != '\n') cout << char(toupper(znak));
cout << endl;
```

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 i 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".

Bilo kakva iole složenija manipulacija sa tekstualnim podacima bila bi izuzetno mukotrpna ukoliko bi se oni morali obrađivati isključivo *znak po znak*. Stoga je pomoću funkcije "getline" nad objektom

ulaznog toka moguće 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 je da se može smjestiti i nul-graničnik koji služi kao oznaka kraja stringa). U slučaju da se prije toga dostigne kraj reda, biće pročitano manje znakova, odnosno neće biti zatražen unos novih znakova. Drugim riječima, sekvenca naredbi

```
char recenica[50];
cin.getline(recenica, 50);
```

deklarira niz od 50 znakova, a zatim traži od korisnika unos sa ulaznog uređaja (recimo tastature), nakon čega smješta *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). Da bismo izbjegli brojanje koliki je kapacitet niza "recenica" i olakšali izmjene programa u slučaju da se kapacitet promijeni, poželjno je koristiti operator "**sizeof**" koji, upotrijebljen nad nekom promjenljivom, daje broj bajtova koja ta promjenljiva zauzima u memoriji (što je u slučaju nizova znakova upravo jednako kapacitetu niza, jer jedan znak zauzima tačno jedan bajt). Stoga se kao drugi parametar funkcije "getline" obično zadaje rezultat operatora "**sizeof**" primijenjenog na prvi parametar, kao u sljedećem fragmentu, koji je funkcionalno ekvivalentan prethodnom fragmentu:

```
char recenica[50];
cin.getline(recenica, sizeof recenica);
```

Ukoliko se funkcija "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, uz pretpostavku da je niz "recenica" kapaciteta 50 znakova, poziv funkcije

```
cin.getline(recenica, sizeof recenica, '.');
```

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 "getline" sa dva parametra identičan pozivu ove funkcije sa tri parametra u kojem je treći parametar jednak oznaci kraja reda '\n', Dakle, sljedeća dva poziva su identična:

```
cin.getline(recenica, sizeof recenica);
cin.getline(recenica, sizeof recenica, '\n');
```

Vrijedi još napomenuti da se i funkcija "get" također može pozivati sa dva ili tri parametra, poput funkcije "getline", sa veoma sličnim dejstvom. Pri tome je jedina razlika što za razliku od funkcije "getline" funkcija "get" *ne uklanja granični znak* (oznaku kraja reda ili znak naveden kao treći parametar) iz ulaznog toka, nego ga ostavlja u ulaznom toku. U oba slučaja, nakon unosa znakova, u odredišnom nizu će nakon svih unesenih znakova biti upisan nul-graničnik koji označava kraj stringa. To omogućava da se sve tehnike rada sa nul-terminiranim stringovima iz jezika C mogu normalno koristiti sa ovako unesenim stringovima. Na primjer, sljedeća sekvenca koristi dobro poznatu funkciju "strlen" iz biblioteke "cstring":

```
char a[100];
cout << "Unesi neki niz znakova (ne više od 99 znakova): ";
cin.getline(a, sizeof a);
cout << "Unijeli ste " << strlen(a) << " znakova.";
```

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 (nul-graničnika). 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 "getline" iz dva razloga. Prvi razlog je nepostojanje ograničenja na broj znakova koji će biti pročitani. Doduše, ovo ograničenje je moguće uvesti pozivom funkcije "width" ili korištenjem manipulatora "setw", na analogan način kao što se pozivom funkcije "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 bjelinu (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;
```

Upotrebom manipulatora "setw(sizeof recenica)" (što je, u konkretnom primjeru, ekvivalentno sa "setw(100)") 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 izdvojiti ponovnom upotrebom 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 nizova znakova sa tastature znatno je povoljnije koristiti funkciju "getline" ili eventualno "get" nego operator ">>".

Bitno je istaći da u slučaju kada se u istom programu naizmjenično koriste operator ">>" i funkcije "getline" odnosno "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 broj: ";
cin >> broj;
cout << "Unesi 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 izdvojiti 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 "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 nul-graničnika, 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 "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 broj: ";
cin >> broj;
cin.ignore(10000, '\n');
cout << "Unesi tekst: ";
cin.getline(tekst, sizeof tekst);
cout << "\nBroj: " << broj << "Tekst: " << tekst << endl;
```

Poziv funkcije "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 "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 broj: ";
cin >> broj >> ws;
cout << "Unesi 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 broj: 123(ENTER)
Unesi tekst: ABC(ENTER)

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 "ignore" ignoriraće sve suvišne znakove unesene prilikom unosa broja. S druge strane rješenje u kojem se koristi "ws" objekat, ukloniće samo prazninu iz ulaznog toka. Nailaskom na poziv funkcije "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! Razlika je jasno prikazana na sljedećoj slici, gdje je sa lijeve strane prikazan scenarij pri prvom rješenju, a sa desne strane scenarij pri drugom rješenju:

```
Unesi broj: 123 XY(ENTER)
Unesi tekst: ABC(ENTER)

Broj: 123 Tekst: ABC
```

```
Unesi broj: 123 XY(ENTER)
Unesi tekst:
Broj: 123 Tekst: XY
```

Prikazano ponašanje uz upotrebu "`ws`" manipulatora 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 "`ignore`", takva sloboda ne bi postojala. Slijedi da odluku o tome da li koristiti objekat "`ws`" ili funkciju "`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, uvijek 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.").

Ako izuzmemo specifičnosti vezane za upotrebu tokova, dalji rad sa nul-terminiranim nizovima znakova odvija se na isti način u C++-u i u C-u. Međutim, za iole složeniju manipulaciju sa stringovima, u jeziku C++ se umjesto klasičnih nul-terminiranih stringova preporučuje korištenje tipa "`string`", koji će sad biti ukratko opisan. 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`", koje za razliku od nul-terminiranih nizova znakova možemo zvati *dinamički stringovi*, deklariraju se na uobičajeni način, kao i sve druge promjenljive (pri tome riječ "`string`", poput "`complex`" ili "`vector`", nije ključna riječ). Za razliku od običnih nizova znakova, pri deklaraciji promjenljivih tipa "`string`" ne navodi se maksimalna dužina stringa, s obzirom da se njihova veličina automatski prilagođava tokom rada. Promjenljive tipa "`string`", ukoliko se eksplicitno 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, odnosno slijed znakova između znakova navoda), 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 (pri čemu se preporučuje druga varijanta):

```
string s = "Ja sam string";
string s("Ja sam string");
```

Također, uz pretpostavku da je "recenica" neki obični nul-terminirani niz znakova (tj. niz čiji su elementi tipa "**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 pomoću 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;
```

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, tj. nizovi se ne mogu naći sa lijeve strane operatora dodjele. Bitno je prihvatiti 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 pomoću operatora ">>", pri čemu ovaj put ne moramo voditi računa o maksimalno dozvoljenoj dužini (pri tome se izdvajanje iz ulaznog toka 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 (broj znakova) promjenljive tipa "string" možemo saznati primjenom funkcije "size", na isti način kao i u slučaju promjenljivih tipa "vector". Umjesto funkcije "size", za promjenljive tipa "string" može se ravnopravno koristiti (sa istim značenjem) i funkcija "length". Tako, sljedeći isječak učitava rečenicu sa tastature i ispisuje je naopačke:

```
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];
```

Klasični nul-terminirani stringovi su po prirodi nizovi znakova, pa se kao i svi drugi nizovi mogu prenositi kao parametri u funkcije *isključivo kao pokazivači* (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 bez pretvorbe u pokazivače. Ovo je demonstrirano na primjeru sljedeće funkcije nazvane "`Sastavi`", koja *vraća kao rezultat* string koji nastaje sastavljanjem dva stringa koji su proslijeđeni kao parametri:

```
string Sastavi(string s1, 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. S obzirom da se nad promjenljivim tipa "`string`" može primjenjivati funkcija "`push_back`", kao i sa vektorima, može se napraviti i mnogo jednostavnija realizacija iste funkcije, poput sljedeće:

```
string Sastavi(string s1, string s2) {
  string s3;
  for(int i = 0; i < s1.length(); i++) s3.push_back(s1[i]);
  for(int i = 0; i < s2.length(); i++) s3.push_back(s2[i]);
  return s3;
}
```

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 (za razliku od nje, funkcija "`strcat`" *nadovezuje* sadržaj svog drugog parametra na kraj prvog parametra, mijenjajući tako sadržaj svog prvog parametra). Tako, 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", kao stvarni parametar je moguće upotrijebiti *klasični niz znakova*, uključujući i stringovne konstante. Obrnuta automatska konverzija (tj. konverzija iz tipa "string" u tip "niz znakova") *ne postoji*, tako da funkcije koje kao formalne parametre imaju klasične nizove znakova (poput funkcija iz biblioteke "cstring", poput "strlen", "strcpy", "strcat", "strcmp" itd.) *neće prihvatiti* kao stvarne parametre promjenljive i izraze tipa "string". Vidjećemo uskoro kako se ovaj problem može riješiti u slučaju da se za takvom pretvorbom pojavi potreba.

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 mogle 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 ekvivalentnan izrazu "s1 = s1 + s2". Na primjer:

```
string s("Tehnike ");
s += "programiranja";
cout << s;
```

Veoma je korisna i funkcija "substr". Ova funkcija 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", naslijeđena iz jezika C, također je 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 se upućuju na širu

literaturu koja razmatra biblioteku "string". Međutim, već su i do sada razmotrene operacije sasvim dovoljne za posve udoban rad sa tekstualnim podacima. Ipak, potrebno je razmotriti 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 funkcija "c_str" (bez parametara) kojom se ovakva konverzija može zatražiti *eksplicitno*. Ova funkcija, primijenjena na promjenljivu tipa "string", vrši njenu konverziju u *nul-terminirani niz znakova*, odnosno u niz znakova tipa "**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 funkcija "c_str" vrši odgovarajuću pretvorbu sa ciljem pretvorbe promjenljive "s" u tip kakav funkcija "strcpy" očekuje, a ostatak posla obavlja upravo ova funkcija.

Sljedeća stvar s kojom se trebamo upoznati u jeziku C++ su tzv. *izuzeci* (engl. *exceptions*). Naime, prilikom pisanja iole komplikovanijih 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 iscrtava kvadrat sastavljen od zvjezdica čije se dimenzije zadaju kao parametar nema smisla ukoliko se kao parametar zada negativan broj). Stoga je prirodno postaviti pitanje š*ta 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 proslijeđeni takvi parametri za koje ona nema smisla. U jeziku C, kao i u C++-u prije pojave standarda ISO 98 jezika C++, ovaj problem se rješavao na razne načine koji su bili više improvizacije nego prava rješenja. Na ovom mjestu nećemo opisivati sve takve načine, kao i njihove nedostatke, s obzirom da je standard ISO 98 jezika C++ predvidio način za rješavanje ovog problema zasnovan na izuzecima koji predstavljaju izuzetno moćan i fleksibilan način za rješavanje ovog problema.

Izuzeci 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 an 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). Uzmimo kao primjer funkciju za računanje faktorijela (povratni tip ćemo deklarirati kao "**double**" da proširimo opseg vrijednosti za koje se može dobiti rezultat). Ova funkcija nije definirana za slučaj kada je argument negativan. Stoga bi ona u tom slučaju trebala baciti izuzetak, kao u sljedećoj izvedbi:

```
double Faktorijel(int n) {
  if(n < 0) throw n;
  double p(1);
  for(int i = 1; i <= n; 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 preduzmemo nikakve izmjene u ostatku programa, nailazak na naredbu "**throw**" prosto će *prekinuti izvršavanje programa*, odnosno prva nepredviđena situacija dovešće 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 prihvatiti 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, koji koristi faktorijel za računanje binomnog koeficijenta *n-nad-k*, 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 binomnog koeficijenta 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 kasnije. 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 <= n; i++) p *= i;
  return p;
}
```

Za hvatanje takvih izuzetaka, odgovarajući tip formalnog parametra u naredbi "**catch**" mora biti konstantni niz znakova tipa "**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. Pretpostavimo 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 naiđe 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 dođe 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 "BinomniKoeficijent", koja opet poziva funkciju "Faktorijel" koja može eventualno baciti izuzetak (kako je ovo samo ilustracija, ovdje ćemo zanemariti činjenicu da je vrlo nepraktično i neefikasno računati binomne koeficijente svođenjem na računanje faktorijela):

```
#include <iostream>

using namespace std;

double Faktorijel(int n) {
  if(n < 0) throw n;
  double p(1);
  for(int i = 1; i <= n; i++) p *= i;
  return p;
}

double BinomniKoeficijent(int n, int k) {
  return Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
}
```

```
int main() {
  try {
    int n, k;
    cin >> n >> k;
    cout << BinomniKoeficijent(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 "`BinomniKoeficijent`" 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 "`BinomniKoeficijent`". Kako unutar nje funkcija "`Faktorijel`" nije pozvana iz "**`try`**" bloka, prekida se i izvršavanje funkcije "`BinomniKoeficijent`", i tok programa se vraća u "`main`" funkciju. Kako je u "`main`" funkciji funkcija "`BinomniKoeficijent`" 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 "`BinomniKoeficijent`". Sasvim drugu situaciju imamo u sljedećem primjeru, u kojem se izuzetak hvata već u funkciji "`BinomniKoeficijent`":

```
#include <iostream>

using namespace std;

double Faktorijel(int n) {
  if(n < 0) throw n;
  double p(1);
  for(int i = 1; i <= n; i++) p *= i;
  return p;
}

double BinomniKoeficijent(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 << BinomniKoeficijent(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 "`BinomniKoeficijent`" 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 "`BinomniKoeficijent`". Međutim, u ovom primjeru funkcija "`Faktorijel`" je pozvana iz "**`try`**" bloka unutar funkcije "`BinomniKoeficijent`", tako da izuzetak biva uhvaćen već u funkciji "`BinomniKoeficijent`". Stoga se tok programa prenosi u odgovarajući "**`catch`**" blok unutar funkcije "`BinomniKoeficijent`". Tamo se nalazi naredba "**`return`**" kojom se vrši *regularni povratak* iz funkcije "`BinomniKoeficijent`", pri čemu se kao njen rezultat vraća vrijednost 0 (navedena kao argument naredbe "**`return`**"). Stoga će u glavnom programu kao rezultat biti ispisana upravo nula. Dakle, funkcija "`BinomniKoeficijent`" 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 izuzetak *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;

double Faktorijel(int n) {
  if(n < 0) throw "Pogrešan argument u funkciji Faktorijel";
  double p(1);
  for(int i = 1; i <= n; i++) p *= i;
  return p;
}

double BinomniKoeficijent(int n, int k) {
  try {
    return Faktorijel(n) / (Faktorijel(k) * Faktorijel(n - k));
  }
  catch(...) {
    throw "Pogrešni argumenti u funkciji BinomniKoeficijent";
  }
}

int main() {
  try {
    int n, k;
    cin >> n >> k;
    cout << BinomniKoeficijent(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 "`BinomniKoeficijent`". 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 BinomniKoeficijent*", na osnovu čega se jasno vidi da je u "`main`" funkciji uhvaćen (novi) izuzetak koji je bacila funkcija "`BinomniKoeficijent`", a ne prvobitno bačeni izuzetak koji je bacila funkcija "`Faktorijel`" (i koji je uhvaćen u funkciji "`BinomniKoeficijent`").

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 prosljeđuje 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.

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 (kako procesorskih, tako i memorijskih). 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 kompajler 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 nije i neophodno) 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++ naslijeđene 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 kompajlera 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 usljed 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`".

Predavanje 4

Veliko ograničenje jezika C sastoji se u činjenici da je podržan samo jedan način prenosa parametara u funkcije, poznat pod nazivom *prenos parametara po vrijednosti* (engl. *passing by value*). Ovaj mehanizam omogućava prenošenje vrijednosti sa mjesta 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 parametria 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.

Posljedica pomenutog ograničenja je da su se u jeziku C morali prečesto koristiti pokazivači, što je čest izvor grešaka. Zamislimo, na primjer, da želimo napisati funkciju "`Udvostruci`" koja udvostručava vrijednost svog argumenta. Na primjer, želimo da sljedeća sekvenca naredbi ispiše broj 10:

```
int a(5);
Udvostruci(a);
cout << a;
```

Jasno je da to ne možemo učiniti funkcijom poput sljedeće, s obzirom da se po mehanizmu prenosa parametara u funkcije, stvarni parametar pri pozivu (u našem slučaju "`a`") samo *kopira* u formalni parametar funkcije "`x`", i svaka dalja manipulacija sa formalnim parametrom "`x`" utiče samo na njega, a ne na stvarni parametar:

```
void Udvostruči(int x) {
  x *= 2;
}
```

Kako postići da funkcija "`Udvostruci`" radi ono što smo naumili? U jeziku C nikako – naime, u C-u ne postoji nikakav način da funkcija promijeni vrijednost svog stvarnog parametra putem formalnog parametra. Najbliže tome što u C-u možemo uraditi je da formalni parametar funkcije deklariramo kao *pokazivač*, koji unutar te funkcije moramo eksplicitno dereferencirati pomoću operatora "`*`" sa ciljem da pristupimo sadržaju na koji on pokazuje, a ne samom pokazivaču:

```
void Udvostruči(int *x) {
  *x *= 2;
}
```

Međutim, posljedica ovakvog rješenja je da ćemo pri pozivu funkcije "`Udvostruci`", umjesto samog stvarnog parametra "`a`" morati prenostiti njegovu *adresu*, koju možemo dobiti pomoću operatora "&" (s obzirom da i formalni parametar, koji je pokazivač, očekuje adresu):

```
int a(5);
Udvostruci(&a);
cout << a;
```

Ovim jesmo postigli traženi efekat, ali korisnika funkcije tjeramo da pri pozivu funkcije eksplicitno prenosi adresu argumenta umjesto samog argumenta, dok sama funkcija mora dereferencirati svoj formalni argument primjenom operatora "`*`". Ovo se mora raditi kod svih funkcija koje treba da promijene vrijednost svog stvarnog argumenta ili da smjeste neku vrijednost u njega (sjetite se koliko ste puta zaboravili da upotrijebite operator "&" u funkciji "`scanf`"). Navedimo još jedan karakterističan primjer. Neka je potrebno napisati funkciju "`Razmijeni`" koja razmjenjuje vrijednost svojih argumenata (realnog tipa). Na primjer, ukoliko (realna) promjenljiva "`a`" ima vrijednost 5.12, a (realna) promjenljiva "`b`" vrijednost 8, želimo da nakon poziva

```
Razmijeni(a, b);
```

promjenljiva "a" dobije vrijednost 8, a promjenljiva "b" vrijednost 5.12. U C-u je tačno ovo nemoguće postići. Naime, ukoliko napišemo funkciju poput sljedeće:

```
void Razmijeni(double x, double y) {
  double pomocna(x);
  x = y; y = pomocna;
}
```

nećemo postići željeni efekat – razmjena će se obaviti nad formalnim parametrima "x" i "y" koji predstavljaju *kopije* stvarnih parametara "a" i "b", ali su *potpuno neovisni od njih*. Djelomično rješenje ponovo možemo postići primjenom pokazivača, odnosno prepravljanjem funkcije "Razmijeni" da izgleda ovako:

```
void Razmijeni(double *x, double *y) {
  double pomocna(*x);
  *x = *y; *y = pomocna;
}
```

Međutim, nedostatak ovog rješenja je što kao stvarne parametre ponovo moramo zadavati adrese:

```
Razmijeni(&a, &b);
```

Izloženi problem se rješava upotrebom tzv. *referenci* (ili *upućivača*, kako se ovaj termin ponekad prevodi). Reference su specijalni objekti koji su po internoj građi vrlo slični pokazivačima, ali se sa aspekta upotrebe veoma razlikuju. I reference i pokazivači 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 referenca 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 ("*p" u prvom slučaju, odnosno samo "p" u drugom slučaju).

Zbog činjenice da se reference u potpunosti poistovjećuju sa objektima na koje upućuju, reference 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. 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 cjelobrojna promjenljiva, referencu "b" vezanu za promjenljivu "a" možemo deklarirati na jedan od sljedeća dva ekvivalentna načina:

```
int &b = a;
int &b(a);          // Ova sintaksa je više u duhu C++-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, reference 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 cjelobrojna 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.

Referenca na neki objekat *nije* taj objekat (tehnički gledano, ona je zapravo neka vrsta prerušenog pokazivača na njega), ali je bitno da se ona u svemu ponaša kao da ona *jeste* taj objekat. Drugim riječima, ne postoji apsolutno 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. 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**") dotle je promjenljiva "b" tipa *referenca na cijeli broj*. 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".

Reference 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 predavanja. Naime, da bismo postigli da funkcija promijeni vrijednost svog stvarnog parametra, *dovoljno je da odgovarajući formalni parametar bude referenca*. Na primjer, funkciju "Udvostruci" trebalo bi modificirati na sljedeći način:

```
void Udvostruci(int &x) {
  x *= 2;
}
```

Da bismo vidjeli šta se ovdje zapravo dešava, pretpostavimo da je ova funkcija pozvana na sljedeći način ("a" je neka cjelobrojna promjenljiva):

```
Udvostruci(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 "Udvostruci", 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*:

```
Udvostruci(7);
Udvostruci(2 * a - 3);
```

Ovakvi pozivi zapravo i nemaju smisla. Funkcija "Udvostruci" 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 nizova, vektora i dekova također l-vrijednosti, kao stvarni parametar koji se prenosi po referenci može se upotrijebiti i individualni element niza, vektora ili deka. Tako, na primjer, ukoliko imamo deklaraciju

```
int niz[10];
```

tada se sljedeći poziv sasvim legalno može iskoristiti za uvećavanje elementa niza sa indeksom 2:

```
Udvostruci(niz[2]);
```

Ovo je posljedica činjenice da se referenca može vezati za bilo koju l-vrijednost, pa tako i za individualni element niza, vektora ili deka. 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. Tako će dodjela poput "element = 5" imati isti efekat kao i dodjela "niz[2] = 5".

Pomoću referenci ćemo jednostavno riješiti i ranije pomenuti problem vezan za funkciju "Razmijeni" koja bi trebala da razmijeni vrijednosti svojih argumenata. Naime, dovoljno je funkciju prepraviti tako da joj formalni parametri budu reference:

```
void Razmijeni(double &x, double &y) {
  double pomocna(x);
  x = y; y = pomocna;
}
```

Ova funkcija će razmijeniti proizvoljne dvije realne promjenljive (ili, općenitije, l-vrijednosti) koje joj se proslijede kao stvarni parametri. Na primjer, ukoliko je vrijednost promjenljive "a" bila 2.13 a sadržaj promjenljive "b" 3.6, nakon izvršenja poziva "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 "x" i "y", 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]);
```

razmijeniti elemente niza sa indeksima 2 i 5 (tj. treći i šesti element).

Reference u jeziku C++ su mnogo fleksibilnije nego u većini drugih programskih jezika. Na primjer, u C-u one uopće ne postoje, dok u jeziku Pascal samo formalni parametri mogu biti reference, odnosno referenca kao pojam ne postoji izvan konteksta formalnih parametara. Stoga se u Pascal-u 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.

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 prenosa parametara po referenci, pri čemu će funkcija koristeći reference prosto *smjestiti* tražene vrijednosti u odgovarajuće stvarne parametre koji su joj proslijeđeni. Ova tehnika je ilustrirana 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 zadanom broju sekundi. Ovaj prenos se ostvaruje tako što su drugi i treći formalni parametar ove funkcije ("`sati`", "`minute`" i "`sekunde`") deklarirani kao reference, koje se za vrijeme izvršavanja funkcije vezuju za odgovarajuće stvarne argumente:

```
#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);
```

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 referenca 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 cjelobrojnu 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 cjelobrojna. U skladu sa djelovanjem referenci, ova vrijednost trebala bi da se zapravo smjesti u promjenljivu za koju je ova referenca vezana, što je očigledno nemoguće s obzirom da se radi o cjelobrojnoj 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. Način kako riješiti ovaj problem razmotrićemo nešto kasnije.

Interesantan je sljedeći primjer, u kojem funkcija istovremeno daje izlazne informacije putem prenosa parametara po referenci, i kao rezultat daje *statusnu informaciju* o obavljenom odnosno neobavljenom poslu. Prikazana funkcija "KvJed" 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>
#include <cmath>

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 << "Unesi koeficijente a, b i c: ";
  cin >> a >> b >> c;
  double x1, x2;
  bool rjesenja_su_realna(KvJed(a, b, c, x1, x2));
  if(rjesenja_su_realna)
    cout << "x1 = " << x1 << "\nx2 = " << x2;
  else cout << "Jednačina nema realnih rješenja!";
  return 0;
}
```

Napomenimo da smo, s obzirom da funkcija "KvJed" vraća vrijednost tipa "**bool**" kao rezultat, mogli pisati i sekvencu naredbi poput sljedeće

```
if(KvJed(a, b, c, x1, x2))
  cout << "x1 = " << x1 << "\nx2 = " << x2;
else cout << "Jednačina nema realnih rješenja!";
```

u kojoj se računanje rješenja odvija kao propratni efekat uvjeta "**if**" naredbe. Međutim, ovakve konstrukcije treba izbjegavati, zbog smanjene čitljivosti. Uvjete sa propratnim efektima treba koristiti samo ukoliko se njihovom primjenom zaista postiže nešto korisno što bi na drugi način rezultiralo

osjetno kompliciranijim ili neefikasnijim kodom. Također, prenos parametara po referenci treba koristiti *samo kada je to zaista neophodno*. Pored toga, kako kod prenosa po referenci iz samog poziva funkcije nije vidljivo da vrijednosti stvarnih parametara mogu biti promijenjene, funkcijama koje koriste prenos po referenci treba davati takva imena da ta činjenica bude jasno vidljiva iz naziva funkcije (npr. ime poput "`SmjestiMinIMax`" ili "`NadjiMinIMax`" znatno je sugestivnije po tom pitanju nego ime poput "`MinIMax`").

Do sada smo vidjeli da formalni parametri funkcija mogu biti reference. Međutim, interesantno je da rezultat koji vraća funkcija *također može biti referenca* (nekada se ovo naziva *vraćanje vrijednosti po referenci*). Ova mogućnost se ne koristi prečesto, ali kasnije ćemo vidjeti da se neki problemi u objektno-orijentiranom pristupu programiranju 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 "r" ne predstavlja referencu na referencu "q", već referencu na promjenljivu za koju je referenca "q" vezana, odnosno na promjenljivu "p" (odnosno "r" 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 promjenljva (ili l-vrijednost). Tako su, na primjer, sasvim legalne konstrukcije poput sljedećih (ovdje su "Udvostruci" i "Razmijeni" funkcije iz ranijih primjera, samo što je funkcija "Razmijeni" modificirana da razmjenjuje *cjelobrojne vrijednosti*, a "c" neka cjelobrojna promjenljiva):

```
VeciOd(a, b) = 10;
VeciOd(a, b)++;
VeciOd(a, b) += 3;
Udvostruci(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! 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 sljedeća deklaracija posve legalna:

```
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*

referenca (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. Viseće reference su na neki način analogne tzv. *divljim pokazivačima* (engl. *wild pointers*), odnosno pokazivačima koji su "odlutali" u nepredviđene dijelove memorije. Dobri kompajleri 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 kompajlera, tako da dodatni oprez nije na odmet.

Pored običnih referenci, postoje i *reference na konstante* (preciznije, *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 ispisa promjenljive "b" ispisati vrijednost 5, ali pokušaj promjene vezanog objekta pomoću naredbe "b = 6" dovešće 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 adresa* 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 (takvi tipovi su, recimo, tipovi "vector", "deque" i "string"), 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 interno 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 uništena 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 poglavljima. 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). Kao konkretan primjer, razmotrimo funkcije "`Prosjek`", "`ZbirVektora`" i "`Sastavi`" koje su rađene na prethodnim predavanjima. Ove funkcije mogu se učiniti mnogo efikasnijim ukoliko se formalni parametri ovih funkcija deklariraju kao reference na konstantne objekte. Na taj način neće dolaziti do kopiranja masivnih objekata prilikom prenosa u funkcije, što je naročito bitno u slučaju da se kao stvarni parametri upotrijebe vektori ili stringovi sa velikim brojem elemenata. Strogo rečeno, kopiranje parametara koji nisu reference odvija se posredstvom tzv. *konstruktora kopije*, tako da stepen poboljšanja u efikasnosti bitno ovisi od toga kako su izvedeni konstruktori kopije tipova "`vector`" i "`string`". Bez obzira na to, kopiranje svakako treba izbjeći (ako nije zaista neophodno), tako da ove funkcije treba obavezno izvesti na sljedeći način:

```
double Prosjek(const vector<int> &v) {
  double suma(0);
  for(int i = 0; i < v.size(); i++) suma += v[i];
  return suma / v.size();
}
```

```
vector<int> ZbirVektora(const vector<int> &a, const vector<int> &b) {
  vector<int> c;
  for(int i = 0; i < a.size(); i++) c.push_back(a[i] + b[i]);
  return c;
}

string Sastavi(const string &s1, const string &s2) {
  string s3;
  for(int i = 0; i < s1.length(); i++) s3.push_back(s1[i]);
  for(int i = 0; i < s2.length(); i++) s3.push_back(s2[i]);
  return s3;
}
```

Slične modifikacije bi trebalo uraditi u funkcijama za rad sa matricama koje su prezentirane na prethodnom poglavlju, što možete uraditi kao korisnu vježbu. Poboljšanje efikasnosti u tom primjeru može biti drastično, pogotovo pri radu sa velikim matricama. U svim daljim izlaganjima *uvijek* ćemo koristiti konstantne reference kao formalne parametre kad god su parametri masivnih tipova, osim u slučaju kada postoji jak razlog da to ne činimo (što će biti posebno naglašeno).

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. Na primjer, 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 Crtaj(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:

```
Crtaj(5, 8, '*');
```

Pretpostavimo sada da u većini slučajeva želimo za iscrtavanje pravougaonika koristiti zvjezdicu, dok neki drugi znak želimo koristiti samo u iznimnim slučajevima. Tada stalno navođenje zvjezdice kao trećeg parametra pri pozivu potprograma možemo izbjeći ukoliko formalni parametar "`znak`" proglasimo 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 "`Crtaj`" koja koristi ovu ideju:

```
void Crtaj(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

```
Crtaj(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

```
Crtaj(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

```
Crtaj(5, 8, '0');
```

formalni parametar "`znak`" dobiti vrijednost stvarnog parametra `'0'`, odnosno dobićemo 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*. 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 Crtaj(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:

```
Crtaj(5, 8, '+');
Crtaj(5, 8);            // ima isti efekat kao Crtaj(5, 8, '*');
Crtaj(5);               // ima isti efekat kao Crtaj(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 zagrade, koje označavaju poziv potprograma, ne smiju izostaviti, već samo ostaju prazne, kao u slučaju potprograma bez parametara). Treba još napomenuti da se pri zadavanju podrazumijevanih vrijednosti *mora koristiti sintaksa sa znakom* "=", a ne konstruktorska sintaksa sa zagradama, koja je preporučena pri običnoj inicijalizaciji promjenljivih. Također, u slučaju da se koriste *prototipovi potprograma*, eventualne podrazumijevane vrijednosti parametara navode se *samo u prototipu*, ali ne i u definiciji potprograma, inače će kompajler prijaviti grešku (kao i pri svakoj drugoj dvostrukoj definiciji).

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.

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`" cjelobrojna 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, kompajler 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 cjelobrojnog tipa u drugi (npr. iz tipa "**`char`**" u tip "**`int`**") ili iz

jednog realnog tipa u drugi (npr. iz "**float**" u "**double**") smatra "logičnijom" odnosno "direktnijom" od konverzije iz cjelobrojnog 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 logičnijim 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 kompajler 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 ovdje 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 dozu opreza, moguće je formirati korisne potprograme koji kombiniraju opisane tehnike. Na primjer, razmotrimo sljedeće potprograme:

```
void Crtaj(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 Crtaj(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 njegov 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:

```
Crtaj(5, 10, '+');
Crtaj(5, 10);
Crtaj(5, '+');
Crtaj(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 efekte 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. 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). 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.

Preklapanje po tipu možemo iskoristiti i da riješimo probleme koje smo sreli kad smo pisali funkcije "`Razmijeni`" koju smo ranije napisali, a koja je mogla razmjenjivati samo parametre tipa "**double**". Djelimično rješenje je definirati još jednu funkciju "`Razmijeni`" koja bi, recimo, razmjenjivala parametre tipa "**int**". Tako bismo mogli razmjenjivati promjenljive čiji su tipovi "**double**" ili "**int**". Međutim, poželimo li razmjenjivati parametre tipa "**char**", ili recimo tipa "**float**", morali bismo kreirati još jednu verziju funkcije "`Razmijeni`", itd. što može postati veoma naporno. Sličan problem ne javlja se samo pri upotrebi referenci, nego i pri upotrebi nizova, vektora i sličnih tipova podataka. Naime, zbog nekih tehničkih razloga, ne postoje konverzije iz niza ili vektora čiji su elementi jednog tipa u niz ili vektor čiji su elementi drugog tipa. Na primjer, sljedeće funkcije, koje su namijenjene za ispis elemenata nekog niza ili vektora, mogu se koristiti samo sa nizovima odnosno vektorima čiji su elementi tipa "**double**":

```
void IspisiNiz(double niz[], int broj_elemenata) {
  for(int i = 0; i < broj_elemenata; i++) cout << niz[i] << " ";
}
void IspisiVektor(vector<double> v) {
  for(int i = 0; i < v.size(); i++) cout << v[i] << " ";
}
```

Naravno, i u ovom slučaju bismo mogli koristiti preklapanje funkcija po tipu, ali 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).

Opisani problemi se rješavaju uvođenjem tzv. *generičkih funkcija*. Generičke funkcije predstavljaju jedno od najkorisnijih svojstava jezika C++, koje su znatno doprinijeli popularnosti ovog jezika. One 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*. One se u jeziku C++ realiziraju uz pomoć tzv. *predložaka* ili *šablona* (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`" koja kao rezultat vraća veći od svoja dva argumenta:

```
template <typename NekiTip>
NekiTip VeciOd(NekiTip x, NekiTip y) {
  if(x > y) return x;
  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. Ključna riječ "**`template`**" govori da se radi o šablonu u koji je uklopljena funkcija "`VeciOd`". Iza nje se unutar šiljastih zagrada ("<>") navode *parametri šablona* (engl. *template parameters*). Parametri šabona mogu biti različitih vrsta, ali u ovom trenutku nas zanimaju samo parametri šablona koji predstavljaju ime nekog tipa koji nije poznat u trenutku definiranja funkcije (ovakvi parametri nazivaju se i *metatipovi*). Takvi parametri šablona deklariraju se pomoću ključne riječi "**`typename`**" iza koje slijedi ime parametra šablona (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 deklariranje tzv. *klasa*, o čemu ćemo govoriti kasnije. Takva praksa naslijeđe je prošlosti, s obzirom da je ključna riječ "**`typename`**" uvedena u standard jezika C++ tek nedavno.

Prilikom nailaska na šablon, kompajler 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, kompajler 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`**", kompajler 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 kompajler generirati verziju funkcije "`VeciOd`", uzimajuci da je "`NekiTip`" sinonim za tip "**`int`**",

Ovakvo naknadno generiranje funkcije, do kojeg dolazi tek kada kompajler 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 konkretan *primjerak* (*instanca*) generičke funkcije "`VeciOd`". Ukoliko se kasnije naiđe 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 naiđe na poziv funkcije "`VeciOd`" sa *drugačijim tipom* stvarnih parametara (npr. tipom "**`double`**"), biće instanciarana *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 naiđe na poziv funkcije "`VeciOd`" sa cjelobrojnim 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 uvjetno 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 mjesta 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 << VeciOd<double>(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 "`VeciOd<double>`" kao *jednu riječ*, odnosno kao *ime specijalne verzije* generičke funkcije "`VeciOd`" kod koje parametar šablona "`NekiTip`" predstavlja tip "**`double`**".

Ako pogledamo kako je parametar šablona "`NekiTip`" upotrijebljen u deklaraciji generičke funkcije "`VeciOd`", lako zaključujemo da je pretpostavljeno da će oba 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 kompajler će prijaviti grešku, kao u slučaju sljedeće sekvence naredbi:

```
double realni(5.27);
int cijeli(3);
cout << VeciOd(realni, cijeli);
```

Naime, ovdje kompajler 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 brojne konstante "`5.27`" i "`3`" nisu istog tipa. Naravno, problem bismo mogli riješiti eksplicitnom specifikacijom parametara šablona, odnosno upotrebom nekog od sljedećih poziva (prvi poziv je svrsihodniji, 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 prevođenju od strane kompajlera. 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 kompajler 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 šta 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 šta nije u redu sa tom operacijom, ukoliko nije svjestan da je funkcija pozvana sa takvim argumentima za koje ta operacija zaista nije definirana.

Opisani problem nije lako riješiti, stoga većina kompajlera primjenjuje solomonsko rješenje: greška će biti prijavljena *i na mjestu poziva funkcije*, *i na mjestu nedozvoljene operacije*. Prijavljene greške 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 ponuđene 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, tipovi argumenata u ovoj funkciji ipak ne mogu biti posve proizvoljni: oni moraju biti *objekti koji se mogu upoređivati po veličini*. Ova činjenica nameće izvjesna ograničenja na tipove argumenata 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 kompajler 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 koncepata 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 "`VeciOd`" definirati pomoću šablona koji izgleda na primjer ovako:

```
template <typename UporediviObjekat>
UporediviObjekat VeciOd(UporediviObjekat x, UporediviObjekat y) {
  if(x > y) return x;
  else return y;
}
```

Tako će svako ko pogleda zaglavlje 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).

Moguće je definirati šablone sa više parametara šablona. Pogledajmo sljedeću definiciju generičke funkcije "`VeciOd`" koja je definirana pomoću šablona sa *dva parametra*:

```
template <typename Tip1, typename Tip2>
Tip1 VeciOd(Tip1 x, Tip2 y) {
  if(x > y) return x;
  return y;
}
```

U ovom slučaju, poziv poput "`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 "`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 eksplicitna specifikacija parametara šablona, odnosno poziv poput "`VeciOd<double,double>(3, 5.27)`".

Podržano je i navođenje *nepotpune eksplicitne specifikacije* parametara šablona, kao u pozivu poput "`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 eksplicitnu 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 eksplicitno 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 "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 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 da želimo napisati funkciju koja se za većinu tipova argumenata izvodi na jedan način, ali se za neki specijalan tip (npr. tip "**int**") izvodi na *drugačiji način*. Tada možemo posebno definirati *generičku funkciju* i istoimenu *običnu funkciju* koja prihvata željeni specijalni tip (tzv. *specijalizaciju* generičke funkcije za konkretan tip). Na primjer:

```
template <typename NekiTip>
NekiTip F(NekiTip x) {
  ...
}

int F(int x) {
  ...
}
```

Prilikom poziva funkcije "F", kompajler 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", 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;
  if(y > z) return y;
  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*):

```
template <typename NekiTip>
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*, 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_elemenata) {
  for(int i = 0; i < broj_elemenata; 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 potvrdan ukoliko umjesto *djelimične* upotrijebimo *potpunu dedukciju tipa*. Naime, nećemo kompajleru 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 pretpostavi da parametar šablona "`NekiTip`" predstavlja ne tip "**`int`**", već tip *niza cijelih brojeva*. Tako će kompajler 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. Dakle, greška će biti uočena tek kada se sa parametrom pokuša izvesti operacija koja na njega nije primjenljiva (u ovom primjeru, indeksiranje). Zbog toga, potpunu dedukciju treba koristiti samo u slučaju kada je zaista potrebna *izuzetno 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 prihvatiti bilo koji objekat na koji se može primijeniti indeksiranje (za sada znamo da je to moguće za nizove, vektore i dekove). 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, vektori i dekovi 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`**" (odsjecanjem decimala).

Pomoću generičkih funkcija moguće je postići zaista veliku univerzalnost. Na primjer, u sljedećem primjeru definiraćemo funkciju "`NadjiElement`" koja pronalazi i vraća kao rezultat indeks traženog

elementa u nizu, odnosno vrijednost –1 u slučaju da traženi element nije nađen. Generički mehanizam će nam omogućiti da napišemo takvu funkciju 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 -1;
}
```

Ne bi bilo teško (korištenjem potpune dedukcije) prepraviti ovu funkciju da radi sa proizvoljnim kontejnerom sa slučajnim pristupom (npr. vektorom ili dekom).

Naredni 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). Primjer je posebno interesantan, jer se kao rezultat ne vraća prosto najveći element niza nego *referenca na njega*:

```
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];
}
```

Vraćanje reference umjesto same vrijednosti povećava mogućnosti primjene ove funkcije. Na primjer, ukoliko želimo na mjesto najvećeg elementa u nizu "a" od 10 elemenata upisati vrijednost 0, možemo koristiti poziv poput sljedećeg (da nismo vratili referencu, poziv funkcije ne bi bio l-vrijednost pa se, prema tome, ne bi mogao naći sa lijeve strane znaka jednakosti):

```
MaxElement(a, 10) = 0;
```

Veoma je bitno da ovu funkciju nismo mogli napisati na sljedeći način (koji bi bio sasvim legalan da želimo prosto vratiti kao rezultat najveći element niza, a ne referencu na njega). Naime, već smo rekli da ne smijemo vratiti iz funkcije referencu na objekat koji *prestaje postojati po završetku funkcije*, a lokalna promjenljiva "max" je upravo takva:

```
template <typename UporediviTip>
UporediviTip &MaxElement(const UporediviTip niz[], int br_elemenata) {
  UporediviTip max(niz[0]);
  for(int i = 1; i < br_elemenata; i++)
    if(niz[i] > max) max = niz[i];
  return max;
}
```

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++ (naročito funkcije iz biblioteke "algorithm") izvedene su upravo kao generičke funkcije.

Predavanje 5.

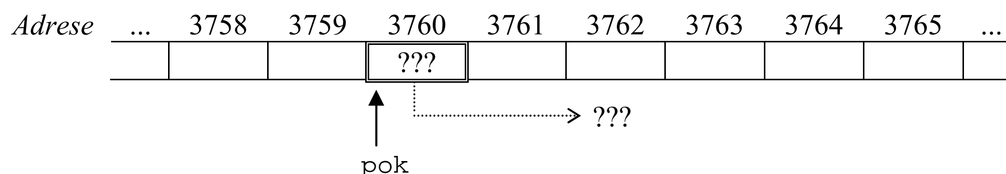
Jezik C++ je znatno osavremenio mehanizme za *dinamičku alokaciju* (*dodjelu*) *memorije* u odnosu na jezik C. Pod dinamičkom alokacijom memorije podrazumijevamo mogućnost da program *u toku izvršavanja* zatraži da mu se dodijeli određena količina memorije koja ranije nije bila zauzeta, i kojom on može raspolagati sve dok eventualno ne zatraži njeno oslobađanje. Takvim, dinamički dodijeljenim blokovima memorije, može se pristupati *isključivo pomoću pokazivača*. Dinamička alokacija memorije se u jeziku C ostvaruje pozivom funkcije "`malloc`" ili neke srodne funkcije iz biblioteke "`cstdlib`" (zaglavlje ove biblioteke u jeziku C zove se "`stdlib.h`"). 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 naslijeđene iz jezika C. Također treba napomenuti da je potreba za dinamičkom alokacijom memorije u svakodnevnim primjenama uveliko opala nakon što su u C++ uvedeni *dinamički tipovi podataka*, kao što su "`vector`", "`string`", itd. Međutim, dinamički tipovi podataka su izvedeni tipovi podataka, koji su definirani u standardnoj biblioteci jezika C++, i koji su implementirani upravo pomoću dinamičke alokacije memorije! Drugim riječima, da ne postoji dinamička alokacija memorije, ne bi bilo moguće ni kreiranje tipova poput "`vector`" i "`string`". Stoga, ukoliko želimo u potpunosti da ovladamo ovim tipovima podataka i da samostalno kreiramo *vlastite tipove podataka* koji su njima srodni, moramo shvatiti mehanizam koji stoji u pozadini njihovog funkcioniranja, a to je upravo dinamička alokacija memorije! Pored toga, vidjećemo kasnije da dinamička alokacija memorije može biti korisna za izgradnju drugih složenijih tipova podataka, koje nije moguće jednostavno svesti na postojeće dinamičke tipove podataka.

U jeziku C++, preporučeni način za dinamičku alokaciju memorije je pomoću operatora "**`new`**", koji ima više različitih oblika. U svom osnovnom obliku, iza njega treba da slijedi *ime nekog tipa*. On će tada potražiti u memoriji slobodno mjesto u koje bi se mogao smjestiti podatak navedenog tipa. Ukoliko se takvo mjesto *pronađe*, operator "**`new`**" će kao rezultat vratiti *pokazivač na pronađeno mjesto u memoriji*. Pored toga, pronađeno mjesto će biti označeno kao *zauzeto*, tako da se neće moći desiti da isti prostor u memoriji slučajno bude upotrijebljen 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 98 standard jezika C++ precizirao da u slučaju neuspjeha operator "**`new`**" baca izuzetak (naime, programeri su često zaboravljali ili su bili lijeni da nakon svake upotrebe operatora "**`new`**" provjeravaju da li je vraćena vrijednost možda nul-pokazivač, što je, u slučaju da alokacija ne uspije, moglo imati ozbiljne posljedice).

Osnovnu upotrebu operatora "**`new`**" najbolje je ilustrirati na konkretnom primjeru. Pretpostavimo da nam je data sljedeća deklaracija:

```
int *pok;
```

Ovom deklaracijom definirana je pokazivačka promjenljiva "`pok`" koja inicijalno ne pokazuje ni na šta konkretno, ali koja, u načelu, treba da pokazuje na *cijeli broj*. Stanje memorije nakon ove deklaracije možemo predstaviti sljedećom slikom (adrese su pretpostavljene proizvoljno):

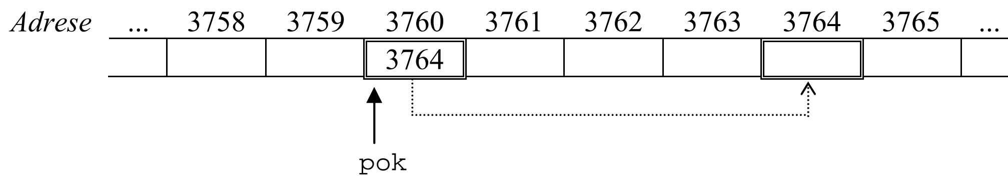


Ukoliko sad izvršimo naredbu

```
pok = 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 "pok". 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*, koji se može dodijeliti pokazivaču "pok", tako da će on pokazivati na adresu 3764. Stanje u memoriji će sada izgledati ovako:



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 daljnjeg sigurno neće biti iskorištena za neku drugu namjenu. Stoga je sasvim sigurno pristupiti njenom sadržaju putem pokazivača "pok". Znamo da se svaki dereferencirani pokazivač u potpunosti ponaša *kao promjenljiva* (preciznije, dereferencirani pokazivač je *l-vrijednost*) 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):

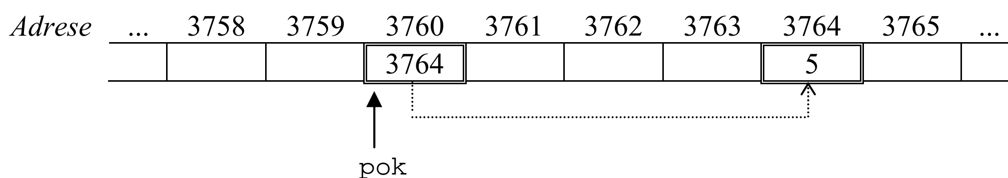
```
*pok = 5;
cout << *pok << endl;
*pok = 3 * *pok + 2;
(*pok)++;              // Oprez: ovo nije isto što i *pok++
cout << *pok << endl;
```

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 "pok" da pokazuje na nju:

```
pok = new int(5);
```

Sada će stanje u memoriji biti kao na sljedećoj slici:



Razumije se da se rezultat operatora "**new**" mogao odmah iskoristiti za inicijalizaciju pokazivača "pok" već prilikom njegove deklaracije, kao na primjer u sljedećim deklaracijama:

```
int *pok = new int(5);         // Ove dvije deklaracije su
int *pok(new int(5));          //   ekvivalentne...
```

Za dinamičku promjenljivu je, kao i za svaku drugu promjenljivu, moguće vezati *referencu*, čime možemo indirektno *dodijeliti ime* novostvorenoj dinamičkoj promjenljivoj. Tako, ukoliko je "`pok`" pokazivačka promjenljiva koja pokazuje na novostvorenu dinamičku promjenljivu, kao u prethodnom primjeru, moguće je izvršiti sljedeću deklaraciju:

```
int &dinamicka(*pok);
```

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 kasnije 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 *pok(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 zaglavlje biblioteke "`new`" u program):

```
try {
  int *pok(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

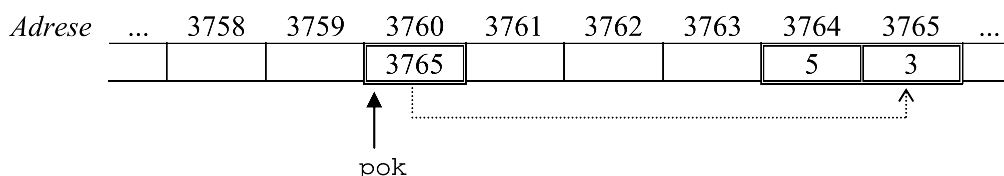
```
delete pok;
```

uništićemo dinamičku promjenljivu na koju pokazivač "pok" pokazuje. Pri tome je važno da razjasnimo šta se podrazumijeva pod tim *uništavanjem*. Pokazivač "pok" 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 "pok" pokazuje. Pokazivači koji pokazuju na objekte koji su iz bilo kojeg razloga *prestali da postoje* nazivaju se ***viseći pokazivači*** (engl. *dangling pointers*). Jedan od mogućih načina da kreiramo viseći pokazivač je eksplicitno uništavanje sadržaja na koji pokazivač pokazuje pozivom operatora "**delete**", ali postoje i mnogo suptilniji načini da stvorimo viseći pokazivač (npr. da iz funkcije kao rezultat vratimo pokazivač na neki lokalni objekat unutar funkcije, koji prestaje postojati po završetku funkcije). Treba se čuvati visećih pokazivača, jer su oni 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 *pok(new int);
*pok = 5;
pok = new int;
*pok = 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 "pok" 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 "pok" 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:



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. Mada je curenje memorije manje fatalno u odnosu greške koje nastaju uslijed visećih pokazivača, ono se također teško uočava i može da dovede do ozbiljnih problema. Razmotrimo na primjer sljedeću sekvencu naredbi:

```
int *pok;
for(int i = 1; i <= 30000; i++) {
  pok = new int;
  *pok = 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 "pok" 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 *pok(new int);
  *pok = 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, tako da svaki poziv funkcije "Curenje" stvara novu izgubljenu promjenljivu, odnosno prilikom svakog njenog poziva 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 *pok(new int(n));
  return pok;
}
```

Ova funkcija stvara novu dinamičku promjenljivu, inicijalizira je na vrijednost zadanu parametrom, i *vraća kao rezultat* pokazivač na novostvorenu promjenljivu (zanemarimo to što 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 *pok;
pok = Stvori(10);
cout << *pok;
delete pok;
```

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

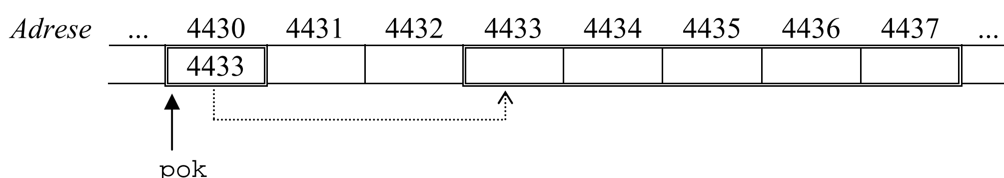
```
*Stvori(5) = 8;
```

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 indirektno kreirati nizovi čija veličina *nije unaprijed poznata* (ovaj mehanizam zapravo leži u pozadini funkcioniranja tipova poput tipa "vector"). 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. Pri tome, 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 "pok" deklarirana kao pokazivač na cijeli broj (kao i u dosadašnjim primjerima), tada će naredba

```
pok = 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 "pok" (u uglastoj zagradi nije morala biti konstanta 5, nego je mogao biti i proizvoljan cjelobrojni izraz). Na primjer, neka je pronađen prostor na adresi 4433, a neka se sama pokazivačka promjenljiva "pok" nalazi na adresi 4430. Tada memorijska slika nakon uspješnog izvršavanja prethodne naredbe izgleda kao na sljedećoj slici (radi jednostavnosti je pretpostavljeno da jedna cjelobrojna promjenljiva zauzima jednu memorijsku lokaciju, što u stvarnosti nije ispunjeno):



Ovako kreiranom "dinamičkom" nizu možemo pristupiti samo preko pokazivača. Međutim, kako se na pokazivače mogu primjenjivati operatori indeksiranja (podsjetimo se da se izraz "pok[n]" u slučaju kada je "pok" pokazivač interpretira kao "*(pok + 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++) pok[i] = 0;
```

Stoga, sa aspekta programera gotovo da nema nikakve razlike između korištenja običnih i dinamičkih nizova. Strogo rečeno, dinamički nizovi *nemaju imena* i njima se može pristupati samo preko pokazivača koji pokazuju na njihove elemente. Tako, ukoliko izvršimo deklaraciju poput

```
int *dinamicki_niz(new int[100]);
```

mi zapravo stvaramo *dva objekta*: dinamički niz od 100 cijelih brojeva koji *nema ime*, i pokazivač "dinamicki_niz" koji je inicijaliziran tako da pokazuje na *prvi element ovako stvorenog dinamičkog niza*. Međutim, kako promjenljivu "dinamicki_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 "dinamicki_niz" predstavlja dinamički niz (iako je prava istina da je ona zapravo pokazivač koji pokazuje na prvi element dinamičkog niza).

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 generičku funkciju:

```
template <typename NekiTip>
NekiTip *StvoriNizPopunjenNulama(int n) {
  NekiTip *pok(new NekiTip[n]);
  for(int i = 0; i < n; i++) pok[i] = 0;
  return pok;
}
```

Uz pomoć ovakve funkcije možemo stvarati inicijalizirane dinamičke nizove proizvoljnog tipa kojem se može dodijeliti nula. Na primjer,

```
double *dinamicki_niz(StvoriNizPopunjenNulama<double>(100));
```

Primijetimo da smo prilikom poziva funkcije eksplicitno morali navesti tip elemenata niza u šiljastim zagradama "<>". To je potrebno stoga što iz samog poziva funkcije nije moguće zaključiti šta tip "NekiTip" 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 NekiTip>
NekiTip *StvoriInicijaliziraniNiz(int n) {
  NekiTip *pok(new NekiTip[n]);
  for(int i = 0; i < n; i++) pok[i] = NekiTip();
  return pok;
}
```

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. osloboditi prostor koji je zauzimao) pomoću operatora "**`delete`**", samo uz neznatno drugačiju sintaksu u kojoj se koristi par uglastih zagrada. Tako, ukoliko pokazivač "`pok`" pokazuje na dinamički niz, uništavanje dinamičkog niza realizira se pomoću naredbe

```
delete[] pok;
```

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.. Mada postoje 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 kompajler *neće upozoriti* ne upotrijebimo li uglaste zagrade, s obzirom na činjenicu da kompajler *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 br_elemenata;
  cout << "Koliko želite brojeva? ";
  cin >> br_elemenata;
  int *niz(new int[br_elemenata]);
  cout << "Unesite brojeve:\n";
  for(int i = 0; i < br_elemenata; i++) cin >> niz[i];
  cout << "Niz brojeva ispisan naopako glasi:\n";
  for(int i = br_elemenata - 1; 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)

Treba napomenuti da se rezervacija memorije za čuvanje elemenata vektora i dekova također interno realizira pomoću dinamičke alokacije memorije i operatora "**`new`**". Drugim riječima, prilikom deklaracije vektora ili dekova također može doći do bacanja izuzetka (tipa "`bad_alloc`") ukoliko količina raspoložive memorije nije dovoljna da se kreira vektor ili dek odgovarajućeg kapaciteta. Zbog toga bi se i deklaracije vektora i dekova načelno trebale nalaziti unutar "**`try`**" bloka. Stoga, ukoliko bismo u prethodnom primjeru željeli izbjeći eksplicitnu dinamičku alokaciju memorije, i umjesto nje koristiti tip "`vector`", modificirani primjer trebao bi izgledati ovako:

```
try {
  int br_elemenata;
  cout << "Koliko želite brojeva? ";
  cin >> br_elemenata;
  vector<int> niz(br_elemenata);
  cout << "Unesite brojeve:\n";
  for(int i = 0; i < br_elemenata; i++) cin >> niz[i];
  cout << "Niz brojeva ispisan naopako glasi:\n";
  for(int i = br_elemenata; 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 ili deka (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). Kasnije ćemo detaljno razmotriti na koji je način ovo postignuto.

Slično običnim dinamičkim promjenljivim, dinamički nizovi se također uništavaju tek na završetku programa, ili eksplicitnom upotrebom operatora "**delete**[]". Stoga, pri njihovoj upotrebi također treba voditi računa da ne dođe do curenja memorije, koje može biti znatno ozbiljnije nego u slučaju običnih dinamičkih promjenljivih. 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 niko 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 nesvjesno 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 *pok(new int[n]);
  …
  G(n);
  …
  delete[] pok;
}
```

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 "pok", ali dinamički niz na čiji početak "pok" pokazuje nikada se ne uništava automatski, već samo eksplicitnim pozivom operatora "**delete**[]". Stoga, ukoliko se operator "**delete**[]" ne izvrši eksplicitno, zauzeta memorija neće biti oslobođena! Ovaj problem se može riješiti na sljedeći način:

```
void F(int n) {
  int *pok(new int[n]);
  …
  try {
    G(n);
  }
  catch(...) {
    delete[] pok;
    throw;
  }
  …
  delete[] pok;
}
```

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 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 komplicirana, ali da je potrebno preduzeti dosta mjera predostrožnosti da ne dođe do curenja memorije. Dodatne probleme izaziva činjenica da i sam operator "**new**" može također baciti 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 rogobatan, 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 pretpostavke, 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 eksplicitno voditi računa o brisanju zauzete memorije). Ovo rješenje zasniva se na činjenici da standard jezika C++ garantira da operator "**`delete`**" *ne radi ništa* ukoliko im se kao argument proslijedi *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. Uspput, nula je jedina cjelobrojna vrijednost koja se smije neposredno dodijeliti promjenljivim pokazivačkog tipa (i ona tada predstavlja nul-pokazivač). Radi bolje čitljivosti, u nekim bibliotekama jezika C++ definirana je konstanta "`NULL`" koja se obično koristi za inicijalizaciju pokazivačkih promjenljivih na nul-pokazivač, ali je sasvim legalno koristiti i običnu konstantu "0" (čime izbjegavamo potrebu za uključivanjem zaglavlja biblioteka koje definiraju konstantu "`NULL`").

Bitno je napomenuti da je ponašanje operatora "**`delete`**" *nedefinirano* (i može rezultirati krahom programa) ukoliko se primijeni na pokazivač koji ne pokazuje na prostor koji je zauzet u postupku dinamičke alokacije memorije. Naročito česta greška je primijeniti operator "**`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`**" 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 "`pok`" pokazuje na prvi element nekog dinamičkog niza, uništavanje tog niza najbolje je izvesti sljedećom konstrukcijom:

```
delete[] pok;
pok = 0;
```

Eksplicitnom dodjelom "`pok = 0`" zapravo postižemo dva efekta. Prvo, takvom dodjelom eksplicitno naglašavamo da pokazivač "`pok`" više ne pokazuje ni na šta. Drugo, ukoliko slučajno ponovo primijenimo operator "**`delete`**" na pokazivač "`pok`", neće se desiti ništa, jer je on sada nul-pokazivač.

U praksi se često javlja potreba i 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 *složenije pokazivačke tipove*, kao što su *pokazivači na nizove*, *nizovi pokazivača* i *pokazivači na pokazivače*, zavisno od primjene. Nije prevelik problem deklarirati ovakve složene pokazivačke tipove. Na primjer, razmotrimo sljedeće deklaracije:

```
int *pok1[30];
int (*pok2)[10];
int **p3;
```

U ovom primjeru, "`pok1`" je *niz od 30 pokazivača na cijele brojeve*, "`pok2`" je *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 "`pok3`" *pokazivač na pokazivač na cijele brojeve*. Obratimo pažnju između razlike u deklaraciji "`pok1`" koja predstavlja *niz pokazivača*, i deklaracije "`pok2`" koja predstavlja *pokazivač na niz*. Na izvjestan način, pojam "niz" prilikom deklaracija ima prioritet u odnosu na pojam "pokazivač", pa su u drugom primjeru zagrade bile neophodne da "izvrnu" ovaj prioritet. Ovakve konstrukcije se mogu usložnjavati unedogled (npr. principijelno je moguće napraviti *niz od 20 pokazivača na niz od 10 pokazivača na pokazivače na niz od 50 nizova od 30 realnih brojeva*), mada se potreba za tako složenim konstrukcijama javlja iznimno rijetko, i samo u veoma specifičnim primjenama. Čisto radi kurioziteta, pokažimo kako bi izgledala deklaracija ovakvog pokazivača:

```
double (**(*p[20])[10])[50][30];
```

Interesantno je navesti pravilo koje olakšava čitanje i konstrukciju složenih pokazivačkih tipova. Značenje promjenljivih najlakše čitamo tako što u deklaraciji promjenljive pođemo od njenog imena pa prvo čitamo sve što se eventualno nalazi iza imena promjeljive redom slijeva nadesno, sve do prve zatvorene male zagrade ili do kraja, a zatim čitamo sve što se eventualno nalazi ispred imena promjenljive redom zdesna nalijevo, sve do prve otvorene male zagrade ili do početka. Kada tako pročitamo sve što se nalazilo unutar malih zagrada (ako ih je bilo) nastavljamo dalje čitanje nadesno od mjesta gdje smo stali (dakle na prvoj zatvorenoj maloj zagradi) sve do sljedeće zatvorene male zagrade ili do kraja, a nakon toga nastavljamo dalje čitanje nalijevo od mjesta gdje smo stali (na otvorenoj maloj zagradi) sve do sljedeće otvorene male zagrade ili do početka. Takvo "cik cak" čitanje nastavljamo dalje sve dok ne dostignemo kraj odnosno početak deklaracije. Razumijevanje ovog pravila lako možete provjeriti na prethodnoj deklaraciji.

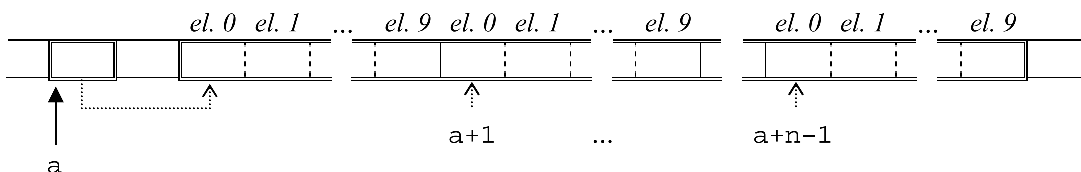
U nastavku izlaganja zanemarićemo egzistenciju ovako složenih pokazivačkih tipova i ograničićemo se samo na pokazivače na nizove, nizove pokazivača i pokazivače na pokazivače. Potreba za ovakvim konstrukcijama relativno se često javlja u izvjesnim specijalnim okolnostima. Klasični primjer situacije u kojoj se javlja potreba za složenim pokazivačkim tipovima je dinamička alokacija *višedimenzionalnih nizova*. Opći slučaj dinamičke alokacije za višedimenzionalne nizove sa većim brojem dimeznija je vrlo složen problem u koji se nećemo upuštati. Razmotrimo stoga 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, sasvim je lako ostvariti dinamičku alokaciju dvodimenzionalnog niza kod kojeg je *druga dimenzija unaprijed poznata*, npr. matrice čiji je broj kolona unaprijed poznat (situacija u kojpk druga dimenzija nije apriori poznata je složenija, kao što ćemo uskoro vidjeti). Pretpostavimo 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:

```
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 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. Operator "**new**" ima samo jedan parametar, mada on dopušta i drugi par uglastih zagrada, samo što se u njemu *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*.

Razmotrimo malo detaljnije šta smo ovom konstrukcijom postigli. Ovdje smo pomoću operatora "**new**" alocirali dinamički niz od "`n`" elemenata, koji su sami za sebe tipa "niz od 10 cijelih brojeva" (što nije ništa drugo nego dvodimenzionalni niz sa "`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 čiji su elementi tog tipa. Primijetimo da je u ovom slučaju pokazivač "`a`" *pokazivač na* (*čitav*) *niz*, odnosno *pokazivač na niz kao cjelinu.* 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 dereferenciranje i 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 nizovi* od 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 od 10 cijelih brojeva", to nakon njegovog dereferenciranja dobijamo element tipa "niz od 10 cijelih brojeva" 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, adresa na koju "`a`" pokazuje uvećava se za *čitavu dužinu nizovnog tipa* "niz od 10 cijelih brojeva". 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:



U gore prikazanoj konstrukciji, za inicijalizaciju složene pokazivačke promjenljive "`a`" iskorištena je sintaksa koja koristi znak "`=`". Razumljivo je da je moguće koristiti i konstruktorsku sintaksu sa zagradama, koja bi u konkretnom primjeru izgledala ovako:

```
int (*a)[10](new int[n][10]);
```

Međutim, u ovom primjeru, upotreba konstruktorske sintakse na samom početku izlaganja vjerovatno bi zbunila početnika zbog mnoštva zagrada, tako da je ona namjerno izbjegnuta u početnom razmatranju.

Prethodno 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. Strogo rečeno, takva dinamička alokacija zbog nekih tehničkih razloga zapravo *nije ni moguća*, ali se može veoma 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[1] = 5;             // Indirektno mijenja promjenljivu "br_2"
cout << niz_pok[2];          // Ovo ispisuje adresu...
cout << *niz_pok[2];         // A ovo sadržaj dinamičke promjenljive...
delete niz_pok[2];           // Briše dinamičku promjenljivu
delete[] niz_pok[4];         // Briše dinamički niz
niz_pok[3][5] = 100;         // Vrlo interesantno...
```

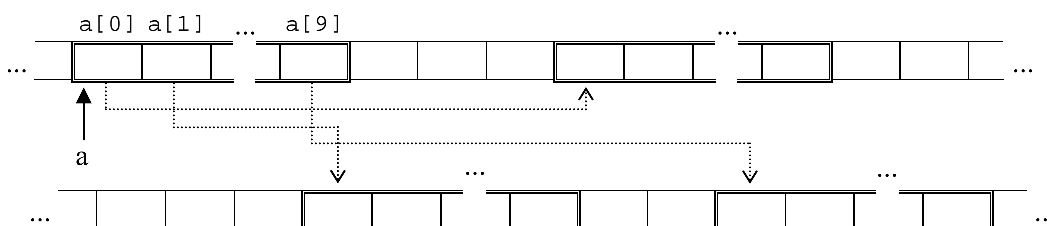
Posljednja naredba je posebno interesantna, jer pokazuje da se nizovima pokazivača može pristupati kao da se radi o *dvodimenzionalnim nizovima* (pri čemu je takav pristup smislen samo ukoliko odgovarajući element niza pokazivača pokazuje na element nekog drugog niza, koji može biti i dinamički kreiran). 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)".

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*. Jedan od često korištenih načina za to je da se prvo formira *niz pokazivača*, a zatim se pokazivačima u tom nizu dodijele *adrese 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 "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 posmatraču sigurno neće promaći upadljiva sličnost između opisane tehnike za dinamičku alokaciju dvodimenzionalnih nizova i kreiranja nizova ili vektora čiji su elementi vektori. Kasnije ćemo vidjeti da u oba slučaja u pozadini zapravo stoji isti mehanizam.

Kada smo govorili o vektorima čiji su elementi vektori, govorili smo o mogućnosti kreiranja struktura podataka nazvanih *grbave matrice*, koji predstavljaju strukture koje po načinu upotrebe liče na obične matrice, 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.

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. osloboditi 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];
```

Razmotrimo kako sada 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. To možemo uraditi recimo ovako:

```
int **a(new int*[n]);
for(int i = 0; i < n; i++) a[i] = new int[m];
```

Obratite pažnju na zvjezdicu koja govori da alociramo niz *pokazivača* na cijele brojeve a ne niz cijelih brojeva. 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). Naravno, 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;
```

Vidjeli smo da se kod opisanog postupka dinamičke alokacije dvodimenzionalnih nizova čija druga dimenzija nije poznata svaki od redova tako kreiranog niza alocira posebno, što za posljedicu ima da individualni redovi tako kreiranog niza nisu nužno kontinualno raspoređeni u memoriji, odnosno mogu biti razbacani svuda po memoriji. Stoga, opisani postupak dinamičke alokacije nazivamo *fragmentirana alokacija*. Nekontinualni razmještaj individualnih redova rijetko predstavlja problem u praksi. Međutim, moguće je izvršiti i takvu dinamičku alokaciju dvodimenzionalnih nizova koja garantira da će se njihovi redovi nalaziti kontinuirano u memoriji, redom jedan za drugim. U tom slučaju govorimo o *kontinualnoj alokaciji*. Da bismo izvršili kontinualnu alokaciju dinamičkog dvodimenzionalnog niza sa "n" redova i "m" kolona, umjesto da individualno kreiramo "n" dinamičkih nizova od kojih svaki ima "m" elemenata (što daje ukupno "n * m" elemenata), kreiraćemo *jedan* dinamički niz od "n * m" elemenata, koji predstavlja prostor koji na jednom mjestu čuva sve elemente dvodimenzionalnog niza, red po red. Nakon toga ćemo prvom pokazivaču u nizu pokazivača dodijeliti adresu prvog elementa alociranog prostora, a svakom sljedećem pokazivaču adresu koja se nalazi "m" elemenata ispred adrese na koju pokazuje prethodni pokazivač. Na taj način, svaki pokazivač u nizu pokazuje na prostor koji je dovoljan za smještanje tačno "m" elemenata jednog reda, a da pri tome ne dođe do konflikta sa sadržajem ostalih redova elemenata. Konkretno, to bi se moglo izvesti recimo ovako:

```
int **a(new int*[n]);
a[0] = new int[n * m];
for(int i = 1; i < n; i++) a[i] = a[i – 1] + m;
```

Jasno je da je na sličan način moguće izvršiti i kontinualnu alokaciju grbavih nizova. Ovaj način alokacije ima izvjesne prednosti nad fragmentiranom alokacijom. Na prvom mjestu, kontinualna alokacija omogućava legalne trikove sa upotrebom pokazivačke aritmetike koji se zasnivaju na kontinualnom razmještaju redova u memoriji. Pored toga, kontinualno alocirane dvodimenzionalne dinamičke nizove je jednostavnije uništiti nego u slučaju fragmentirane alokacije. Zaista, da bismo uništili kontinualno alocirani dvodimenzionalni dinamički niz, dovoljno je izvršiti

```
delete[] a[0]; delete[] a;
```

S druge strane, osnovni nedostatak kontinualne alokacije leži u činjenici da je slobodni prostor u memoriji veoma često također fragmentiran, tako da postoji mogućnost da se ne može pronaći prostor za smještanje "n * m" elemenata, iako se može naći "n" prostora (razbacanih po memoriji) dovoljnih za smještanje "m" elemenata. Drugim riječima, ukoliko su "n" i "m" veliki, kontinualna alokacija ima znatno više šansi za neuspjeh u odnosu na fragmentiranu alokaciju.

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, uzmemo li 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 komplicirano zbog dvostruke indirekcije, ali oni nisu ništa kompliciraniji 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 se u jeziku C dvojni pokazivači intenzivno koriste i u nekim situacijama u kojima je u jeziku C++ prirodnije upotrijebiti *reference na pokazivač* (tj. reference vezane za neki pokazivač). Na primjer, pomoću deklaracije

```
int *&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`". 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, u skladu sa ranije navedenim pravilom o čitanju složenijih deklaracija, kada čitanje vršimo od imena promjenljive *zdesna nalijevo*, jasno vidimo da "`ref`" zaista 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 izmijeni sadržaj samog pokazivača (a ne objekta na koji pokazivač pokazuje). Na primjer, sljedeća funkcija vrši razmjenu dva pokazivača koji su joj proslijeđeni kao stvarni parametri (zanemarimo ovom prilikom činjenicu da će generička funkcija "`Razmijeni`" koju smo razmatrali na prethodnim predavanjima sasvim lijepo razmijeniti i dva pokazivača, pri čemu će se u postupku dedukcije tipa izvesti zaključak da nepoznati tip predstavlja tip pokazivača na realne brojeve):

```
void RazmijeniPokazivace(double *&x, double *&y) {
  double *pomocna(x);
  x = y; y = pomocna;
}
```

Kako u jeziku C ne postoje reference, sličan efekat se može 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), pri čemu nakon uzimanja adrese dobijamo dvojni pokazivač. 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. Naročito treba paziti da u slučaju da alokacija ne uspije do kraja, prije nego što nastavimo dalje išta raditi neophodno je ukloniti ono što je u postupku alokacije bilo alocirano (u suprotnom, alocirani prostor neće niko osloboditi, tako da će doći do curenja memorije). Postoji više načina da to uradimo. Recimo, jedan od načina da korektno obavimo fragmentiranu dinamičku alokaciju matrice realnih brojeva sa "`n`" redova i "`m`" kolona može izgledati recimo ovako:

```
try {
  double **a(new double*[n]);
  for(int i = 0; i < n; i++) a[i] = 0;
  try {
    for(int i = 0; i < n; i++) a[i] = new double[m];
  }
  catch(...) {
    for(int i = 0; i < n; i++) delete[] a[i];
    delete[] a;
    throw;
  }
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
```

U ovom primjeru, prije same dinamičke alokacije redova matrice, svim pokazivačima na redove matrice su prvo dodijeljeni nul-pokazivači. Time smo postigli da u slučaju da neke od alokacija ne uspiju, u "**catch**" bloku ne moramo voditi računa koje su alokacije redova uspjele a koje nisu, nego prosto primjenjujemo operator "**delete**" na sve pokazivače, znajući da on ne radi ništa ukoliko mu se kao parametar ponudi nul-pokazivač. Ovo je trik koji se često koristi kod dinamičke alokacije složenih i fragmentiranih struktura podataka. Drugi način za rješavanje problema neuspješnih alokacija je da brojimo koliko je bilo uspješnih alokacija, i da u slučaju neuspjeha primijenimo operator "**delete**" samo na one pokazivače koji pokazuju na uspješno alocirane blokove. Mada je ovaj način neznatno efikasniji, njega je teže generalizirati na slučajeve još složenijih alokacija od ovdje opisanih. Za slučaj klasične fragmentirane dinamičke alokacije dvodimenzionalnih nizova, ovaj način bi se mogao implementirati recimo na sljedeći način:

```
try {
  double **a(new double*[n]);
  int brojac(0);
  try {
    while(brojac < n) a[brojac++] = new double[m];
  }
  catch(...) {
    for(int i = 0; i < brojac; i++) delete[] a[i];
    delete[] a;
    throw;
  }
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
```

Ukoliko umjesto fragmentirane koristimo kontinualnu alokaciju, situacija je jednostavnija zbog znatno manjeg broja alociranih blokova. Tako se kontinualna dinamička alokacija matrice sa "n" redova i "m" kolona uz vođenje računa o uspješnosti alokacija može izvesti recimo ovako:

```
try {
  double **a(new double*[n]);
  try {
    a[0] = new int[n * m];
    for(int i = 1; i < n; i++) a[i] = a[i – 1] + m;
  }
  catch(...) {
    delete[] a[0]; delete[] a;
    throw;
  }
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
```

Vidimo da dinamička alokacija matrica proizvoljnih dimenzija može biti mukotrpna ukoliko vodimo računa o mogućim memorijskim 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 obavljaju dinamičko stvaranje odnosno uništavanje matrica proizvoljnih dimenzija*, koje na sebe preuzimaju opisani postupak (naravno, još jednostavnije rješenje je koristiti vektore vektora, ali ovdje želimo da naučimo mehanizme nižeg nivoa koji leže u pozadini rada takvih tipova podataka). Sljedeći primjer pokazuje kako bi takve funkcije mogle izgledati za slučaj fragmentirane alokacije (funkcije su napisane kao generičke funkcije, tako da omogućavaju stvaranje odnosno uništavanje matrica čiji su elementi proizvoljnog tipa):

```
template <typename TipElemenata>
void UnistiMatricu(TipElemenata **mat, int broj_redova) {
  if(mat == 0) return;
  for(int i = 0; i < broj_redova; i++) delete[] mat[i];
  delete[] mat;
}

template <typename TipElemenata>
TipElemenata **StvoriMatricu(int broj_redova, int broj_kolona) {
  TipElemenata **mat(new TipElemenata*[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 TipElemenata[broj_kolona];
  }
  catch(...) {
    UnistiMatricu(mat, broj_redova);
    throw;
  }
  return mat;
}
```

Naravno, trivijalno je prepraviti ove funkcije da umjesto fragmentirane koriste kontinualnu alokaciju. Međutim, u ovim funkcijama potrebno je obratiti pažnju na nekoliko detalja. 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 ponaša operator "**`delete`**". Ova konzistencija će nam kasnije biti od velike koristi. Konačno, funkcija "`StvoriMatricu`" 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;
double **a(0), **b(0);
try {
  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 prenositi 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 "`<>`" eksplicitno specificirati tip elemenata matrice, s obzirom da se tip elemenata ne može odrediti dedukcijom tipa iz parametara funkcije. Ovaj problem možemo izbjeći ukoliko modificiramo funkciju "`StvoriMatricu`" tako da kao jedan od parametara prihvata dvojni pokazivač za pristup elementima matrice. Tako modificirana funkcija treba da u odgovarajući parametar *smjesti* adresu dinamički alocirane matrice (umjesto da tu adresu *vrati 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 prenositi 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 možete sami uraditi kao korisnu 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 dinamičoj 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).

Predavanje 6.

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-terminiranih 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. Mnogo je bolje kreirati *niz dinamičkih stringova*, odnosno niz objekata tipa "string", 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 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 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.

Nizovi pokazivača su zaista korisne strukture podataka, i njihove primjene u jeziku C++ su višestruke, tako da ćemo se sa njima intenzivno susretati kasnije. 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 dinamičkog niza ili vektora 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 različit broj znakova. Sljedeći primjer ilustrira kako to najbolje možemo izvesti:

```
int broj_recenica;
cout << "Koliko želite unijeti rečenica: ";
cin >> broj_recenica;
cin.ignore(10000, '\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 pomoću funkcije "strcpy". Postupak se ponavlja za svaku unesenu rečenicu. Primijetimo da je pomoćni niz "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.

Kao korisna alternativa nizovima čiji su elementi pokazivači, mogu se koristiti *vektori čiji su elementi pokazivači*, čime izbjegavamo potrebu za eksplicitnom alokacijom i dealokacijom memorije za potrebe dinamičkog niza. Sljedeći primjer prikazuje kako bi prethodni primjer izgledao uz korištenje vektora čiji su elementi pokazivači (obratite pažnju kako se deklarira vektor čiji su elementi pokazivači na znakove):

```
int broj_recenica;
cout << "Koliko želite unijeti rečenica: ";
cin >> broj_recenica;
cin.ignore(10000, '\n');
vector<char*> recenice(broj_recenica);
for(int i = 0; i < broj_recenica; i++) {
  char radni_prostor[1000];
  cin.getline(radni_prostor, 1000);
  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;
```

Opisana rješenja rade veoma dobro. Ipak, ona su znatno komplikovanija od rješenja koja se zasnivaju na nizu ili vektoru čiji su elementi tipa "string". Prvo, u prikazanim primjerima moramo se eksplicitno brinuti o alokaciji memorije za svaku rečenicu, dok na kraju moramo eksplicitno uništiti alocirani prostor. U slučaju korištenja tipa "string" sve se odvija potpuno automatski (mada je, interno posmatrano, niz ili vektor čiji su elementi tipa "string" u memoriji organiziran na vrlo sličan način kao i opisani grbavi niz znakova, odnosno kao niz ili vektor 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, sa aspekta efikasnosti, rješenje zasnovano na upotrebi 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), o čemu ćemo govoriti u nastavku ovog teksta.

Iz izloženih primjera vidimo da grbavi nizovi mogu biti izuzetno efikasni sa aspekta utroška memorije. Međutim, pri radu sa njima neophodan je izvjestan oprez zbog činjenice da njegovi redovi mogu imati različite dužine. Pretpostavimo, recimo, da je potrebno *sortirati* uneseni niz rečenica iz prethodnog primjera. 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);
```

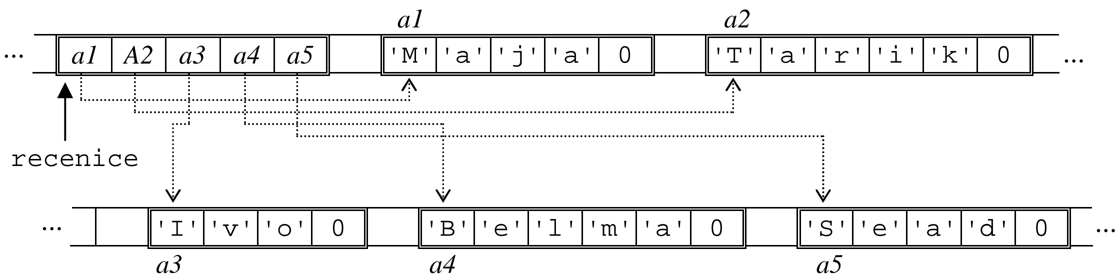
S druge strane, ukoliko je niz "`recenice`" izveden kao niz elemenata tipa "`string`", razmjenu bismo vjerovatno izvršili ovako (na isti način kao kad razmjenjujemo recimo elemente niza brojeva):

```
string pomocna(recenice[i]);
recenice[i] = recenice[j]; recenice[j] = pomocna;
```

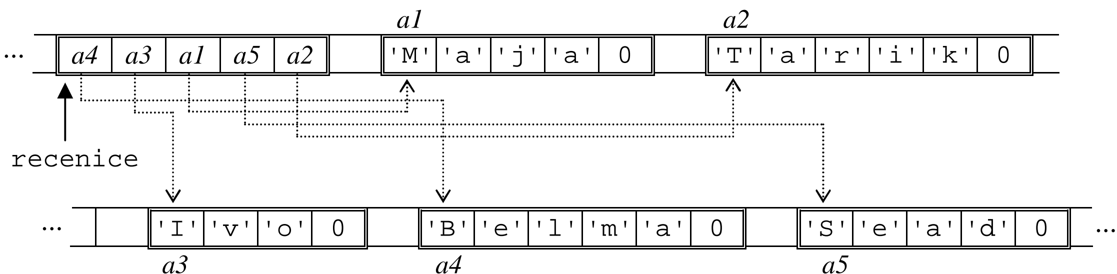
Šta bismo, međutim, trebali uraditi ukoliko koristimo niz "`recenice`" koji je organiziran kao *grbavi niz znakova*, odnosno niz pokazivača na početke svake od rečenica? Jasno je da varijanta u kojoj bismo kopirali čitave rečenice pomoću funkcije "`strcpy`" nije prihvatljiva, s obzirom da smo za svaku 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 "Maja", "Tarik", "Ivo", "Belma" i "Sead" (on zapravo sadrži *pokazivače* na mjesta 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):



Nakon izvršenog sortiranja, u kojem se vrši razmjena pokazivača, situacija u memoriji izgledala bi ovako:



Vidimo da ukoliko pokazivače čitamo redom, oni pokazuju respektivno na imena "Belma", "Ivo", "Maja", "Sead" i "Tarik", 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 bismo poređenje rečenica (sa ciljem utvrđivanja da li su u ispravnom poretku) morali vršiti pomoću funkcije "`strcmp`" iz biblioteke "`cstring`" ili na neki sličan način, jer bi obična upotreba relacionih operatora "<" ili ">" upoređivala *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 (ili vektora pokazivača) u jeziku C++. Kada god je potrebno kreirati nizove (ili vektore) 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 izlaganjima 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!

U programiranju se često javlja potreba da neka funkcija poziva neku drugu funkciju, ali pri čemu nije unaprijed poznato koju tačno funkciju treba pozvati, nego se odluka o tome koju funkciju treba prva funkcija pozvati donosi tek na mjestu poziva ove funkcije. Jezici C i C++ omogućavaju rješavanje ovog problema tako što se nekoj funkciji prilikom poziva može kao parametar poslati *ime neke druge funkcije*. Da bismo uvidjeli potrebu za opisanim scenariom, 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 "`a`" i "`b`" i tačnost "`eps`". 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 pronađemo 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 je potvrdan. Naime, unutar funkcije "`NulaFunkcije`", umjesto poziva konkretne funkcije "`F`", treba uvesti *formalni parametar* koji je tipa *funkcije koja prima jedan realan argument, a vraća realni broj kao rezultat* (koji ćemo, recimo, označiti sa "`f`"). Prilikom poziva funkcije, kao stvarni parametar će se zadavati *ime konkretne funkcije koja treba biti upotrijebljena na mjestu formalnog parametra* "`f`". 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;
  }
```

Primijetimo da je u deklaraciji formalnog parametra "`f`" izostavljeno ime formalnog parametra same funkcije "`f`", što je dozvoljeno uraditi, s obzirom da nam ono nizašta ne treba. Pretpostavimo sada 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 (0,1) je " << NulaFunkcije(G, 0, 1) << endl;
cout << "Nula sin(x) na (3,4) je " << NulaFunkcije(sin, 3, 4) << 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 formalnog parametra "`f`".

Da bismo objasnili kakav tačno mehanizam stoji u pozadini mogućnosti prenosa funkcija kao parametara drugim funkcijama, podsjetimo se malo šta su prototipovi funkcija. Znamo da u tipičnim situacijama svaka funkcija koju pozivamo treba biti definirana *prije mjesta njenog poziva*. Prototipovi funkcija nam omogućavaju da funkcije definiramo *iza mjesta poziva funkcije*, pa čak i u nekoj *drugoj programskoj datoteci*. Prototipovi funkcija informiraju kompajler koliko parametara prihvata funkcija, kakvi su tipovi parametara, i koji je tip povratne vrijednosti funkcije. Oni se sastoje samo od zaglavlja funkcije, bez njenog tijela (umjesto tijela stavlja se tačka-zarez). Na primjer, sljedeća deklaracija

```
int fn(int x, int y);
```

predstavlja prototip funkcije "`fn`" koja prima dva cjelobrojna argumenta, i vraća cjelobrojnu vrijednost kao rezultat. S obzirom da su u prototipovima imena argumenata posve nebitna, mogu se izostaviti, tako da smo isti prototip mogli napisati i ovako:

```
int fn(int, int);
```

Uvođenje prototipova omogućava sigurno pozivanje funkcija koje su definirane *nakon mjesta poziva*, pod uvjetom da se prije mjesta poziva negdje nalazi deklariran njen prototip. Bitno je istaći da su

prototipovi mnogo značajniji u jeziku C++ nego u jeziku C. Naime, kompajler za jezik C će *prihvatiti* poziv funkcije koja nije definirana prije mjesta poziva čak i bez njenog prototipa, ali će pri tome podrazumijevati da su svi parametri tipa "**int**", kao i da je povratna vrijednost tipa "**int**". Ukoliko se kasnije naiđe na definiciju funkcije koja odstupa od ovih pretpostavki prijavljuje se greška. To i ne bi bilo toliko loše da nema jednog problema. Naime, ukoliko se definicija funkcije uopće ne nalazi u istoj programskoj datoteci kao i poziv funkcije, greška se ne može prijaviti, s obzirom da se prevođenje različitih programskih datoteka obavlja posve neovisno, odnosno svaka programskih datoteka se prevodi posve neovisno od drugih programskih datoteka. Kao posljedicu, u takvim situacijama možemo dobiti *loše preveden program*, odnosno program koji je preveden u izvršni kod, ali *ne radi ispravno*. U jeziku C++ nisu željeli dozvoliti ovakvu nedosljednost, te je uvedeno pravilo: poziv neke funkcije *uvijek dovodi do prijave greške* ukoliko prije toga nije viđena niti definicija funkcije, niti njen prototip. Ovo je još jedna od minornih nekompatibilnosti između jezika C i C++, ali je znatno doprinijela sigurnosti.

Sada smo u mogućnosti da objasnimo kakav tačno mehanizam stoji u pozadini prenosa funkcija kao parametara drugim funkcijama. Za tu svrhu koriste se posebni objekti nazvani *pokazivači 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". Ovo je u potpunosti u skladu sa već opisanim načinom "cik cak" čitanja deklaracija pokazivačkih tipova.

Kao što pokazivači na ma koji objekat interno sadrže *adresu tog objekta*, tako pokazivači na funkcije u sebi sadrže *adresu funkcije*, što je zapravo *adresa mašinskog programa* koji izvršava datu funkciju. Na pokazivače na funkcije se ne može primijenjivati 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 pokazivač pokazuje. Dereferenciranjem pokazivača na funkciju kao rezultat se dobija *funkcija na koji oni pokazuju* (što je uostalom i logično). Kako je jedina operacija koju možemo izvesti sa funkcijama njihov *poziv*, jedina stvar koju možemo uraditi sa dereferenciranim pokazivačem na funkcije je pozvati odgovarajuću funkciju. 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 x, int y) {
  if(x < y) return x;
  else return y;
}

int Max(int x, int y) {
  if(x > y) return x;
  else return y;
}
```

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 indirektan pristup sadržaju promjenljivih i tokom izvršavanja programa mogu pokazivati na različite promjenljive. Međutim, sintaksa dopušta da se na pokazivač na funkcije *direktno primijeni sintaksa poziva funkcije*, kao da se radi o funkciji a ne o pokazivaču (naravno, prilikom takvog poziva, poziva se funkcija na koju pokazivač pokazuje). Stoga se pokazivači na funkciju gotovo nikada ne dereferenciraju (osim ukoliko želimo sintaksno jasnije naglasiti da se radi o pokazivaču na funkciju a ne o funkciji), nego se na njih direktno primjenjuje sintaksa poziva funkcije, kao u sljedećoj konstrukciji:

```
pok_na_fn = Min;
cout << "Minimum je: " << pok_na_fn(a, b);
pok_na_fn = Max;
cout << "Maksimum je: " << pok_na_fn(a, b);
```

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 zagrada 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. Upravo zbog ove automatske konverzije, nije potrebno koristiti adresni operator "&" kada želimo uzeti adresu neke funkcije i smjestiti je u pokazivač na funkciju (mada upotreba ovog operatora nije ni zabranjena". U suštini, ukoliko je "`f`" neka funkcija, tada izrazi "`f`" (bez zagrada za poziv funkcije) i "`&f`" imaju identično značenje.

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 započeti izvršavanje mašinskog programa na nepredvidljivoj adresi i uz nepredvidljive posljedice.

Sad možemo precizno reći kako je omogućeno prenos funkcija kao parametara u druge funkcije. Svaki put kada deklariramo formalni parametar neke funkcije koji je tipa funkcije, taj formalni parametar zapravo postaje *pokazivač na funkciju*. Sintaksno je dozvoljeno, ali ne i neophodno, da eksplicitno deklariramo formalni parametar kao pokazivač na funkciju. Drugim riječima, definicija funkcije "`NulaFunkcije`" mogla je izgledati i ovako:

```
double NulaFunkcije(double (*f)(double), double a, double b,
  double eps = 1e-7) {
    // Tijelo funkcije ostaje isto kao u prethodnom slučaju
}
```

Recimo još nekoliko riječi o *nizovima pokazivača na funkcije*. Ovakve egzotične tvorevine također mogu imati veoma interesantne primjene. Na primjer, sljedeća deklaracija

```
double (*f[10])(int);
```

deklarira "`f`" kao niz od 10 pokazivača na funkcije, koje prihvataju jedan cijeli broj kao argument, a vraćaju realni broj kao rezultat (primijetimo da je ova deklaracija u potpunosti u skladu sa "cik cak" pravilom o interpretaciji deklaracija pokazivačkih tipova). 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 korisnici će vjerovatno uočiti i mogućnosti za brojne druge primjene pored ovdje navedenih primjena.

Izlaganje o pokazivačima na funkcije završićemo objašnjenjem *kako deklarirati funkciju koja kao svoj rezultat vraća pokazivač na funkciju* (takve funkcije susreću se prilično rijetko, ali i za njima se kadkad ukaže potreba). Na primjer, neka je potrebno deklarirati funkciju koja prima cijelobrojni parametar "n", a vraća kao rezultat pokazivač na funkciju koja prima dva realna parametra, a ne vraća nikakav rezultat. Korisno bi bilo da sami probate doći do tačnog odgovora, koji glasi:

```
void (*f(int n))(double, double);
```

Ukoliko niste uspjeli sami doći do tačnog odgovora, pažljivo analizirajte ponuđeni odgovor i vidjećete da je on u potpunosti u skladu sa "cik cak" konvencijom o interpretaciji deklaracija pokazivačkih tipova.

Razvoj programa u jeziku C++ osjetno je olakšan uvođenjem standardne biblioteke "algorithm". Ova biblioteka sadrži mnoštvo korisnih funkcija za kojima se u svakodnevnom programiranju često javlja potreba. Veliki broj ovih funkcija su posve jednostavne, i ne bi ih bilo nikakav problem samostalno napisati, mada među njima ima i dosta složenih funkcija. 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 toplu vodu". Također, sve spomenute funkcije napisane su veoma optimalno, tako da će vjerovatno raditi efikasnije nego analogne funkcije koje bismo eventualno napisali sami.

Većina funkcija iz biblioteke "algorithm" predviđena je za manipulaciju sa nizovima, vektorima, stringovima, dekovima i drugim kontejnerskim strukturama koje posjeduje jezik C++, a o kojima ne možemo govoriti zbog nedostatka prostora, a mogu raditi čak i sa kontejnerskim strukturama podataka koje definira sam programer, pod uvjetom da one zadovoljavaju određene uvjete (odnosno, da predstavljaju modele odgovarajućeg koncepta). Ove funkcije uglavnom prihvataju kao parametre pokazivače i poopćenja pokazivača poznata pod nazivom *iteratori*, pri čemu se pokazivači mogu koristiti ukoliko koristimo nizove, vektore i stringove, dok se u slučaju drugih kontejnerskih struktura podataka moraju koristiti isključivo iteratori (pri čemu se iteratori mogu koristiti i sa vektorima, tako da vektori podržavaju i pokazivače i iteratore, odnosno pokazivači i iteratori se pri radu sa vektorima ponašaju istovjetno).

Prikazaćemo prvo neke od najjednostavnijih i najčešće korištenih funkcija iz ove biblioteke. 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*, a ne tačno na posljednji element. Također, za one koje to zanima, u svim pomenutim funkcijama, na svim mjestima gdje se prihvaćaju pokazivači, biće kao parametri prihvaćeni i iteratori:

`copy(`*p1*`,` *p2*`,` *p3*`)`	Kopira elemente između pokazivača *p1* i *p2* na lokaciju određenu pokazivačem *p3* (usput 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 ove funkcije i funkcije "`copy`" može biti važna ukoliko se izvorni i odredišni blok preklapaju.
`fill(`*p1*`,` *p2*`,` *v*`)`	Popunjava sve elemente između pokazivača *p1* i *p2* sa vrijednošću *v*.
`fill_n(`*p*`,` *n*`,` *v*`)`	Popunjava *n* elemenata počev od pokazivača *p* nadalje sa vrijednošću *v*.
`swap_ranges(`*p1*`,` *p2*`,` *p3*`)`	Razmjenjuje blok elemenata između pokazivača *p1* i *p2* sa blokom elemenata na koji pokazuje pokazivač *p3* (usput vraća kao rezultat pokazivač na poziciju tačno iza bloka na koji pokazuje *p3*).
`reverse(`*p1*`,` *p2*`)`	Izvrće blok elemenata između pokazivača *p1* i *p2* u obrnuti poredak, tako da prvi element postane posljednji, itd.
`reverse_copy(`*p1*`,` *p2*`,` *p3*`)`	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.
`replace(`*p1*`,` *p2*`,` *v1*`,` *v2*`)`	Zamjenjuje sve elemente između pokazivača *p1* i *p2* koji imaju vrijednost *v1* sa elementima sa vrijednošću *v2*.
`replace_copy(`*p1*`,` *p2*`,` *p3*`,` *v1*`,` *v2*`)`	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*.
`rotate(`*p1*`,` *p2*`,` *p3*`)`	"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).
`rotate_copy(`*p1*`,` *p2*`,` *p3*`,` *p4*`)`	Isto kao "`rotate`", ali kopira rotiranu verziju bloka elemenata na poziciju određenu pokazivačem *p4*, dok sam izvorni blok ostaje neizmijenjen.
`count(`*p1*`,` *p2*`,` *v*`)`	Vraća kao rezultat broj elemenata između pokazivača *p1* i *p2* koji imaju vrijednost *v*.
`equal(`*p1*`,` *p2*`,` *p3*`)`	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`**".
`min_element(`*p1*`,` *p2*`)`	Vraća kao rezultat pokazivač na najmanji element u bloku između pokazivača *p1* i *p2*.
`max_element(`*p1*`,` *p2*`)`	Vraća kao rezultat pokazivač na najveći element u bloku između pokazivača *p1* i *p2*.
`find(`*p1*`,` *p2*`,` *v*`)`	Vraća kao rezultat pokazivač na prvi 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*`)`	Reorganizira elemente bloka između pokazivača *p1* i *p2* tako što uklanja sve elemente koji imaju vrijednost *v*. Kao rezultat vraća pokazivač takav da u bloku između pokazivača *p1* i vraćenog pokazivača niti jedan element neće imati vrijednost *v*. Ukoliko niti jedan element nije imao vrijednost *v*, funkcija vraća *p2* kao rezultat. Ova funkcija ne mijenja sam broj elemenata niza niti vektora (broj elemenata niza nije ni moguće mijenjati), nego samo premješta elemente da obezbijedi traženo svojstvo. Sadržaj bloka između vraćenog pokazivača i pokazivača *p2* na kraju je nedefiniran.
`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*`)`	Reorganizira elemente bloka između pokazivača *p1* i *p2* tako što uklanja sve elemente koji su jednaki elementima koje im prethodi. Kao rezultat vraća pokazivač *p* takav da u bloku između pokazivača *p1* i pokazivača *p* neće biti susjednih elemenata koji su međusobno jednaki. Ukoliko u nizu nije bilo susjednih elemenata koji su međusobno jednaki, funkcija vraća *p2* kao rezultat. Bitno je naglasiti da ova funkcija neće ukloniti nesusjedne elemente koji su međusobno jednaki, već samo susjedne. Također, ova funkcija ne mijenja sam broj elemenata niza niti vektora nego samo premješta elemente da obezbijedi traženo svojstvo. Sadržaj bloka između vraćenog pokazivača i *p2* na kraju je nedefiniran.
`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.

Razmotrimo nekoliko jednostavnih primjera primjene ovih funkcija. Pretpostavimo prvo 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);
```

Podsjetimo se da je, zahvaljujući pokazivačkoj aritmetici, ovaj poziv ekvivalentan pozivu

```
copy(&niz1[0], &niz1[10], &niz2[0]);
```

s obzirom da se imena nizova, upotrijebljena sama za sebe (bez indeksiranja), automatski konvertiraju u pokazivače na prve elemente nizova. Tako se izrazi "`niz1`" i "`niz2`" automatski interpretiraju kao izrazi "`&niz1[0]`" i " `&niz2[0]`", dok se izraz "`niz1 + 10`" interpretira kao "`&niz1[0] + 10`", što je, zahvaljujući pokazivačkoj aritmetici, isto što i "`&niz1[10]`"

Radi boljeg razumijevanja funkcije "`copy`", navedimo da je ona implementirana prinicipijelno ovako, samo često na efikasniji način, uz pozivanje izvjesnih mašinskih potprograma:

```
template <typename PokTip1, typename PokTip2>
  PokTip2 copy(PokTip1 pocetak, PokTip1 kraj, PokTip2 odrediste) {
    while(pocetak != kraj) *odrediste++ = *pocetak++;
    return odrediste;
  }
```

Primijetimo da je u ovoj funkciji korištena *potpuna dedukcija*, odnosno nisu navedeni nikakvi detalji o tome šta parametri "pocetak", "kraj" i "odrediste" predstavljaju, pa čak nije naglašeno ni da su oni pokazivači (nema zvjezdice u deklaraciji). Time je postignuto da ovi parametri *i ne moraju biti pokazivači*. Zaista, oni će moći biti i recimo iteratori, odnosno bilo kakvi objekti na koje je moguće primijeniti dereferenciranje i pokazivačku aritmetiku (možemo reći objekti koji pripadaku *konceptu pokazivačkih tipova*, odakle i ime metatipova "PokTip1" i "PokTip2"). Primijetimo da tip parametara "pocetak" i "kraj" mora biti isti, ali parametar "odrediste" može biti drugačijeg tipa. Recimo, legalno je da "pocetak" i "kraj" budu pokazivači, a "odrediste" iterator.

Uzmimo sada da je elemente niza "niz1" sa indeksima u opsegu od 4 do 9 potrebno pomjeriti za jedno mjesto *naniže*, 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 *naviše*, 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);
```

U slučaju da umjesto nizova koristimo *vektore*, moramo eksplicitno koristiti operator uzimanja adrese "&", s obzirom da se imena vektora upotrijebljena bez indeksiranja ne konvertiraju automatski u adresu prvog elementa vektora. Tako, ukoliko je "v" neki vektor, efekte slične efektima prethodne dvije naredbe *ne bismo mogli* ostvariti konstrukcijama pomoću sljedećih (one ne bi bile sintaksno ispravne):

```
copy(v + 4, v + 10, v + 3);
copy_backward(v + 5, v + 9, v + 6);
```

Umjesto toga, morali bismo eksplicitno pisati konstrukcije poput sljedećih:

```
copy(&v[4], &v[10], &v[3]);
copy_backward(&v[5], &v[9], &v[6]);
```

Neka je sada potrebno ispisati najveći element u nizu "niz1". To možemo uraditi recimo ovako:

```
cout << *max_element(niz1, niz1 + 10);
```

Zvjezdica je potrebna zbog činjenice da funkcija "max_element" ne vraća sam maksimalni element, nego *pokazivač na njega*, tako da je potrebno primijeniti dereferenciranje. Ova konvencija je uvedena zbog povećane fleksibilnosti. Naime, ukoliko sad hoćemo na mjesto najvećeg elementa upisati nulu, možemo prosto pisati:

```
*max_element(niz1, niz1 + 10) = 0;
```

Ranije smo vidjeli da se sličan efekat može izvesti bez upotrebe dereferenciranja, ukoliko se umjesto pokazivača upotrijebe reference. Autori biblioteke "algorithm" ipak su se odlučili za pokazivače a ne za reference, s obzirom da su pokazivači generalniji od referenci.

Razmotrimo sada kako bi se funkcija "max_element" mogla iskoristiti da ispišemo najveći element vektora "v" za koji ne znamo koliko ima elemenata. Jedan od načina je da primijenimo funkciju "size" nad vektorom "v", sa ciljem da utvrdimo broj njegovih elemenata:

```
cout << *max_element(&v[0], &v[v.size()]);
```

Međutim, postoji i bolji način. Funkcije "begin" i "end" bez parametara, primijenjene na neki vektor, vraćaju respektivno pokazivač na prvi element vektora, i pokazivač na mjesto iza posljednjeg elementa vektora, tako da su, za slučaj vektora, konstrukcije "v.begin()" i "v.end()" praktično ekvivalentne sa "&v[0] " i "&v[v.size()]" respektivno. Tako se prethodna naredba može čitljivije napisati ovako:

```
cout << *max_element(v.begin(), v.end());
```

Strogo rečeno, funkcije "begin" i "end" ne vraćaju kao rezultat *pokazivače* nego *iteratore*, ali u slučaju vektora pokazivači i iteratori su u većini konteksta praktično jedno te isto (uz izvjesne iznimke na koje će biti ukazano u nastavku teksta). Međutim, kako su iteratori općenitiji, oni se mogu koristiti i sa složenijim strukturama podataka od vektora, kakvi su na primjer dekovi. Tako, ukoliko je "d" neki dek, konstrukcija poput

```
cout << *max_element(d.begin(), d.end());
```

radiće savršeno ispravno. S druge strane, konstrukcija poput

```
cout << *max_element(&d[0], &d[d.size()]);
```

ne mora davati ispravan rezultat (bez obzira što je sintaksno ispravna), s obzirom da se za elemente dekova ne garantira da slijede jedan za drugim u memoriji. Treba zapamtiti da su pokazivači i pokazivačka aritmetika validni samo kada se primjenjuju na elemente nizova i vektora. Oni se nikada ne smiju koristiti sa dekovima i drugim složenijim strukturama podataka.

Ilustrirajmo još i funkciju "remove". Pretpostavimo da niz "niz1" sadrži redom elemente 3, 7, 2, 5, 7, 4, 7, 1, 7 i 3. Pretpostavimo dalje da je "p" neki pokazivač na tip elemenata niza (recimo "**double**"). Nakon poziva

```
p = remove(niz1, niz1 + 10, 7);
```

elementi niza "niz1" sa indeksima od 0 do 5 biće redom 3, 2, 5, 4, 1 i 3, dok će elementi niza sa indeksima od 6 do 9 biti nedefinirani (u većini implementacija, biće onakvi kakvi su bili i ranije, odnosno 7, 1, 7 i 3, ali se na tu osobinu ne treba oslanjati). Pokazivač "p" će pokazivati na element sa indeksom 6, tako da će, zahvaljujući pokazivačkoj aritmetici, izraz "p – niz" sadržavati broj elemenata u bloku koji više ne sadrži element 7. Stoga, elemente nakon obavljenog "odstranjivanja", možemo ispisati ovako:

```
for(int i = 0; i < p - niz1; i++) cout << niz1[i] << endl;
```

Alternativno smo mogli neposredno koristiti pokazivačku aritmetiku:

```
for(double *p1 = niz1; p1 < p; p1++) cout << *p1 << endl;
```

Treba imati na umu da funkcija "remove" ne uklanja fizički elemente sa navedenom vrijednošću, nego samo premješta elemente niza tako da u navedenom bloku više ne bude elemenata sa navedenom vrijednošću, dok međusobni poredak ostalih elemenata ostaje netaknut (slično vrijedi i za funkciju "unique"). Za slučaj nizova, fizičko uklanjanje elemenata nije ni moguće izvesti (s obzirom da je broj njihovih elemenata fiksan), dok je u slučaju vektora fizičko uklanjanje nakon obavljene reorganizacije moguće izvesti (recimo, pozivom funkcije "resize" nad vektorom).

U vezi sa funkcijama iz biblioteke "algorithm" koje vraćaju kao rezultat pokazivače, kao što su "unique", "remove", "find" i slične, treba paziti na jednu bitnu činjenicu. Naime, ove funkcije će kao rezultat vratiti pokazivač *samo ukoliko su im kao argumenti također proslijeđeni pokazivači*. Ukoliko se ovim funkcijama kao argumenti proslijede *iteratori*, kao rezultat funkcije će *također biti vraćen iterator* (a ne pokazivač). U mnogim kontekstima ovo nije bitno, jer se iteratori u većini slučajeva mogu koristiti na isti način kao i pokazivači. Na primjer, iteratori se mogu dereferencirati na isti način kao i pokazivači.

Međutim, u općem slučaju, *iterator se ne mogu dodjeljivati pokazivačima niti obratno*. Ukoliko o ovome ne vodimo računa, mogu nastupiti problemi. Neka je, na primjer, "v" neki vektor cijelih brojeva. Mada je konstrukcija poput

```
int *p(find(&v[0], &v[v.size()], 5));
```

posve korektna, to ne vrijedi za konstrukciju poput

```
int *p(find(v.begin(), v.end(), 5));
```

Naime, kako su izrazi "`v.begin()`" i "`v.end()`" po tipu *iteratori* (a ne pokazivači), funkcija "`find`" će kao rezultat vratiti također *iterator*, koji se ne može dodijeliti *pokazivaču* "`p`". Da bi ova posljednja konstrukcija bila ispravna, "p" bi trebalo deklarirati kao *iterator na elemente vektora cijelih brojeva*, a ne kao pokazivač na cijeli broj. Čisto radi kompletnosti, navedimo da bi ta deklaracija izgledala ovako (detaljnija objašnjenja izlaze izvan okvira ovog kursa):

```
vector<int>::iterator p(find(v.begin(), v.end(), 5));
```

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 prihvataju parametre izrazito različitih tipova. Stoga je potrebno voditi računa da parametri koje prosljeđujemo ovim funkcijama imaju smisla. U protivnom, moguće su prijave vrlo čudnih grešaka od strane kompajlera, a može se desiti i da ne bude prijavljena nikakva greška, ali da funkcija ne radi u skladu sa očekivanjima.

Biblioteka "`algorithm`" također sadrži mnoštvo funkcija koje kao neke od svojih parametara prihvataju druge 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 ove biblioteke koje prihvataju druge funkcije kao parametre. Pri opisu značenja parametara korištene su iste konvencije kao i ranije:

`for_each(`*p1*`,` *p2*`,` *f*`)`	Poziva uzastopno funkciju *f* prenoseći joj redom kao argument sve elemente između pokazivača *p1* i *p2*.
`transform(`*p1*`,` *p2*`,` *p3*`,` *f*`)`	Poziva uzastopno funkciju *f* prenoseći joj redom kao argument sve elemente između pokazivača *p1* i *p2*. Pri tome se vrijednosti vraćene iz funkcije smještaju redom u blok memorije koji počinje od pokazivača *p3*.
`transform(`*p1*`,` *p2*`,` *p3*`,` *p4*`,` *f*`)`	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*.
`equal(`*p1*`,` *p2*`,` *p3*`,` *f*`)`	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`**".
`count_if(`*p1*`,` *p2*`,` *f*`)`	Vraća kao rezultat broj elemenata između pokazivača *p1* i *p2* za koje funkcija *f* vraća kao rezultat "**`true`**" kad joj se proslijede kao argument.

`find_if(`*p1*`,` *p2*`,` *f*`)`	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.
`replace_if(`*p1*`,` *p2*`,` *f*`,` *v*`)`	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*.
`replace_copy_if(`*p1*`,` *p2*`,` *p3*`,` *f*`,` *v*`)`	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*.
`remove_if(`*p1*`,` *p2*`,` *f*`)`	Reorganizira elemente bloka između pokazivača *p1* i *p2* tako što uklanja sve elemente za koje funkcija *f* vraća kao rezultat "**`true`**" kad joj se proslijede kao argument. Kao rezultat vraća pokazivač takav da u bloka 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. Ova funkcija ne mijenja sam broj elemenata niza niti vektora, nego samo premješta elemente da obezbijedi traženo svojstvo. Sadržaj bloka između vraćenog pokazivača i *p2* na kraju je nedefiniran.
`remove_copy_if(`*p1*`,` *p2*`,` *p3*`,` *f*`)`	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.

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" zbirove 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.

Nije teško prepraviti navedene primjere da umjesto nizova koriste vektore. Jedino je umjesto konstrukcija poput "niz1" i "niz1 + 10" potrebno koristiti konstrukcije poput "&v[0]" i "&v[10]" (odnosno "&v[v.size()]" ukoliko ne znamo koliko vektor ima elemenata) ili, alternativno, konstrukcije poput "v.begin()" i "v.end()", ali u tom slučaju u posljednjem primjeru "pok" treba deklarirati kao iterator, a ne kao pokazivač.

Vrlo je interesantan i sljedeći primjer. Neka je "s" neki niz znakova (tj. niz čiji su elementi tipa "**char**"), i neka je data neka funkcija koja pretvara mala slova u velika, recimo poput sljedeće (u kojoj je iskorištena funkcija "toupper" iz biblioteke "cctype"):

```
char NapraviVeliko(char znak) {
  return toupper(znak);
}
```

U ovoj funkciji nije potrebna eksplicitna konverzija rezultata funkcije "toupper" u tip "**char**", s obzirom da će do te konverzije svakako doći zbog činjenice da je funkcija "NapraviVeliko" deklarirana kao funkcija koja vraća vrijednost tipa "**char**". Sada, ukoliko želimo pretvoriti sva mala slova u nizu znakova "s" u velika, možemo koristiti poziv poput sljedećeg:

```
transform(s, s + strlen(s), s, NapraviVeliko);
```

Prirodno je postaviti pitanje možemo li nekako izbjeći potrebu za pisanjem funkcije "NapraviVeliko" i iskoristiti direktno funkciju "toupper" unutar funkcije "transform". Pokušaj da prosto upotrijebimo ime funkcije "toupper" na mjestu gdje smo koristili ime funkcije "NapraviVeliko", tj. da napišemo

```
transform(s, s + strlen(s), s, toupper);
```

neće raditi, već će dovesti do prijave greške od strane kompajlera. Razlog za ovo nije principijelne prirode (tj. u načelu bi ovo trebalo da radi), nego je problem u tome što je funkcija "toupper" iz biblioteke "cctype" *preklopljena funkcija*, tj. postoji više različitih funkcija sa imenom "toupper" (koje primaju različit broj parametara), tako da se samo navođenjem imena funkcije "toupper" (bez pripadnih parametara) ne zna na koju se tačno funkciju "toupper" navod odnosi. Postoji mogućnost da se ovaj problem riješi, doduše uz upotrebu malo rogobatne sintakse (koja je, u principu, vrsta *type-castinga*, koja ovaj put ne služi za pretvorbu tipova, nego za pomoć u izboru odgovarajuće funkcije). Tako se, u sljedećoj konstrukciji, eksplicitno navodi da mislimo na onu funkciju "toupper" koja prima parametar tipa "**int**", i čiji je rezultat također tipa "**int**". Stoga ona radi ispravno, u skladu sa očekivanjima:

```
transform(s, s + strlen(s), s, (int(*)(int))toupper);
```

Slične manipulacije mogu se upotrijebiti i sa dinamičkim stringovima, odnosno sa promjenljivim tipa "string", samo što moramo uvažiti činjenicu da se imena dinamičkih stringova ne konvertiraju automatski u pokazivače, nego je potrebno eksplicitno uzimati adrese pomoću operatora "&". Na

primjer, neka je "`s`" neki dinamički string. Transformaciju svih malih slova u stringu "`s`" u velika slova možemo izvesti recimo ovako:

```
transform(&s[0], &s[s.length()], &s[0], (int(*)(int))toupper);
```

Naravno, umjesto rogobatne konstrukcije "`(int(*)(int))toupper`", možemo koristiti i vlastitu funkciju "`NapraviVeliko`". Još je bolja sljedeća konstrukcija, s obzirom da tip "`string`" također podržava funkcije "`begin`" i "`end`":

```
transform(s.begin(), s.end(), s.begin(),(int(*)(int))toupper);
```

Kod funkcije "`transform`" također treba paziti da odredište sadrži dovoljan prostor da prihvati transformirani blok. U protivnom, posljedice su nepredvidljive i obično fatalne.

Ako razmotrimo do sada 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, prenos funkcija kao parametara u druge funkcije, i općenitije, pokazivači na funkcije, omogućavaju da neka funkcija koja je ugrađena u sam programski jezik ili operativni sistem ponovo poziva funkciju iz našeg programa (tako što ćemo joj npr. ime te funkcije proslijediti 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 upravljani 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 imena funkcija 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*.

U praktičnom programiranju vrlo se često susreće potreba za *sortiranjem* elemenata nizova, vektora i srodnih struktura podataka. U jeziku C++, postupak sortiranja nije potrebno posebno programirati, s obzirom da biblioteka "`algorithm`" posjeduje funkciju "`sort`", koja izvodi sortiranje niza. Parametri ove funkcije su pokazivači koji omeđuju dio niza koji treba sortirati, slično kao u većini drugih funkcija iz biblioteke "`algorithm`". Pored funkcije "`sort`", ova biblioteka sadrži i neke njoj srodne funkcije, od kojih su najvažnije prikazane u sljedećoj tabeli:

`sort(`*p1*`,` *p2*`)`	Sortira elemente između pokazivača *p1* i *p2* u rastući poredak (tj. u poredak definiran operatorom "<").
`stable_sort(`*p1*`,` *p2*`)`	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.
`partial_sort(`*p1*`,` *p2*`,` *p3*`)`	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.
`nth_element(`*p1*`,` *p2*`,` *p3*`)`	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.

`partial_sort_copy(`*p1*`,` *p2*`,` *p3*`,` *p4*`)` 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 *p*1 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 (podvarijanta median-3) i sortiranja umetanjem (pri čemu se sortiranje umetanjem koristi za slučaj manjih ili skoro sortiranih nizova), tako da će funkcija "`sort`" praktično 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. Da bi se omogućilo njihovo korištenje za sortiranje u opadajućem ili nekom drugom poretku, ove funkcije primaju kao opcionalni posljednji parametar funkciju kriterija koja treba da definira kriterij poređenja. Funkcija kriterija prima dva parametra, koji predstavljaju dva elementa koji se porede (njihov tip treba odgovarati tipu elemenata niza koji se sortira). Ona 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);
```

Slično, za sortiranje vektora "`v`" (nepoznatog broja elemenata) u opadajući poredak, iskoristićemo poziv

```
sort(v.begin(), v.end(), 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 funkciju kriterija.

Zahvaljujući mogućnosti zadavanja funkcije kriterija, moguće je sortirati i nizove odnosno vektore za čije elemente nije definiran standardni poredak (npr. nizove kompleksnih brojeva), ukoliko samostalno definiramo kriterij poretka. Na primjer, niz kompleksnih brojeva možemo sortirati u rastući poredak po apsolutnoj vrijednosti, ukoliko zadamo funkciju kriterija poput sljedeće:

```
bool ManjiPoApsolutnoj(complex<double> a, complex<double> b) {
  return abs(a) < abs(b);
}
```

Tada ćemo, za sortiranje niza "`niz_kompleksnih`" od 100 elemenata u rastući poredak po apsolutnoj vrijednosti sortirati pomoću poziva

```
sort(niz_kompleksnih, niz_kompleksnih + 100, ManjiPoApsolutnoj);
```

Funkcija kriterija je uvijek potrebna kad želimo sortirati nizove ili vektore za čije elemente nije definiran podrazumijevani poredak. Na primjer, koristeći strukturne tipove, moguće je 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 (ili vektor) 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 kasnije.

Postoje situacije kada treba sortirati nizove ili vektore 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 (ili vektor) *pokazivača na znakove*. Na primjer, zamislimo da imamo niz poput sljedećeg:

```
const char *imena[5] = {"Maja", "Tarik", "Ivo", "Belma", "Sead"};
```

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 mjesta u memoriji gdje su ova imena zapisana. Prilikom poziva funkcije "`sort`", ona bi za poređenje elemenata niza koristila relacioni operator "<". Međutim, kako su elementi ovog niza pokazivači, operator "<" 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`" iz biblioteke "`cstring`". Ova funkcija kao parametar prihvata dva niza znakova (obična, ne dinamička stringa), ili pokazivače na početke takvih nizova, i vraća kao rezultat neku vrijednost manju od nule ukoliko je prvi niz ispred drugog po abecednom poretku (preciznije, po poretku ASCII kodova), vrijednost nula ukoliko su nizovi jednaki, a vrijednost veću od nule ukoliko je prvi niz iza drugog po abecednom poretku. Slijedi da je potrebno definirati funkciju kriterija poput sljedeće:

```
bool IspredPo_ASCII_Kodovima(const char *a, const char *b) {
  return strcmp(a, b) < 0;
}
```

Sada bi poziv poput

```
sort(imena, imena + 5, IspredPo_ASCII_Kodovima);
```

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. Izmijeniće se samo *poredak pokazivača koji na njih pokazuju*, o čemu smo ukratko govorili na početku ovog predavanja.

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 (odnosno poretku ASCII kodova). 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.

Funkcija kriterija se ipak može vrlo efikasno iskoristiti i pri sortiranju dinamičkih stringova. Naime, poznato je da se poređenje stringova obavlja na osnovu poređenja odgovarajućih ASCII kodova, što nije uvijek najpoželjnije. Na primjer, string "`bab`" biće smješten ispred stringa "`Aab`", jer sva velika slova

imaju manje ASCII kodove od bilo kojeg velikog slova. Ovo možemo ispraviti ukoliko napravimo funkciju kriterija koja definira ispravno poređenje po abecedi. Jedan od načina je da funkciju kriterija napišemo recimo ovako:

```
bool IspredPoAbecedi(string a, string b) {
  for(int i = 0; i < a.length(); i++) a[i] = toupper(a[i]);
  for(int i = 0; i < b.length(); i++) b[i] = toupper(b[i]);
  return a < b;
}
```

Alternativno bismo mogli koristiti i funkciju "`transform`", na primjer na sljedeći način:

```
bool IspredPoAbecedi(string a, string b) {
  transform(a.begin(), a.end(), a.begin(), (int(*)(int))toupper);
  transform(b.begin(), b.end(), b.begin(), (int(*)(int))toupper);
  return a < b;
}
```

U svakom slučaju, poziv funkcije "`sort`" nad nizom ili vektorom stringova, uz navođenje funkcije "`IspredPoAbecedi`" kao trećeg parametra, izvršio bi istinsko sortiranje po abecednom poretku bez obzira na to da li se u stringovima nalaze mala ili velika slova. Analogne funkcije kriterija mogle bi se napraviti i za sortiranje nizova pokazivača na znakove, što možete uraditi samostalno za vježbu.

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`" iz iste biblioteke (koja je zapravo identična funkciji "`Razmijeni`" koju smo pisali kada smo objašnjavali prenos parametara po referenci). 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-terminirani 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`" (postoji doduše i mogućnost da se takvi nizovi sortiraju funkcijom "`qsort`" iz biblioteke "`cstdlib`", ali vidjećemo uskoro da se ova funkcija u jeziku C++ smije koristiti samo uz izuzetan oprez i veliko znanje). Ovo nije preveliko ograničenje, s obzirom da upotrebu dvodimenzionalnih nizova znakova svakako treba izbjegavati, zbog neracionalnog trošenja memorije.

Funkcije "`find`" i "`find_if`" koje smo spominjali, obavljaju klasičnu linearnu odnosno sekvencijalnu pretragu (pretraga se obavlja redom, element po element). Ukoliko su elementi niza ili vektora prethodno sortirani, postoje mnogo efikasniji načini za pretragu, kao što je npr. *binarna pretraga*, koja se zasniva na postupku uzastopnog polovljenja. Biblioteka "`algorithm`" sadrži i funkcije koje obavljaju binarnu pretragu, tako da su znatno brže od srodnih funkcija poput "`find`" koje obavljaju klasičnu sekvencijalnu 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*.

Na kraju, treba reći da u jeziku C također postoje neke općenite funkcije koje obavljaju zadatke srodne pojedinim funkcijama iz biblioteke "algorithm". Ove funkcije se uglavnom nalaze u biblioteci "cstdlib". Tako se, na primjer, funkcije "memcpy", "memset", "qsort" i "bsearch" u jeziku C koriste umjesto funkcija "copy", "fill", "sort" i "lower_bound", na dosta sličan način, ali uz nešto komplikovanije parametre. Programeri u jeziku C intenzivno koriste ovakve funkcije. Mada se one, u načelu, mogu koristiti i u jeziku C++, postoje veoma 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, ove funkcije svoj rad baziraju na pretpostavci da je manipulacije sa svim tipovima podataka moguće izvoditi *bajt po bajt*. Ta pretpostavka 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 *nisu*.

Opisani problemi sa korištenjem pojedinih funkcija naslijeđenih iz jezika C 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", dok umjesto funkcije "qsort" treba koristiti funkciju "sort". Ove funkcije, ne samo da su univerzalnije od srodnih funkcija naslijeđenih iz jezika C, već su i jednostavnije za upotrebu. Činjenica je da se programeri koji posjeduju veće iskustvo u jeziku C teško odriču upotrebe funkcija poput "memcpy" i "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" tipično nisu *ništa manje efikasne*. Zapravo, u mnogim implementacijama biblioteke "algorithm", poziv funkcija poput "copy" automatski se 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 prihvatiti činjenicu: funkcije poput "memcpy", "memset", "qsort" i njima slične *zaista više ne treba koristiti* u C++ programima, osim u *vrlo iznimnim slučajevima* (recimo, funkcija "qsort" je u stanju da sortira dvodimenzionalni niz znakova, dok funkcija "sort" to nije u stanju, kao što je već naglašeno). One su zadržane pretežno sa ciljem da omoguće kompatibilnost sa već napisanim programima koji su bili na njima zasnovani!

Predavanje 7.

Prije nego što se upoznamo sa pojmom *klasa*, koje čine okosnicu objektno-zasnovanog i objektno-orijentiranog stila programiranja u jeziku C++, prvo je potrebno da detaljno obnovimo i proširimo znanje o složenim tipovima podataka naslijeđenim iz jezika C, koji se zovu *strukture*, *slogovi* ili *zapisi* (engl. *records*), s obzirom da se klase definiraju kao prirodno proširenje ovakvih tipova podataka. S obzirom da 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*). Podsjetimo se da strukture predstavljaju složene tipove podataka čuvaju skupinu *raznorodnih podataka*, koji se mogu razlikovati kako po prirodi tako i po tipu, i kojima se, za razliku od nizova, ne pristupa po *indeksu* već po *imenu*. Pri deklaraciji strukture se iza ključne riječi "**struct**" prvo navodi *ime strukture*, a zatim se u vitičastim zagradama navode *deklaracije individualnih elemenata* od kojih se struktura sastoji. Na primjer, sljedeća deklaracija uvodi strukturni 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 na čiju ćemo ulogu uskoro ukazati. Individualni elementi strukture kao što su "ime", "odjeljenje", "platni_broj" i "plata" u jeziku C obično su se nazivali se *polja* (engl. *fields*) strukture, dok se u jeziku C++, pod utjecajem objektno-orijentirane terminologije, obično govori o *atributima* ili *podacima članovima* (engl. *data members*) *strukture*.

Ključna riječ "**struct**", isto kao i ključna riječ "**enum**", samo definira *novi tip* (koji ćemo zvati *strukturni tip*), a ne i konkretne primjerke objekata tog tipa. Tako, na primjer, tip "Radnik" samo opisuje koja karakteristična svojstva (atribute) posjeduje bilo koji radnik. Tek nakon što deklariramo odgovarajuće promjenljive strukturnog 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. U jeziku C se prilikom deklaracije strukturnih promjenljivih mora ponoviti ključna riječ "**struct**", kao u sljedećoj deklaraciji:

```
struct Radnik sekretar, sluzbenik, monter, portir;
```

Mada je, zbog kompatibilnosti, ovakva sintaksa dozvoljena i u jeziku C++, ona se smatra izrazito nepreporučljivom, jer je u konfliktu sa nekim novouvedenim konceptima jezika C++ vezanim za strukturne tipove podataka.

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;
```

Uloga tačka-zareza iza zatvorene vitičaste zagrade ovim postaje jasna. Naime, njime govorimo da u tom trenutku ne želimo da definiramo ni jedan konkretan primjerak upravo deklariranog strukturnog tipa. Ipak, danas se smatra da je deklariranje konkretnih primjeraka strukture istovremeno sa deklariranjem strukture loš stil pisanja, koji treba izbjegavati.

Kao što znamo, pojedinačna polja (atributi, podaci članovi) unutar slogovnih promjenljivih ponašaju se poput *individualnih promjenljivih* odgovarajućeg tipa, kojima se pristupa tako što prvo navedemo *ime slogovne promjenljive*, zatim znak (zapravo operator) *tačka* ("`.`") i, konačno, *ime polja unutar slogovne 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;
```

Za postavljanje polja "`ime`" i "`odjeljenje`" ne možemo koristiti operator dodjele "=" nego funkciju "`strcpy`", s obzirom da se radi o poljima nizovnog tipa (naravno, operator "=" mogli bismo koristiti da smo ova polja deklarirali kao dinamičke stringove, odnosno kao polja tipa "`string`"). Treba naglasiti da nije moguće *jednim iskazom* postaviti sva polja u nekoj strukturnoj promjenljivoj, 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). Inicijalizacija se vrši tako što se u vitičastim zagradama navode inicijalizacije za *svako od polja posebno*, onim redom kako su definirana u deklaraciji sloga. Na primjer:

```
Radnik sekretar = {"Ahmed Hodžić", "Marketing", 34, 530};
```

Inače, tipovi podataka koji se sastoje od *fiksnog broja* individualnih komponenti, i koje se mogu inicijalizirati navođenjem vrijednosti pojedinih komponenti u vitičastim zagradama, 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* (u stvari, nije baš jedina – 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;
```

Kako polja strukture ne postoje kao neovisni objekti, nego samo kao dijelovi neke konkretne promjenljive strukturnog tipa, polja koja čine strukturu ne mogu se inicijalizirati unutar deklaracije same strukture, s obzirom 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 specifikacijama "**`const`**" i "**`static`**" (preciznije, sa obje ove specifikacije istovremeno). O smislu takvih specifikacija 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. Također, za razliku od nizova, strukture se mogu i *vraćati kao rezultati iz funkcija*. Ova tehnika ilustrirana je u sljedećem primjeru, u kojem je definiran strukturni tip podataka "`Vektor3d`" koji opisuje vektor u prostoru koji se, kao što znamo, može opisati sa tri koordinate x, y i z. U prikazanom programu, prvo se čitaju koordinate dva vektora sa tastature, a zatim se računa i ispisuje njihova suma i njihov vektorski prozivod (pri čemu se vektor ispisuje u formi tri koordinate međusobno razdvojene zarezima, unutar vitičastih zagrada):

```
#include <iostream>
using namespace std;
struct Vektor3d {
  double x, y, z;
};
void UnesiVektor(Vektor3d &v) {
  cout << "X = "; cin >> v.x;
  cout << "Y = "; cin >> v.y;
  cout << "Z = "; cin >> v.z;
}
Vektor3d ZbirVektora(const Vektor3d &v1, const Vektor3d &v2) {
  Vektor3d v3;
  v3.x = v1.x + v2.x; v3.y = v1.y + v2.y; v3.z = v1.z + v2.z;
  return v3;
}
Vektor3d VektorskiProizvod(const Vektor3d &v1, const Vektor3d &v2) {
  Vektor3d 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 Vektor3d &v) {
  cout << "{" << v.x << "," << v.y << "," << v.z << "}";
}
int main() {
  Vektor3d 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 prozivod je: ";
  IspisiVektor(VektorskiProizvod(a, b));
  return 0;
}
```

Listing ovog programa može se skinuti pod imenom "`vektor3d.cpp`" sa web stranice kursa. Primijetimo 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:

```
Vektor3d ZbirVektora(const Vektor3d &v1, const Vektor3d &v2) {
  Vektor3d v3 = {v1.x + v2.x, v1.y + v2.y, v1.z + v2.z};
  return v3;
}
Vektor3d VektorskiProizvod(const Vektor3d &v1, const Vektor3d &v2) {
  Vektor3d 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 parametara. 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 "`Vektor3d`" (npr. rezultat neke druge funkcije koja vraća objekat tipa "`Vektor3d`"), 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, alternativna mogućnost je da formalni parametri ovih funkcija uopće ne budu reference, odnosno da se koristi prenos parametara *po vrijednosti*. Međutim, korištenjem prenosa po referenci sprečava se nepotrebno 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 "`Vektor3d`" 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 strukturnog tipa *po vrijednosti* ne treba koristiti ukoliko to nije zaista neophodno.

U praksi se često javlja potreba za obradom *skupine slogova*, recimo podacima o *svim studentima* na jednoj godini studija. Za modeliranje takvih podataka iz stvarnog života možemo koristiti *nizove struktura* (ili *vektore struktura*) odnosno nizove (ili vektore) *čiji je svaki element strukturnog tipa*. Na primjer, sljedeće deklaracije omogućavaju nam da predstavimo skupinu radnika:

```
struct Radnik {
  char ime[40];
  char odjeljenje[20];
  int platni_broj;
  double plata;
};
Radnik radnici[100];
```

Fleksibilnost dodatno možemo povećati upotrijebimo li *vektor struktura* umjesto niza struktura, tj. upotrijebimo li deklaraciju poput

```
vector<Radnik> radnici(100);
```

S obzirom da *svaki element ovakvog niza odnosno vektora predstavlja strukturu*, polju "ime" trećeg elementa promjenljive "radnici" (preciznije, elementa sa indeksom 3, s obzirom da numeracija indeksa počinje od nule) možemo pristupiti pomoću konstrukcije "radnici[3].ime". Tako, na primjer, ukoliko pretpostavimo da koristimo vektor čiji su elementi tipa "Radnik", možemo iskoristiti konstrukciju poput

```
for(int i = 0; i < radnici.size(); 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, vektorskog, pa čak i ponovo strukturnog tipa (pri tome, pri korištenju vektora kao elemenata struktura nastaju neke specifičnosti, o kojima ćemo govoriti malo kasnije). Tako, kombinirajući nizove, vektore 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 strukturni tip "Ucenik", koji opisuje jednog učenika. Atributi strukture "Ucenik" mogu biti "ime", "prezime", "datum_rodjenja", "ocjene", "prosjek" i "prolaz". Atribute "ime" i "prezime" definiraćemo kao stringovni tip, atribut "prosjek" će biti realnog tipa, dok će atribut "prolaz" biti logička promjenljiva. Atribut "ocjene" ćemo definirati kao obični niz cijelih brojeva, dok ćemo atribut "datum_rodjenja" definirati da bude tipa "Datum", pri čemu je "Datum" ponovo strukturni tip koji se sastoji od tri atributa: "dan", "mjesec" i "godina". Na kraju, definiraćemo promjenljivu "ucenici", koja će biti vektor čiji su elementi tipa "Ucenik" (dimenziju vektora ovom prilikom nećemo specificirati). Na osnovu provedene analize možemo napisati sljedeće deklaracije:

```
const int BrojPredmeta(12);

struct Datum {
  int dan, mjesec, godina;
};

struct Ucenik {
  string ime, prezime;
  Datum datum_rodjenja;
  int ocjene[BrojPredmeta];
  double prosjek;
  bool prolaz;
};

vector<Ucenik> ucenici;
```

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). Razumije se da ćemo pojedinim dijelovima ovakve složene strukture pristupati kombinacijom indeksiranja i navođenja imena polja. 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 prosjek i utvrđuje prolaznost za svakog učenika, i na kraju, prikazuje na ekranu izvještaj o svim učenicima u razredu, sortiran u opadajući redoslijed po prosjeku (listing ovog programa dostupan je pod imenom "ucenici.cpp" na web stranici kursa). Ovisno od toga koji su učenici prošli a koji ne, izvještaj se sastoji od rečenica poput

Učenik Marko Marković, rođen 17.3.1989. ima prosjek 3.89
Učenik Janko Janković, rođen 22.5.1989. mora ponavljati razred

Program je pisan struktuirano, 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. Pri tome, treba napomenuti da u programu nije vršena nikakva kontrola ispravnosti unesenih podataka (poput provjere da li su unesene ocjene u ispravnom opsegu, da li je datum rođenja ispravno zadan itd.), čisto da program ne bi ispao isuviše dugačak.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

const int BrojPredmeta(12);

struct Datum {
  int dan, mjesec, godina;
};

struct Ucenik {
  string ime, prezime;
  Datum datum_rodjenja;
  int ocjene[BrojPredmeta];
  double prosjek;
  bool prolaz;
};

int main() {
  void UnesiUcenike(vector<Ucenik> &ucenici);
  void ObradiUcenike(vector<Ucenik> &ucenici);
  void IspisiIzvjestaj(const vector<Ucenik> &ucenici);
  int broj_ucenika;
  cout << "Koliko ima učenika: ";
  cin >> broj_ucenika;
  try {
    vector<Ucenik> ucenici(broj_ucenika);
    UnesiUcenike(ucenici);
    ObradiUcenike(ucenici);
    IspisiIzvjestaj(ucenici);
  }
  catch(...) {
    cout << "Problemi sa memorijom...\n";
  }
  return 0;
}

void UnesiUcenike(vector<Ucenik> &ucenici) {
  void UnesiJednogUcenika(Ucenik &ucenik);
  for(int i = 0; i < ucenici.size(); 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;
```

```
    cout << "  Datum rođenja (D/M/G): ";
    UnesiDatum(ucenik.datum_rodjenja);
    UnesiOcjene(ucenik.ocjene, BrojPredmeta);
  }

  void UnesiDatum(Datum &datum) {
    char znak;
    cin >> datum.dan >> znak >> datum.mjesec >> znak >> 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(vector<Ucenik> &ucenici) {
    void ObradiJednogUcenika(Ucenik &ucenik);
    bool DaLiJeBoljiProsjek(const Ucenik &u1, const Ucenik &u2);
    for(int i = 0; i < ucenici.size(); i++)
      ObradiJednogUcenika(ucenici[i]);
    sort(ucenici.begin(), ucenici.end(), 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(const vector<Ucenik> &ucenici) {
    void IspisiJednogUcenika(const Ucenik &ucenik);
    cout << endl;
    for(int i = 0; i < ucenici.size(); 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(3) << 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 operatora "<" (mada se, pomoću tehnike preklapanja operatora, može dati smisao i ovom operatoru primijenjenom na strukture).

U navedenom primjeru, sortiranje se 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 zadržaće 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;
```

Zagrade u konstrukcijama poput "(*pok_sekretar).plata" su *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 strukturnu promjenljivu "x" koja ima polje "y" pokazivačkog tipa, koje želimo da dereferenciramo. S druge strane, u izrazu "(*x).y", imamo pokazivač "x" na neki strukturni tip, koji želimo da dereferenciramo, i da nakon toga pristupimo atributu "y" tako dereferenciranog objekta. Alternativno, kao sinonim za često korištenu jezičku konstrukciju "(*x).y", možemo koristiti konstrukciju "x->y" Stoga, umjesto izraza "(*pok_sekretar).plata" možemo pisati ekvivelentni izraz "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 kao neovisan operator koji se naziva *operator indirektnog pristupa*, za razliku od *operatora direktnog pristupa* ".". Na ovaj način, dinamičkim strukturnim promjenljivim možemo pristupati preko pokazivača koji na njih pokazuju na potpuno isti način kao da se radi o običnim strukturnim promjenljivim, samo što umjesto operatora direktnog pristupa "." trebamo koristiti operator indirektnog pristupa "->".

Ranije smo govorili da nema previše smisla kreirati individualne dinamičke promjenljive jednostavnih tipova, kao što je recimo tip "**int**". Međutim, kod strukturnih promjenljivih ne mora biti tako. One mogu biti prilično masivne, tako da manipulacije sa njima mogu biti zahtjevne. Na primjer, jedan konkretan primjerak promjenljive tipa "Ucenik" može zauzeti priličan prostor u memoriji. Sada, ukoliko na primjer treba kopirati promjenljivu "u1" tipa "Ucenik" u promjenljivu "u2" istog tipa,

potrebno je kopirati *čitav sadržaj memorije* koji ona zauzima. 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 strukturnim promjenljivim mogu učiniti mnogo efikasnije ukoliko umjesto sa samim strukturnim promjenljivim manipuliramo sa *pokazivačima koje na njih pokazuju*. Izmjene koje za ovu svrhu treba učiniti najčešće 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 vektora elemenata tipa "`Ucenik`" upotrijebimo *vektor pokazivača na dinamički kreirane objekte tipa* "`Ucenik`". Ova ideja iskorištena je u sljedećem programu:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
const int BrojPredmeta(12);
struct Datum {
  int dan, mjesec, godina;
};
struct Ucenik {
  string ime, prezime;
  Datum datum_rodjenja;
  int ocjene[BrojPredmeta];
  double prosjek;
  bool prolaz;
};
int main() {
  void UnesiUcenike(vector<Ucenik*> &ucenici);
  void ObradiUcenike(vector<Ucenik*> &ucenici);
  void IspisiIzvjestaj(const vector<Ucenik*> &ucenici);
  void OslobodiMemoriju(vector<Ucenik*> &ucenici);
  int broj_ucenika;
  cout << "Koliko ima učenika: ";
  cin >> broj_ucenika;
  try {
    vector<Ucenik*> ucenici(broj_ucenika);
    try {
      UnesiUcenike(ucenici);
    }
    catch(...) {
      OslobodiMemoriju(ucenici);
      throw;
    }
    ObradiUcenike(ucenici);
    IspisiIzvjestaj(ucenici);
    OslobodiMemoriju(ucenici);
  }
  catch(...) {
    cout << "Problemi sa memorijom...\n";
  }
  return 0;
}
void UnesiUcenike(vector<Ucenik*> &ucenici) {
  void UnesiJednogUcenika(Ucenik *ucenik);
```

```
  for(int i = 0; i < ucenici.size(); i++) {
    cout << "Unesite podatke za " << i + 1 << ". učenika:\n";
    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;
  cout << "  Datum rođenja (D/M/G): ";
  UnesiDatum(ucenik->datum_rodjenja);
  UnesiOcjene(ucenik->ocjene, BrojPredmeta);
}
void UnesiDatum(Datum &datum) {
  char znak;
  cin >> datum.dan >> znak >> datum.mjesec >> znak >> 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(vector<Ucenik*> &ucenici) {
  void ObradiJednogUcenika(Ucenik *ucenik);
  bool DaLiJeBoljiProsjek(const Ucenik *u1, const Ucenik *u2);
  for(int i = 0; i < ucenici.size(); i++)
    ObradiJednogUcenika(ucenici[i]);
  sort(ucenici.begin(), ucenici.end(), 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(const vector<Ucenik*> &ucenici) {
  void IspisiJednogUcenika(const Ucenik *ucenik);
  cout << endl;
  for(int i = 0; i < ucenici.size(); 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(3) << ucenik->prosjek;
  else
```

```
    cout << " mora ponavljati razred";
  cout << endl;
}
void IspisiDatum(const Datum &datum) {
  cout << datum.dan << "." << datum.mjesec << "." << datum.godina;
}
void OslobodiMemoriju(vector<Ucenik*> &ucenici) {
  for(int i = 0; i < ucenici.size(); i++) delete ucenici[i];
}
```

Listing ovog programa dostupan je pod imenom "`ucenici_pok.cpp`" na web stranici kursa. 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 proslijedi funkciji "`UnesiJednogUcenika`". Napomenimo da smo ovu funkciju mogli napisati tako da joj se ne prosljeđuje *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 (odnosno referencu na njega, sa ciljem da izbjegnemo nepotrebno kopiranje). 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 usljed memorijskih problema pri alokaciji individualnih učenika. Veoma je bitno da temeljito analizirate i međusobno uporedite prethodna dva programa, s obzirom da je razumijevanje izloženih koncepata od ključne važnosti za razumijevanje materije koja slijedi dalje (listinzi oba programa dostupni su na web stranici kursa).

U prethodna dva primjera, struktura "`Ucenik`" za memoriranje ocjena koristila je *obični niz* "`ocjene`". Fleksibilnost možemo povećati ukoliko za tu svrhu također koristimo vektor. Međutim, prilikom deklariranja vektora kao atributa strukture, nije dozvoljeno deklarirati *dimenziju vektora*, s obzirom da je ona dinamičko svojstvo vektora, i razni primjerci iste strukture mogu imati vektore različitih dimenzija. Stoga, deklaracija poput sljedeće *nije legalna*:

```
struct Ucenik {
  string ime, prezime;
  Datum datum_rodjenja;
  vector<int> ocjene(BrojPredmeta);
  double prosjek;
  bool prolaz;
};
```

Umjesto toga, dimenziju vektora "`ocjene`" treba unutar deklaracije strukture "`Ucenik`" ostaviti nedefiniranu. Dimenziju ovog vektora ćemo definirati *naknadno* (recimo, pozivom funkcije "`resize`") za svaki konkretan primjerak promjenljive tipa "`Ucenik`". Na primjer, da bismo postavili dimenziju vektora "`ocjene`" na iznos "`BrojPredmeta`" u *svim elementima vektora* "`ucenici`" čiji su elementi tipa "`Ucenik`", možemo koristiti konstrukciju poput sljedeće:

```
for(int i = 0; i < ucenici.size(); i++)
  ucenici[i].ocjene.resize(BrojPredmeta);
```

Kasnije, kada se upoznamo sa načinom deklaracije tzv. *konstruktora*, vidjećemo kako se ovaj postupak može automatizirati.

U prethodnim primjerima, koristili smo *vektore* "`ucenici`" čiji su elementi bili strukture tipa "`Ucenik`", odnosno pokazivači na strukture tipa "`Ucenik`". Naravno, umjesto vektora bismo mogli koristiti nizove, ali bi u tom slučaju maksimalni broj učenika *morao biti unaprijed poznat*, s obzirom da

nije moguće deklarirati nizove čiji broj elemenata nije poznat apriori. Kao alternativu korištenju vektora, samo na nižem nivou, možemo koristiti i *dinamičke nizove*. 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];
```

Pored toga, u oba slučaja, u funkcije "`UnesiUcenike`", "`ObradiUcenike`", "`IspisiIzvjestaj`" i "`OslobodiMemoriju`" bi umjesto vektora trebalo prenositi pokazivač na početak dinamičkog niza i broj elemenata niza, s obzirom da se taj podatak ne može dobiti iz samog pokazivača. Uz ove izmjene, sve ostalo bi u ova dva programa, zahvaljujući činjenici da se na pokazivače može primjenjivati indeksiranje, moglo ostati uglavnom isto, osim što bi na kraj programa trebalo dodati i naredbu

```
delete[] ucenici;
```

Razumjevanje izložene materije provjerite tako što ćete samostalno izvršiti predložene izmjene, tj. prepraviti prikazane programe tako da umjesto vektora koriste dinamičke nizove. Vodite računa da se, za razliku od vektora, elementi niza ne inicijaliziraju automatski. Stoga ćete, u primjeru koji koristi dinamički niz pokazivača, morati ručno na početku inicijalizirati njegove elemente na nulu ("**`for`**" petljom ili pomoću funkcije "`fill`") da biste izbjegli eventualne probleme sa operatorom "**`delete`**" u funkciji "`OslobodiMemoriju`" u slučaju neuspješne alokacije nekog od učenika.

Moguće je formirati i *generičke strukture*, odnosno strukture kod kojih tipovi određenih atributa nisu unaprijed poznati. Ovakve strukture se također deklariraju uz pomoć *šablona*, odnosno ključne riječi "**`template`**". Slijedi jedan vrlo jednostavan primjer deklaracije generičke strukture, kao i konkretnih primjeraka objekata ovakvih struktura:

```
template <typename TipElemenata>
struct GenerickiNiz {
  TipElemenata elementi[10];
  int broj_elemenata;
};

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 šablona, 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 šablona*, tako da konstrukcije poput "`GenerickiNiz<double>`" ili "`GenerickiNiz<int>`" *jesu tipovi*. Pri tome, konkretni tipovi dobijeni iz istog šablona uz različite parametre šablona 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.

Prikazana generička struktura "GenerickiNiz" vrlo je jednostavna, i navedena je više kao ilustracija koncepta generičkih struktura. Međutim, ona ujedno ilustrira i jednu interesantnu činjenicu. Poznato je da kada se u funkciju prenose nizovi kao parametri, tada se gotovo uvijek kao dodatni parametar prenosi i broj elemenata niza, koji funkcija drugačije ne bi mogla saznati. Sve ovo se može izbjeći ukoliko formiramo strukturu čiji će atributi biti sam niz, kao i stvarni broj njegovih elemenata (što je upravo urađeno u gore prikazanoj strukturi). Na ovaj način, sve informacije koje opisuju niz (njegovi elementi i broj elemenata) upakovane su u jednu strukturu, koju možemo prenijeti kao jedinstven parametar u funkciju, pa čak i vratiti kao rezultat iz funkcije.

Veoma interesantne mogućnosti dobijamo kreiranjem struktura koje kao svoje atribute koriste *pokazivače*. Na primjer, moguće je kreirati strukturu "Matrica", koja kao svoje atribute sadrži dvojni pokazivač na dinamički alocirani dvodimenzionalni niz, kao i dimenzije matrice (broj redova i kolona). Naravno, za rad sa matricama mnogo je jednostavnije i sigurnije koristiti vektor čiji su elementi vektori, ali razmatranje ovakvih struktura će nam postupno omogućiti razumijevanje internih mehanizama koji stoje u pozadini rada dinamičkih struktura podataka. Stoga ćemo ilustrirati ovaj koncept kroz jedan dosta složen program, koji deklarira generičku strukturu "Matrica", sa neodređenim tipom elemenata matrice, koja sadrži (dvojni) pokazivač na dinamički alociranu matricu, kao i dimenzije matrice. S obzirom da se radi o generičkoj strukturi, sve funkcije koje s njom rade 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. Najsloženija funkcija u ovom programu je funkcija "StvoriMatricu" koja kreira matricu dinamički, i vraća kao rezultat odgovarajuću strukturu "Matrica". Pored toga, ona vodi brigu o tome da ne dođe do curenja memorije ukoliko u procesu kreiranja matrice ponestane memorije, o čemu smo ranije detaljno govorili. Njoj komplementarna funkcija je funkcija "UnistiMatricu", 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 prenositi 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 TipElemenata>
struct Matrica {
  int broj_redova, broj_kolona;
  TipElemenata **elementi;
};

template <typename TipElemenata>
void UnistiMatricu(Matrica<TipElemenata> mat) {
  if(mat.elementi == 0) return;
  for(int i = 0; i < mat.broj_redova; i++) delete[] mat.elementi[i];
  delete[] mat.elementi;
}

template <typename TipElemenata>
Matrica<TipElemenata> StvoriMatricu(int broj_redova, int broj_kolona) {
  Matrica<TipElemenata> mat;
  mat.broj_redova = broj_redova; mat.broj_kolona = broj_kolona;
  mat.elementi = new TipElemenata*[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 TipElemenata[broj_kolona];
  }
```

```
    catch(...) {
      UnistiMatricu(mat);
      throw;
    }
    return mat;
  }

  template <typename TipElemenata>
  void UnesiMatricu(char ime_matrice, Matrica<TipElemenata> &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 TipElemenata>
  void IspisiMatricu(const Matrica<TipElemenata> &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 TipElemenata>
  Matrica<TipElemenata> ZbirMatrica(const Matrica<TipElemenata> &m1,
    const Matrica<TipElemenata> &m2) {
      if(m1.broj_redova != m2.broj_redova
        || m1.broj_kolona != m2.broj_kolona)
          throw "Matrice nemaju jednake dimenzije!\n";
    Matrica<TipElemenata> m3(StvoriMatricu<TipElemenata>(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;
  }
```

Mada je prikazani program vrlo elegantan (listing ovog programa je dostupan pod imenom "`matrica_struct.cpp`" na web stranici kursa), u njemu se javljaju neki prilično nezgodni detalji na

koje treba posebno obratiti pažnju. Prvo, primijetimo da smo pored matrica "a" i "b" definirali i matricu "c", a umjesto jednostavnog poziva poput

```
IspisiMatricu(ZbirMatrica(a, b), 7);
```

koristili smo znatno nezgrapniji poziv

```
IspisiMatricu(c = ZbirMatrica(a, b), 7);
```

koji je funkcionalno ekvivalentan 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 eksplicitno obrisati upotrebom operatora "**delete**". Ova potreba da se eksplicitno 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 pokazivačko polje "elementi" bude inicijalizirano na nulu). Ovo je urađeno zbog sljedećeg razloga. Na kraju programa potrebno je eksplicitno *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! Ukoliko sada pažljivije pogledamo funkciju "UnistiMatricu", vidjećemo da ona *ne radi ništa* u slučaju da polje "elementi" u matrici sadrži nulu (tj. *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" proslijedimo 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 (najvjerovatnije ni na šta smisleno). Stoga je ishod ovakvih akcija posve nepredvidljiv (i, najvjerovatnije, ne vodi ničemu dobrom). Stoga, da nismo ručno inicijalizirali polja "elementi" na nulu, morali bismo koristiti ugniježdene "**try**" – "**catch**" strukture, što je, kao što smo već vidjeli, veoma neelegantno 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 eksplicitno vodi računa o propisnoj inicijalizaciji.

Ovom prilikom je neophodno ukazati na još jednu nezgodnu pojavu koja može nastati kod upotrebe *struktura koje kao svoja polja sadrže pokazivače*. Pretpostavimo 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 (umjesto kao pokazivač na dinamički alocirani 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 ovih 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*).

Postoji ipak jedna potencijalna opasnost do koje može doći usljed 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. oslobađanje 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 oslobađanja zauzetog prostora pozivom "`UnistiMatricu(b)`", pokazivač "`elementi`" u matrici "`a`" će pokazivati na upravo oslobođeni prostor, odnosno postaće viseći 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* strukturnih promjenljivih kao parametara u funkcije, kao i prilikom *vraćanja struktura* kao rezultata iz funkcije. Na primjer, posmatrajmo sljedeću funkciju:

```
template <typename TipElemenata>
void AnulirajMatricu(Matrica<TipElemenata> 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*, a ne *po referenci*. 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 proslijeđene 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*). Ovakvo ponašanje uzrokovano 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 strukturnog tipa koji sadrže pokazivače, također ne dovodi do većih problema, sve dok smo ga svjesni. Međutim, ovakvo ponašanje je moguće promijeniti uz pomoć tzv. *konstruktora kopije*, koji preciziraju način kako će se tačno vršiti kopiranje strukturnih 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 kasnije.

Vjerovatno ćete se sada sa pravom zapitati zbog čega stalno navodimo primjere raznih problematičnih 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: *jako 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 komplicirano (kao što su konstruktori, destruktori, konstruktori kopije, preklapanje operatora dodjele, 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 *veza;
};
```

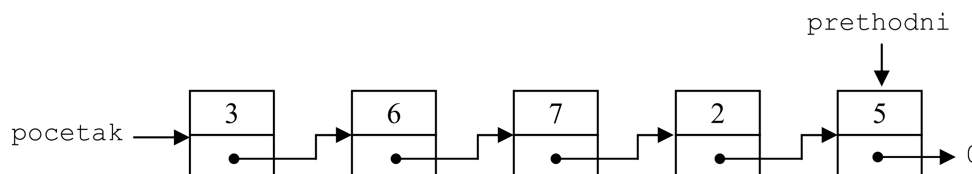
Čvorovi predstavljaju osnovni gradivni element za formiranje tzv. *dinamičkih struktura podataka*, o kojima ćemo govoriti na samom kraju ovog kursa. Na ovom mjestu ćemo ilustrirati samo osnovnu ideju koja ilustrira smisao čvorova. Pretpostavimo da želimo unijeti i zapamtiti slijed brojeva koji se završava nulom, ali da nam broj brojeva koji će biti uneseni nije unaprijed poznat. Pretpostavimo 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 prihvatiti 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 zadanom 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->veza = 0;
  if(pocetak != 0) prethodni->veza = 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 "`veza`" 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:

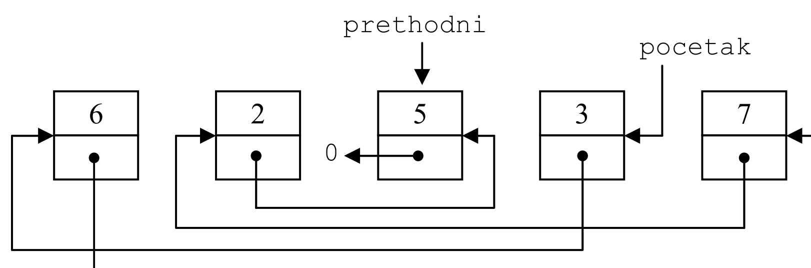
```
cout << pocetak->element;
cout << pocetak->veza->element;
cout << pocetak->veza->veza->element;
```

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 pomoćni 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 pomoću pokazivača. Naime, mi nikada ne znamo *gdje* će tačno u memoriji biti smješten čvor konstruisan operatorom "**new**". Mi ćemo dobiti kao rezultat ovog operatora *adresu* gdje je čvor kreiran, ali ne postoje nikakve garancije da će se čvorovi u memoriji stvarati tako da uvijek čine rastući slijed adresa (mada je takav ishod najvjerovatniji). Tako je principijelno sasvim moguće (ali ne i mnogo vjerovatno) da stvarna memorijska 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 na kraju ovog kursa. 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 *sekvencijalno*, 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 sekvencijalno, 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 ubačenih 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 na kraju ovog kursa.

Treba napomenuti da jednostruko povezane liste nisu jedine strutkure 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 ograničenog prostora, na ovom kursu ne možemo ulaziti u razmatranje ovih struktura podataka, stoga se zainteresirani upućuju na brojnu širu literaturu koja obrađuje ovu problematiku (obično literatura koja obrađuje strukture podataka i algoritme).

Predavanje 8.

Vidjeli smo da strukture omogućavaju jednostavno modeliranje praktično svih tipova podataka koji se susreću u realnom životu. Međutim, one 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 strukturom *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. S druge strane, 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 lišen svakog smisla. Međutim, kako se "d.dan", "d.mjesec" i "d.godina" ponašaju kao obične cjelobrojne promjenljive, ne postoji nikakav mehanizam koji bi nas spriječio da ovakvu 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 tijesno 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 strukturom 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[12] = {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 i tako u nju unijeti nekonzistentan sadržaj. Funkcija "`PostaviDatum`" je potpuno odvojena od strukture "`Datum`", ne predstavlja njen dio, i ničim se ne nameće njeno korištenje. Neko bi se mogao 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 iznostiti 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 premalo samim podacima. Podaci se posmatraju kao sirova hrpa činjenica, čija je jedina svrha postojanja da budu proslijeđeni 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!

Izloženo filozofsko razmatranje dovelo je 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 smatrate 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 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*), koje 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 funkcija 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);
};
```

Promjenljive čiji je tip neka klasa obično nazivamo *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 uvijek primjenjuju *nad nekim konkretnim objektom*. Najčešći način na koji se ta primjena vrši je pomoću operatora "." (tačka), dakle pomoću istog operatora 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):

```
Datum d1, d2;
d1.Postavi(14, 5, 2004);
d2.Postavi(35, 14, 2004);
```

Izvršena promjena sintakse u odnosu na raniji primjer u kojem smo koristili (običnu) funkciju "PostaviDatum" 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, 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) naziva se *operator razlučivosti, vidokruga* ili *dosega* (engl. *scope operator*). Ovaj 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 uvijek 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 izvršimo poziv

```
d1.Postavi(14, 5, 2004);
```

atributi "dan", "mjesec" i "godina" unutar funkcije članice "Postavi" odnosiće se na odgovarajuće atribute 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. Tajna je u tome što se prilikom poziva ma koje funkcije članice nad nekim objektom, njegova adresa 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, 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 prototipom:

```
void Postavi_obicna(Datum *this, int d, int m, int g);
```

Pri tome 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 cjelina posve odvojenih od objekata na koje se one primjenjuju.

U funkciji članici "Postavi", parametre smo nazvali prosto "d", "m" i "g", 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". U jeziku C++ je usvojena konvencija da u slučaju ovakvih konflikata 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`", eksplicitno naglašavamo da mislimo na atribut "`dan`" koji pripada klasi "`Datum`". Stoga smo metodu "`Postavi`" mogli napisati i na sljedeći način, bez korištenja pokazivača "**`this`**":

```
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";
  Datum::dan = dan; Datum::mjesec = mjesec; Datum::godina = godina;
}
```

Uočimo bitnu 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 funkcije članice*, jer će prilikom njenog poziva 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 eksplicitne upotrebe pokazivača "**`this`**", ovaj pokazivač se dosta rijetko koristi eksplicitno. Međutim, postoje situacije kada je njegova eksplicitna upotreba 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. Recimo, uzmimo da klasa "`Datum`" ima neku metodu koja se zove "`NekaMetoda`", koja treba da pozove neku običnu funkciju nazvanu "`NekaFunkcijaKojaPrimaDatumKaoParametar`" koja kao parametar prima objekat tipa "`Datum`" i kojoj je kao argument potrebno prenijeti objekat nad kojim je metoda "`NekaMetoda`" pozvana. Tada bi implementacija metode "`NekaMetoda`" mogla izgledati ovako:

```
void Datum::NekaMetoda() {
  ...
  NekaFunkcijaKojaPrimaDatumKaoParametar(*this);
  ...
}
```

Potreba za eksplicitnom upotrebom 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);
```

Primijetimo da još uvijek nismo riješili osnovni problem koji nas je motivirao da uopće uvedemo pojam klase! Naime, 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). Ovaj problem je riješen 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, pretpostavimo 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`" proglasili *privatnim*, dok smo metodu "`Postavi`" proglasili *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 kompajlera. 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 (treba naglasiti da neki autori ne prave razliku između enkapsulacije i sakrivanja informacija). Bitno je napomenuti da o objektno orijentiranom programiranju možemo govoriti tek ukoliko 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`", a 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();
};
```

Naravno, implementacija metode "`Ocitaj`" je veoma jednostavna:

```
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, nema nikakve štete od mogućnosti zasebnog čitanja atributa "dan", "mjesec" i "godina", dok mogućnost njihovog nekontroliranog mijenjanja nije poželjna (postoje situacije u kojima nije poželjno ni čitati izvjesne atribute klase). Njihovo čitanje možemo omogućiti tako što ćemo dodati tri trivijalne funkcije članice "DajDan", "DajMjesec" i "DajGodinu" bez parametara, koje će prosto vraćati kao rezultat njihove vrijednosti:

```
class Datum {
  int dan, mjesec, godina;
public:
  void Postavi(int d, int m, int g);
  void Ocitaj(int &d, int &m, int &g);
  int DajDan();
  int DajMjesec();
  int DajGodinu();
  void Ispisi();
};
```

Implementacija ove tri metode je trivijalna:

```
int Datum::DajDan() {
  return dan;
}
int Datum::DajMjesec() {
  return mjesec;
}
int Datum::DajGodinu() {
  return godina;
}
```

Sada možemo pisati i ovakve programske isječke:

```
Datum d;
d.Postavi(30, 12, 2002);
cout << "Dan = " << d.DajDan() << " Mjesec = " << d.DajMjesec()
  << "Godina = " << d.DajGodinu();
```

Na ovaj način omogućili smo čitanje atributa "dan", "mjesec" i "godina", doduše ne posve direktno, već pomoću pomoćnih funkcija članica "DajDan", "DajMjesec" i "DajGodinu". Takve funkcije članice obično se nazivaju *pristupne metode* (engl. *access methods*) i na engleskom govornom području gotovo uvijek dobijaju imena koja počinju sa "Get" (npr. "GetDay"). Na prvi pogled, ovakva indirekcija može djelovati 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.

Pri prvom susretu sa objektno orijentiranim programiranjem, mnogi programeri imaju odbojnost prema definiranju trivijalnih metoda poput "DajMjesec" itd. jer smatraju da je na taj način potrebno mnogo pisanja. Naime, svaku takvu metodu potrebno je kako *deklarirati*, tako i *implementirati*. Da bi se smanjila potreba za količinom pisanja, 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 (naravno, 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 DajDan() { return dan; }
    int DajMjesec() { return mjesec; }
    int DajGodinu() { return godina; }
    void Ispisi() {
      cout << dan << ". " << mjesec << ". " << godina << ".";
    }
  };
```

Važno je napomenuti da ipak 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, metoda se ne poziva kao klasična funkcija, nego se njeno tijelo prosto *umeće* u prevedeni kôd na mjestu poziva funkcije (tu govorimo o tzv. *umetnutim funkcijama*). Na taj način se dobija na efikasnosti, jer se ne gubi vrijeme na poziv funkcije, prenos parametara i povratak iz funkcije, ali s druge strane prevedeni kôd se time može nepotrebno produžiti, posebno ukoliko je tijelo funkcije dugačko. 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 o vrlo prostim naredbama poput naredbi dodjeljivanja), čime dobijamo na brzini. Sve ostale metode treba implementirati *izvan deklaracije klase* (ovo pogotovo vrijedi za metode koje sadrže petlje i druge složenije programske konstrukcije).

Treba napomenuti da se neka funkcija može proglasiti za umetnutu funkciju čak i ukoliko se implementira izvan deklaracije klase, pa čak i ukoliko se radi o klasičnoj funkciji (nevezanoj za ikakvu klasu), a ne funkciji članici. To postižemo uz pomoć modifikatora "**inline**". Na primjer, u sljedećoj implementaciji, funkcija članica "Ocitaj" proglašava se za umetnutu funkciju, bez obzira što je implementirana izvan deklaracije klase "Datum":

```
inline void Datum::Ocitaj(int &d, int &m, int &g) {
  d = dan; m = mjesec; g = godina;
}
```

Sve metode se mogu podijeliti u dvije kategorije. Metode iz prve kategorije samo *čitaju* atribute klase, ali ih *ne mijenjaju*. Takve su, u prethodnom primjeru, metode "Ocitaj", "DajDan", "DajMjesec", "DajGodinu" i "Ispisi". Takve metode nazivaju se *inspektori*. Metode iz druge kategorije, u koje spada recimo metoda "Postavi", *mijenjaju* atribute klase nad kojom su primijenjene. Takve funkcije nazivaju se *mutatori*. U jeziku C++ poželjno je 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 kompajleru 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 ekplicitno 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 PredjiNaSljedeciDan();
  void Ocitaj(int &d, int &m, int &g) const {
    d = dan; m = mjesec; g = godina;
  }
  int DajDan() const { return dan; }
  int DajMjesec() const { return mjesec; }
  int DajGodinu() const { return godina; }
```

```
    void Ispisi() const {
      cout << dan << ". " << mjesec << ". " << godina << ".";
    }
  };
```

Ovom prilikom smo u klasu "Datum" dodali još jednu zanimljivu metodu mutator nazvanu "PredjiNaSljedeciDan" koja mijenja objekat na koji je primijenjena tako da sadrži sljedeći dan po kalendaru. Njena implementacija bi mogla izgledati na primjer ovako:

```
  void Datum::PredjiNaSljedeciDan() {
    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.PredjiNaSljedeciDan();
  d.Ispisi();
```

Ključna riječ "**const**" kojom funkciju članicu proglašavamo konstantnom smatra se *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. Naime, 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 takvim 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 strukturnih 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*!

Klasu "Datum" možemo dalje proširiti sa još nekoliko korisnih metoda. Za sada ćemo još dodati metodu "DajImeMjeseca" 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. Obje 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 PredjiNaSljedeciDan();
  void Ocitaj(int &d, int &m, int &g) const {
    d = dan; m = mjesec; g = godina;
  }
  int DajDan() const { return dan; }
  int DajMjesec() const { return mjesec; }
  const char *DajImeMjeseca() const;
  int DajGodinu() const { return godina; }
  void Ispisi() const {
    cout << dan << ". " << mjesec << ". " << godina << ".";
  }
  void IspisiLijepo() const;
};
```

Primijetimo da smo metodu "`DajImeMjeseca`" imenovali koristeći istu konvenciju kao i pristupne metode poput "`DajMjesec`", iako ona ne vraća niti jedan atribut klase. Međutim, zgodno je da korisnik klase svim informacijama o primjercima te klase pristupa na jednoobrazan način, bez obzira da li se te informacije nalaze u nekom od atributa klase ili se one izvode na neki indirektan način (uostalom, korisnik klase uopće i ne treba da zna od čega se klasa interno sastoji, tj. šta su joj atributi). Sama implementacija metoda "`DajImeMjeseca`" i "`IspisiLijepo`" mogla bi izgledati ovako:

```
const char *Datum::DajImeMjeseca() const {
  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 << ". " << DajImeMjeseca() << " " << godina << ".";
}
```

Metoda "`IspisiLijepo`" unutar sebe poziva metodu "`DajImeMjeseca`". Međutim, interesantno je da se ona *ne poziva eksplicitno 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 "`DajImeMjeseca()`" interno interpretira kao "**`this`**`->DajImeMjeseca()`" odnosno kao "`(*`**`this`**`).DajImeMjeseca()`". 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 zaglavlju "iostream" deklariran kao instanca klase "istream", a "cout" kao instanca klase "ostream". Klasa "istream" definira skup atributa i metoda za pristupanje standardnom ulaznom toku, dok klasa "ostream" definira skup atributa i metoda za pristupanje standardnom izlaznom toku. Na primjer, jedna od metoda klase "istream" je metoda "getline", dok je jedna od metoda klase "ostream" metoda "width". Ovo objašnjava upotrebu ranije korištenih 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 "substr"), a isto vrijedi i za tipove podataka "complex" i "vector", koji su također 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". Kasnije ćemo vidjeti da je moguće definirati smisao gotovo svih operatora primijenjenih nad primjercima klase koju definiramo. Očigledno, ima još dosta stvari koje treba da naučimo o klasama, ali je ovdje 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 se uglavnom deklariraju u javnom dijelu klase. Međutim, ponekad nam je potrebna neka pomoćna metoda koja se može korisno poizvati iz neke druge metode, pri čemu *ne želimo dopustiti da se ta metoda može pozvati iz ostatka programa*. U tom slučaju, takvu metodu možemo deklarirati u privatnom dijelu klase. Na primjer, pretpostavimo da smo u prethodnom primjeru klase "Datum" metodu "DajImeMjeseca" deklarirali u privatnom dijelu klase, kao u sljedećoj deklaraciji:

```
class Datum {
  int dan, mjesec, godina;
  const char *DajImeMjeseca() const;
public:
  void Postavi(int d, int m, int g);
  void PredjiNaSljedeciDan();
  void Ocitaj(int &d, int &m, int &g) const {
    d = dan; m = mjesec; g = godina;
  }
  int DajDan() const { return dan; }
  int DajMjesec() const { return mjesec; }
  int DajGodinu() const { return godina; }
  void Ispisi() const {
    cout << dan << ". " << mjesec << ". " << godina << ".";
  }
  void IspisiLijepo() const;
};
```

U tom slučaju, metoda "DajImeMjeseca" 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).

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*. Međutim, nekada je potrebno imati atribute koji će zajednički dijeliti *svi primjerci iste klase*. Takvi atributi nazivaju se *statički atributi* i deklariraju se uz pomoć modifikatora "**static**". Neka, na primjer, imamo klasu "Student" koja je karakterizirana atributima "ime", "prezime", "indeks", "ocjene" i "prosjek" (sa očiglednim značenjima). Dalje, neka je potrebno imati informaciju o *ukupnom broju studenata*, pri čemu ova informacija treba biti dostupna svim primjercima klase "Student", bez obzira gdje se oni nalazili unutar programa. Također, pretpostavimo da samo metode klase "Student" trebaju pristupati ovoj informaciji. Jedno rješenje je definirati *globalnu promjenljivu* "broj_studenata" koja će čuvati traženu informaciju, ali njoj će tada moći pristupati svako, i ko treba i ko ne treba, tako da postoji mogućnost nehotičnog (ili namjernog) neovlaštenog pristupa. Bolja ideja je definirati "broj_studenata" kao atribut klase "Student". Međutim, tada bi svaki primjerak klase "Student" mogao postaviti različite vrijednosti ovog atributa, bez obzira što je jasno da je ukupan broj studentata svojstvo koje je zajedničko za sve primjerke klase "Student". Pravo rješenje je definirati "broj_studenata" kao *statički atribut klase* "Student", kao u sljedećoj deklaraciji:

```
class Student {
  static int broj_studenata;
  string ime, prezime;
  int indeks;
  int ocjene[50];
  double prosjek;
public:
  ...                    // Deklaracije metoda nam zasad nisu bitne
};
```

Ukoliko pretpostavimo da su "s1" i "s2" dva primjerka ovako deklarirane 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 primjerci klase "Student" dijele istu vrijednost atributa "broj_studenata", i njegova eventualna promjena izvršena iz neke metode klase "Student" primijenjene nad nekim konkretnim objektom biće vidljiva svim drugim primjercima 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 memorijski prostor za njegovo pamćenje). Svaki statički atribut se mora na nekom mjestu u programu na globalnom nivou (dakle, izvan definicija funkcija ili klasa) posebno definirati (pri čemu se tom prilikom može evenutalno izvršiti njegova inicijalizacija) deklaracijom poput sljedeće:

```
int Student::broj_studenata(10);
```

Razlog za ovu zavrzlamu 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. 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 donekle gubi smisao njihovog uvođenja. Na primjer, pretpostavimo 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 mjesta 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 konkretan 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 ma kakva

globalna promjenljiva, jedino bi nas sintaksa upućivala da se radi o podatku koji je na neki način logički vezan za klasu "`Student`" (u striktno objektno orjentiranim programskim jezicima kao što je npr. Java, ova konstrukcija koristi se za simulaciju globalnih promjenljivih, koje u takvim jezicima ne postoje).

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 tijesno vezana za samu klasu.

Razmotrimo statičke funkcije članice na konkretnom primjeru klase "`Datum`" koju smo razvili. Ukoliko uporedimo metode "`Postavi`" i "`PredjiNaSljedeciDan`", 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 izbjeći 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, kako ova funkcija ne treba pristupati atributima klase (s obzirom da 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). Pored toga, proglašavanje metode "`BrojDana`" za statičku metodu čini je i neznatno efikasnijom, jer se u tom slučaju njoj ne prenosi pokazivač "**`this`**" kao skriveni parametar. 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;
  const char *DajImeMjeseca() 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 njeno pozivanje izvan funkcija članica klase. Naravno, ovu metodu treba i implementirati. Njena implementacija, kao i izmijenjene implementacije metoda "`Postavi`" i "`PredjiNaSljedeciDan`", koje se oslanjaju na njeno postojanje, mogle bi 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::PredjiNaSljedeciDan() {
  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.

S obzirom da je metoda "BrojDana" definirana kao privatna, njeno 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" (ova konstrukcija se često koristi za simulaciju običnih funkcija, koje ne postoje u čisto objektno orjentiranim jezicima, poput Jave).

Unutar deklaracija klase mogu se naći i 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 interno 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 "Vektor3d" koja će opisivati vektor u prostoru, definiran sa tri koordinate "x", "y" i "z". One 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*), pri čemu ćemo uvesti metode "Postavi", "Ocitaj", "DajX", "DajY" i "DajZ" 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, uvešćemo i metodu "Ispisi" koja ispisuje vektor kao skupinu od tri koordinate unutar vitičastih zagrada razdvojene zarezima, zatim metodu "DajDuzinu" 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 (sve metode su dovoljno jednostavne da se mogu implementirati direktno na mjestu njihove deklaracije):

```
class Vektor3d {
  double x, y, z;
public:
  void Postavi(double x, double y, double z) {
    Vektor3d::x = x; Vektor3d::y = y; Vektor3d::z = z;
  }
  void Ocitaj(double &x, double &y, double &z) const {
    x = Vektor3d::x; y = Vektor3d::y; z = Vektor3d::z;
  }
  void Ispisi() const {
    cout << "{" << x << "," << y << "," << z << "}";
  }
  double DajX() const { return x; }
  double DajY() const { return y; }
  double DajZ() const { return z; }
  double DajDuzinu() const { return sqrt(x * x + y * y + z * z); }
  void PomnoziSaSkalarom(double s) { x *= s; y *= s; z *= s; }
  void SaberiSa(const Vektor3d &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 "Vektor3d" (koji će na ekranu ispisati "7.071068", "{15,20,25}" i "{22,23,27}"), iz čega vidimo da se ove dvije metode na izvjestan način ponašaju poput operatora "*=" i "+=":

```
Vektor3d v1, v2;
v1.Postavi(3, 4, 5);
v2.Postavi(7, 3, 2);
cout << v1.DajDuzinu() << endl;
v1.PomnoziSaSkalarom(5);
v1.Ispisi();
cout << endl;
v1.SaberiSa(v2);
v1.Ispisi();
```

Može se primijetiti da smo i u ovom slučaju metodu "DajDuzinu" koja vraća dužinu vektora imenovali na isti način kao i pristupne metode poput "DajX", čime ostvarujemo uniformirani pristup informacijama o primjercima klase "Vektor3d". Na taj način, korisnik klase nema potrebu da zna da dužina vektora nije neki od njegovih atributa, nego svojstvo koje se indirektno izvodi iz koordinata vektora (koje *jesu* njegovi atributi). I zaista, sasvim bi bilo moguće zamisliti implementaciju klase "Vektor3d" u kojoj koordinate ne bi bile atributi, nego bi atributi bili recimo dužina vektora i uglovi koje vektor zaklapa sa koordinatnim osama, pri čemu bi metode poput "DajX" računale odgovarajuće koordinate iz raspoloživih atributa (uz malu pomoć trigonometrije). Mada bi se implementacija takve klase značajno razlikovala od ovdje date implementacije, korisnik bi tako implementiranu klasu koristio na isti način kao i ovdje razvijenu klasu!

Bitno je naglasiti da princip sakrivanja podataka i enkapsulacije strogo naređuje da, osim ukoliko ne postoji veoma jak razlog za suprotno, sve atribute klase uvijek treba proglasiti za privatne. Na primjer, čak i ukoliko želimo da imamo mogućnost neovisnog postavljanja koordinata vektora, to ne treba učiniti tako što ćemo atribute "x", "y" i "z" prosto proglasiti javnim. Umjesto toga, potrebno je uvesti tri nove

trivijalne metode "PostaviX", "PostaviY" i "PostaviZ" koje će respektivno postavljati vrijednosti atributa "x", "y" odnosno "z" na vrijednost zadanu parametrom (na engleskom govornom području, metode tog tipa gotovo uvijek dobijaju imena koja počinju sa "Set", npr. "SetX"). Tako ćemo ukoliko želimo da postavimo "x" koordinatu vektora "v" na vrijednost "5", umjesto "v.x = 5" pisati "v.PostaviX(5)". Mada ovo djeluje kao potpuno suvišna komplikacija, postoje veoma jaki razlozi za ovakvu 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 "PostaviX", dovoljno je samo promijeniti njihovu implementaciju. U ostatku programa se i dalje može bez problema koristiti konstrukcija poput "v.PostaviX(5)". Slična stvar bi se dogodila i kada bismo recimo izmijenili implementaciju klase tako da se kao atributi vektora umjesto koordinata čuva dužina vektora i uglovi koje vektor zaklapa sa koordinatnim osama, ili ukoliko bismo na bilo koji drugi način izmijenili internu organizaciju klase "Vektor3d".

Neosporno je da objektno zasnovani pristup traži *više pisanja* u odnosu na klasični pristup, pogotovo u slučaju manjih programa. Međutim, takav pristup *nije ni namijenjen za pisanje malih programa*. U objektno zasnovanom 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 "Vektor3d", metode "PomnoziSaSkalarom" i "SaberiSa" napisane su kao metode koje ne vraćaju nikakav rezultat, što može djelovati prirodno. Ipak, zbog toga neke također prirodne konstrukcije poput "v2 = v1.PomnoziSaSkalarom(5)" nisu izvodljive (s obzirom da metoda "PomnoziSaSkalarom" ne vraća ništa kao rezultat što bismo mogli pridružiti drugom vektoru).. Stoga bismo 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 postale ispravne i konstrucije poput "v2 = v1.PomnoziSaSkalarom(5)" pa čak i lančane (kaskadne) konstrukcije poput "v1.PomnoziSaSkalarom(5).Ispisi()". Zaista, poziv poput "v1.PomnoziSaSkalarom(5)" će kao rezultat vratiti objekat tipa "Vektor3d", na koji se onda može primijeniti metoda "Ispisi". Tako dobijamo sljedeću verziju klase "Vektor3d":

```
class Vektor3d {
  ...  // Ovdje treba umetnuti definicije atributa i preostalih metoda
  Vektor3d PomnoziSaSkalarom(double s) {
    x *= s; y *= s; z *= s;
    return *this;
  }
  Vektor3d SaberiSa(const Vektor3d &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:

```
Vektor3d v1, v2;
v1.Postavi(3, 4, 5);
v2.Postavi(7, 3, 2);
v1.PomnoziSaSkalarom(5).Ispisi();
cout << endl;
v1.SaberiSa(v2).Ispisi();
```

Međutim, sljedeći primjer će nas uvjeriti da nije baš sve 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}`":

```
Vektor3d v1, v2;
v1.Postavi(3, 4, 5);
v2.Postavi(7, 3, 2);
v1.PomnoziSaSkalarom(5).SaberiSa(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`**" nađe *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 klasa, nego predstavlja *opće pravilo kako se ma kakve vrijednosti vraćaju iz ma kakve funkcije*. Mada o ovakvom ponašanju nismo do sada posebno razmišljali, ono nam nikada do sada nije smetalo. Međutim, razmotrimo u ovom kontekstu značenje sljedeće konstrukcije:

```
v1.PomnoziSaSkalarom(5).SaberiSa(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 "`Vektor3d`", koji je njegova potpuno identična bezimena kopija! Na tu kopiju (koja također ima vrijednost "`{15,20,25}`") primijenjuje se metoda "`SaberiSa(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 vektorom primijenjuje se metoda "`Ispisi()`" koja naravno ispisuje "`{22,23,27}`". Međutim, samo se prva metoda "`PomnoziSaSkalarom(5)`" zaista izvršila nad vektorom "`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 "`SaberiSa`" treba da umjesto samog objekta tipa "`Vektor3d`", koji je *kopija* objekta koji želimo da vratimo, vrate *referencu* na objekat tipa "`Vektor3d`" koja predstavlja *alternativno ime* za objekat koji vraćamo, odnosno koja se *u potpunosti poistovjećuje* s objektom koji vraćamo. Stoga je u klasi "`Vektor3d`" dovoljno izvršiti samo neznatnu izmjenu, kao u sljedećoj deklaraciji:

```
class Vektor3d {
  ...  // Ovdje treba umetnuti definicije atributa i preostalih metoda
  Vektor3d &PomnoziSaSkalarom(double s) {
    x *= s; y *= s; z *= s;
    return *this;
  }
```

```
    Vektor3d &SaberiSa(const Vektor3d &v) {
      x += v.x; y += v.y; z += v.z;
      return *this;
    }
  };
```

Sada će ulančani poziv poput "v1.PomnoziSaSkalarom(5).SaberiSa(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 se u potpunosti poistovjećuje s njim, tako da će sljedeći poziv metode "SaberiSa(v2)" zaista djelovati na vektor "v1", što smo i željeli da postignemo.

U prethodnim primjerima smo koristili sintaksu poput "v1.SaberiSa(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, sabiranje vektora bi bilo prirodnije izraziti čisto proceduralnom sintaksom poput "ZbirVektora(v1, v2)", pogotovo u kontekstu dodjele rezultata nekom trećem vektoru (npr. "v3 = ZbirVektora(v1, v2)"). Jasno je da time što smo definirali tip "Vektor3d" kao klasu nismo izgubili pravo da pišemo *obične funkcije* (koje nisu funkcije članice) koje kao parametre primaju objekte tipa "Vektor3d", i koje vraćaju objekte tipa "Vektor3d" 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 primijenjenu 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 "Vektor3d" (upravo zbog činjenice da ona nije funkcija članica nego obična funkcija). Zbog toga ćemo, dok ne upoznamo bolje rješenje, ovu funkciju napisati na indirektan način, koristeći funkcije članice "Postavi", "DajX", "DajY" i "DajZ" pomoću kojih ipak možemo posredno pristupiti atributima klase "Vektor3d":

```
Vektor3d ZbirVektora(const Vektor3d &v1, const Vektor3d &v2) {
  Vektor v3;
  v3.Postavi(v1.DajX() + v2.DajX(), v1.DajY() + v2.DajY(),
    v1.DajZ() + v2.DajZ());
  return v3;
}
```

Neko bi mogao prigovoriti da i sintaksa poput "v.DajDuzinu()" 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, i dužina vektora i njegove koordinate su izvjesne informacije *o vektoru* kojima je dobro pristupati na jednoobrazan način, bez obzira što su u konkretnoj implementaciji koordinate vektora izvedene kao njegovi atributi, dok se dužina posredno *računa* na osnovu koordinata. Već smo rekli da to korisnik klase *ne treba da zna*, tim prije što su sasvim moguće i drugačije implementacije za koje ovo ne vrijedi. S druge strane, sintaksa poput "Duzina(v)" središte pažnje prenosi na funkciju "Duzina", pri čemu se jasno naglašava da dužina vektora nije njegovo *svojstvo* nego se ona *računa* posredstvom funkcije "Duzina" koja kao *argument* uzima vektor a daje kao rezultat njegovu *dužinu*. Vidimo da se radi o klasičnom proceduralnom razmišljanju. 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)":

```
double Duzina(const Vektor3d &v) {
  return v.DajDuzinu();
}
```

Primijetimo da je definicija funkcije "Duzina" trivijalna, jer smo prosto iskoristili metodu "DajDuzinu" nad objektom koji joj je proslijeđen kao parametar, koja upravo radi ono što nam i treba. Interesantno je

da mogu postojati i funkcija članica i obična funkcija sa istim imenom, bez opasnosti da ih kompajler 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). Ipak, ukoliko bismo iz neke funkcije članice željeli pozvati *istoimenu* običnu funkciju, morali bismo ispred njenog imena pri pozivu dodati prefiks "`::`" (bez navođenja imena ikakve klase ispred). Na primjer, funkciju "`Duzina`" bismo iz istoimene funkcije članice morali pozivati sa "`::Duzina(v)`", jer bi u suprotnom bilo shvaćeno da ta funkcija članica poziva samu sebe (tj. kao rekurzija)!

Vratimo se na implementaciju funkcije "`ZbirVektora`". Možemo reći da smo imali sreće što klasa "`Vektor3d`" ima metode "`DajX`", "`DajY`" i "`DajZ`", 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:

```
Vektor3d ZbirVektora(const Vektor3d &v1, const Vektor3d &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);
  Vektor3d 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 napisati ovu funkciju. S druge strane, najprirodnija verzija funkcije "`ZbirVektora`" mogla bi se napisati kada bi ova funkcija *imala pravo pristupa privatnim atributima* klase "`Vektor3d`". Ona bi se tada mogla napisati prosto ovako (poput istoimene funkcije sa prethodnih predavanja, dok je tip "`Vektor3d`" bio samo *struktura*, a ne *klasa sa privatnim atributima*):

```
Vektor3d ZbirVektora(const Vektor3d &v1, const Vektor3d &v2) {
  Vektor3d 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) proglasiti *prijateljima klase*, čime one dobijaju pravo pristupa privatnim atributima i metodama te klase. Neku funkciju je moguće proglasiti 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 "`Vektor3d`" koja proglašava funkciju "`ZbirVektora`" prijateljem klase, što omogućava njenu izvedbu na gore prikazani način:

```
class Vektor3d {
  double x, y, z;
public:
  void Postavi(double x, double y, double z) {
    Vektor3d::x = x; Vektor3d::y = y; Vektor3d::z = z;
  }
  void Ocitaj(double &x, double &y, double &z) const {
    x = Vektor3d::x; y = Vektor3d::y; z = Vektor3d::z;
  }
  void Ispisi() const {
    cout << "{" << x << "," << y << "," << z << "}";
  }
  double DajX() const { return x; }
  double DajY() const { return y; }
  double DajZ() const { return z; }
  double DajDuzinu() const { return sqrt(x * x + y * y + z * z); }
```

```
   Vektor3d &PomnoziSaSkalarom(double s) {
     x *= s; y *= s; z *= s;
     return *this;
   }
   Vektor3d &SaberiSa(const Vektor3d &v) {
     x += v.x; y += v.y; z += v.z;
     return *this;
   }
   friend Vektor3d ZbirVektora(const Vektor3d &v1, const Vektor3d &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 "Vektor3d". 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* (engl. *friend functions*) ili *funkcije prijatelji klase*. 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 modifikator "**inline**").

Interesantno je napomenuti da postoji i alternativno rješenje kojim se postiže slična funkcionalnost kao pomoću prijateljskih funkcija, a u kojima se ipak ne koriste obične funkcije, nego samo funkcije članice. Ideja je da se umjesto običnih funkcija koriste *statičke funkcije članice*, koje kao funkcije članice svakako imaju pravo pristupa privatnim dijelovima klase. U konkretnom slučaju koji razmatramo, funkcija "ZbirVektora" bi, umjesto obične funkcije, postala statička funkcija članica:

```
class Vektor3d {
  ...  // U ostatku klase ništa se ne mijenja
  static const Vektor3d ZbirVektora(const Vektor3d &v1,
    const Vektor3d &v2);
};
```

Implementacija ove funkcije članice bila bi praktično identična kao i u slučaju kada smo koristili klasičnu funkciju (obratite pažnju samo na kvalifikator "Vektor3d::" u deklaraciji funkcije):

```
const Vektor3d Vektor3d::ZbirVektora(const Vektor3d &v1,
  const Vektor3d &v2) {
    Vektor3d v3;
    v3.x = v1.x + v2.x; v3.y = v1.y + v2.y; v3.z = v1.z + v2.z;
    return v3;
}
```

Jedini nedostatak ovog rješenja je što sada umjesto konstrukcije "v3 = ZbirVektora(v1, v2)" moramo koristiti rogobatniju konstrukciju "v3 = Vektor3d::ZbirVektora(v1, v2)". Bez obzira na rogobatnost, neki smatraju da je ova sintaksa logičnija i više "u duhu" objektno zasnovanog programiranja, jer jasno ističe logičku pripadnost funkcije "ZbirVektora" klasi "Vektor3d" (u ovom slučaju možda bi dobra ideja bila preimenovati naziv funkcije "ZbirVektora" prosto u "Zbir", jer već sam kvalifikator "Vektor3d::" jasno ističe da se radi o zbiru vektora a ne nečega drugog). Ipak, u jeziku C++ se češće koriste rješenja zasnovana na prijateljskim funkcijama. Ovdje prikazano rješenje koje koristi statičke funkcije članice uglavnom se koristi u čisto objektno orjentiranim jezicima kao što su Java, C# i srodni kod kojih klasične funkcije (ne-članice) uopće ne postoje.

Neka klasa može bilo koju funkciju proglasiti 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 *deklarira lokalnu varijablu* tipa te klase (naravno, moguće je da se dešava i sve ovo skupa, kao u primjeru navedene funkcije "ZbirVektora").

Treba napomenuti da su dugo vremena postojale intenzivne filozofske rasprave o tome da li prijateljske funkcije štete objektno orijentiranoj filozofiji. Gotovo sve tvrdnje koje su išle na ruku takvoj tezi danas su odbačene, tako da se smatra da prijateljske funkcije načelno nisu nikakva smetnja objektno orijentiranom pristupu programiranju. Naime, one ne narušavaju nijedan od postulata objektno orjentiranog programiranja, jedino što postiču sintaksu koja donekle nije u duhu objektno orijentiranog programiranja. Međutim, objektno orijentirano programiranje ne čini sintaksa (ona ga samo dodatno podstiče), nego poštovanje izvjesnih postulata. Čak postoje mnogi slučajevi u kojima poštovanje čisto objektno orijentirane sintakse može dovesti do zabune, tako da je u takvim slučajevima bolje koristiti klasične funkcije i proceduralnu sintaksu (ovim se ne mora nužno pokvariti filozofija objektno orijentiranog programiranja). Razmotrimo, na primjer, funkcije članice "`PomnoziSaSkalarom`" i "`SaberiSa`" klase "`Vektor3d`". 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 mogle bi izgledati recimo ovako:

```
class Vektor3d {
  ...   // Osim naredne dvije funkcije, sve ostaje isto...
  Vektor3d PomnoziSaSkalarom(double s) const {
    Vektor3d v;
    v.x = x * s; v.y = y * s; v.z = z * s;
    return v;
  }
  Vektor3d SaberiSa(const Vektor3d &v) const {
    Vektor3d v1;
    v1.x = x + v.x; v1.y = y + v.y; v1.z = z + v.z;
    return v1;
  }
};
```

Ovakve metode se ne smatraju dobrim, zbog toga što mogu biti zbunjujuće. Na primjer, pogledajmo sljedeću naredbu, uz pretpostavku da imamo ovako definiranu klasu "`Vektor3d`" i dva primjerka "`v1`" i "`v2`" ove klase:

```
v1.SaberiSa(v2);
```

Šta radi ova naredba? Efektivno gledano, *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 nizašta (tj. ignorira se), tako da, na kraju, ova naredba ne ostavlja nikakav efekat. Ovako definirana metoda ima smisla jedino ukoliko 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 zbunjujuća nego korisna. Njeno ime i sintaksa ukazuju na to da će sadržaj vektora "`v1`" biti izmijenjen, a zapravo neće. Ranije napisana funkcija "`ZbirVektora`" je mnogo jasnija, tim prije što ona oba svoja parametra tretira ravnopravno. Ista primjedba bi vrijedila za izmijenjenu verziju metode "`PomnoziSaSkalarom`".

Razmotrimo još jedan primjer. Klasa "`Datum`" imala je metodu mutator "`PredjiNaSljedeciDan`" koja *modificira* objekat na koji je primijenjena, 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 zbunjujuća 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:

```
Datum SljedeciDan(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 zbunjujuće*. Ukoliko ipak želimo definirati i takve metode, zbrku možemo smanjiti ukoliko im pažljivo izaberemo ime. Na primjer, u prethodnom primjeru, konfuziju bismo mogli osjetno smanjiti ukoliko bismo metodu koja vraća sljedeći datum umjesto "`PredjiNaSljedeciDan`" nazvali "`DajSljedeciDan`". Na taj način, samo ime metode sugerira da ona ne mijenja objekat na koji djeluje, već samo *vraća* novi objekat dobijen nakon obavljene transformacije. Slično, u slučaju modificirane klase "`Vektor3d`", zbrka bi mogla biti manja ukoliko bi se metode "`PomnoziSaSkalarom`" odnosno "`SaberiSa`" preimenovale u "`DajProduktSaSkalarom`" odnosno "`DajZbirSa`". Međutim, najbolje je takve metode uopće ne praviti, nego umjesto njih koristiti klasične funkcije, ukoliko nam je već potrebna takva funkcionalnost.

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;
```

Na taj način se čime se čitava klasa "`B`" proglašava prijateljem klase "`A`", te kažemo da klasa "`A`" ima klasu "`B`" za *prijateljsku klasu* (engl. *friend class*). To preciznije znači da 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 predavanja je uvod 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 potvrdan, uvjerićemo se u narednim izlaganjima, 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.

Predavanje 9.

Na prethodnim predavanjima 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 promjenljive čiji je tip neka klasa zasada imaju nedefiniran sadržaj sve dok im se eksplicitno ne dodijeli vrijednost (na sličnu pojavu kod promjenljivih čiji su tipovi prosti ugrađeni tipovi poput tipa "**`int`**" smo se već davno navikli). Ovaj problem bi se doduše mogao riješiti tako što se promjenljiva inicijalizira odmah pri deklaraciji (što bismo mogli uraditi tako što bismo promjenljivoj prilikom inicijalizacije dodijelili rezultat neke funkcije koja vraća neki objekat te klase kao rezultat). Alternativno, mogli bismo odmah nakon deklaracije promjenljive nad njom pozvati neku metodu (nazvanu, 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. Ozbiljnost ovih grešaka raste sa porastom složenosti tipova promjenljivih. Stoga objektno zasnovano programiranje nudi izlaz iz ove situacije uvođenjem *konstruktora*, koji *primoravaju* 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 eksplicitno.

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* (bez obzira na te sličnosti, konstruktori *nisu funkcije*). Jedna klasa može imati više konstruktora, koji se tada moraju razlikovati po broju i/ili tipu parametara. Konstruktori također, slično kao i funkcije, mogu imati i parametre sa *podrazumijevanim vrijednostima*.

Upotrebu konstruktora ćemo prvo ilustrirati na jednom jednostavnom primjeru. Pretpostavimo da smo razvili klasu "`Kompleksni`" koja treba da predstavlja kompleksni broj (značenje atributa i metoda prikazane klase jasni su sami po sebi). Predviđeno je i postojanje jedne prijateljske funkcije nazvane "`ZbirKompleksnih`", koju ćemo implementirati kasnije. Ono na šta ovdje treba obratiti 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 DajRealni() const { return re; }
  double DajImaginarni() const { return im; }
  friend 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 sekvenca naredbi

```
Kompleksni a;
cout << a.DajRealni() << " " << a.DajImaginarni();
```

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(new Kompleksni);
cout << pok->DajRealni() << " " << pok->DajImaginarni();
```

Ovdje je bitno istaći da se dinamičko kreiranje objekata pomoću operatora "**new**" generalno izvodi u *dvije etape*. U prvoj etapi se pronalazi i zauzima prostor za traženu dinamičku promjenljivu, dok se u drugoj etapi poziva odgovarajući konstruktor (ako takav postoji) sa ciljem da izvrši inicijalizaciju novostvorene dinamičke promjenljive.

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 slaže sa navedenim parametrima, kompajler 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(0), *pb(0), *pc(0), *pd(0), *pe(0);
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();
```

U načelu, što se tiče sintakse i semantike, ništa nas ne sprečava da odmah izvršimo inicijalizaciju pokazivača na adrese dinamički kreiranih objekata, kao u sljedećem primjeru:

```
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();
```

Ovakve konstrukcije se ipak ne preporučuju. Naime, zbog eventualnih memorijskih problema pri dinamičkoj alokaciji objekata, sekvenca instrukcija u kojoj se vrši dinamička alokacija trebala bi se izvršavati unutar "**try**" bloka. Ukoliko bismo to učinili sa isječkom instrukcija iz posljednjeg primjeru, mogli bismo imati problema sa curenjem memorije u slučaju da, na primjer, alokacija prva dva objekta uspije a alokacija trećeg objekta ne uspije (problem bi bio kako obrisati prva dva alocirana objekta nakon bacanja izuzetka). Sa isječkom instrukcija iz prethodnog primjera već bi bilo lakše, jer su svi pokazivači na početku inicijalizirani na nul-pokazivače, pa bi kasnija upotreba operatora "**delete**" prosto obrisala samo objekte koji su zaista i kreirani (s obzirom na poznatu osobinu operatora "**delete**" da ne radi ništa ukoliko se primijeni na nul-pokazivač).

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 zagradi, 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*.

Važno je uočiti da se u slučajevima kada želimo izvršiti automatsku inicijalizaciju objekta pomoću konstruktora bez parametara par zagrada *ne piše*. Sintaksa doduše dopušta (ali ne zahtijeva) da se ovaj par zagrada piše ukoliko vršimo dinamičko kreiranje objekata pomoću operatora "**new**". Stoga je sljedeća konstrukcija također sintaksno ispravna (i radi istu stvar kao i kada bismo izostavili zagrade):

```
pa = new Kompleksni();
```

Mada će se sigurno naći pristalice ovakve sintakse (tim prije što programski jezik Java zahtijeva upravo ovakvu sintaksu), ona nije preporučljiva u jeziku C++ zbog činjenice da se prazan par zagrada *ne smije upotrijebiti* pri klasičnom definiranju promjenljivih. Drugim riječima, promjenljivu "a" tipa "Kompleksni" koju želimo automatski inicijalizirati konstruktorom bez parametara ne smijemo deklarisati ovako:

```
Kompleksni a();
```

Najgore je od svega što je gore napisana konstrukcija *sintaksno ispravna* u jeziku C++, ali ne radi ono što se od nje na prvi pogled očekuje. Zaista, lako je vidjeti da ova konstrukcija zapravo predstavlja *prototip funkcije* sa imenom "a" koja *nema parametara*, i koja *vraća objekat tipa* "Kompleksni" *kao rezultat*! Upravo zbog te činjenice, napišemo li zabunom takvu konstrukciju, možemo kasnije očekivati prijavu bizarnih grešaka od strane kompajlera. Na primjer, u sekvenci naredbi

```
Kompleksni a();
a.Ispisi();
```

kompajler će se pobuniti na drugu naredbu. Naime, na osnovu prve naredbe, kompajler će shvatiti da "a" zapravo predstavlja *funkciju* a ne promjenljivu (što nam vjerovatno nije bila namjera), te će se pobuniti pri pokušaju da na funkciju primijenimo operator ".".

Važno je shvatiti da konstruktori *nisu funkcije članice*, iako sintaksno podsjećaju na njih. Recimo, konstruktori se *nikada ne mogu pozivati nad nekim objektom* poput funkcija članica. Drugim riječima, *ne možemo* pisati nešto poput

```
a.Kompleksni(2, 3);
```

sa ciljem da *već postojećoj* promjenljivoj "a" postavimo vrijednost na "(2,3)" (za tu svrhu treba koristiti metodu "Postavi", što objašnjava razlog za postojanje ove metode, čije je tijelo praktično identično tijelu konstruktora). Pokušaj da uradimo nešto ovako dovešće do prijave sintaksne greške. Drugim riječima, konstruktori se koriste za *inicijalizaciju* objekata, dok za njihovu *naknadnu izmjenu* treba koristiti druge funkcije članice. Uskoro ćemo vidjeti da se konstruktori ipak mogu i *eksplicitno pozvati*, ali na drugačiji način (i uz sasvim drugačiji smisao) nego što se pozivaju funkcije članice.

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 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 konstruktore, 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 na prethodnim predavanjima), 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 << dan << ". " << mjesec << ". "
    << godina;
  }
};
```

Pri tome, funkcija "Postavi" će biti implementirana na isti način kao i na prethodnim predavanjima:

```
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. U suštini, čak i ukoliko se metode klase implementiraju odmah unutar deklaracije klase, kompajler ih tretira kao da se na tom mjestu nalaze samo njihovi *prototopovi*, pri čemu se zamišlja da se implementacije nalaze *neposredno iza završetka deklaracije klase*.

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 kreacija poput

```
Datum *pok(new Datum);
```

ali je sljedeća kreacija posve korektna:

```
Datum *pok(new Datum(7, 12, 2003));
```

Nije na odmet još jednom ukazati na razloge zbog kojih smo definirali i metodu "`Postavi`" koja radi istu stvar kao i konstruktor. Naime, već je rečeno da se konstruktor 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);
```

Činjenica da konstruktor klase "`Datum`" i njena metoda "`Postavi`" rade praktično istu stvar može dovesti neiskusnog programera u zabludu. Naime, neko bi mogao doći na ideju da neophodne radnje za postavljanje datuma implementira *unutar konstruktora*, a da iz funkcije "`Postavi`" pozove konstruktor, kao recimo u sljedećoj (neispravnoj) definiciji klase "`Datum`":

```
class Datum {
  int dan, mjesec, godina;
public:
  Datum(int d, int m, int g);
  void Postavi(int d, int m, int g) { Datum(d, m, g); }
  void Ispisi() const { cout << dan << ". " << mjesec << ". "
    << godina;
  }
};
```

Pri tome bi konstruktor ovdje bio implementiran onako kako je bila implementirana funkcija članica "`Postavi`":

```
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;
}
```

Nažalost, ovo je još jedna od podmuklih zamki jezika C++. Ovako napisana klasa *kompajlira se bez problema*, odnosno kompajler ne prijavljuje nikakve greške (s obzirom da sintaksnih grešaka ovdje zaista nema). Međutim, ova klasa neće raditi kako je očekivano (preciznije, metoda "`Postavi`" neće raditi ispravno). Razlog za to je pokušaj da se iz metode "`Postavi`" pozove konstruktor na način kako se inače pozivaju funkcije članice, a već je rečeno da se konstruktori *ne mogu tako pozivati*. U ovom trenutku ostaje doduše nejasno *zbog čega nam je kompajler uopće dozvolio ovakav poziv* i *kakvo je njegovo dejstvo*. Ubrzo ćemo saznati i odgovor na ovo pitanje, a za sada je bitno zapamtiti da *ovako ne treba raditi*.

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, ili eventualno unutar neke od funkcija prijatelja klase ili metoda prijateljske klase (ovo može da ima smisla ukoliko je neka klasa potrebna samo interno za potrebe neke druge klase, pri čemu želimo drugima onemogućiti kreiranje njenih primjeraka). 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 praktično svaki kompajler za C++ sigurno optimizirati stalno zauzimanje memorije za objekat 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 tzv. *destruktor*, i on će se pozivati 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 izbjeći ukoliko lokalnu promjenljivu "`d`" deklariramo kao *statičku* (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 nam je bitno da ograničimo 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.

Ranije smo vidjeli da se primjerci struktura 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, primjerci klasa se *ne mogu ovako inicijalizirati*. Za njihovu inicijalizaciju koriste se konstruktori. Preciznije, primjerak neke klase se ne može inicijalizirati listom vrijednosti u vitičastim zagradama ukoliko klasa 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). Naime, inicijalizacija listom vrijednosti u vitičastim zagradama 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 atribute, 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 kasnije da je kreiranje dinamički alociranih nizova najbolje prepustiti upravo konstruktorima).

Već smo najavili da se konstruktor neke klase može i *eksplicitno pozvati*. Međutim, pri eksplicitnom pozivu, konstruktor klase se poziva kao *obična funkcija*, a ne kao funkcija članica (odnosno 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* (svi tako kreirani privremeni objekti automatski se uništavaju kada se ustanovi da više nisu potrebni, tako da se ne moramo plašiti da će njihovo stvaranje dovesti do curenja memorije). Dakle, konstruktor uvijek kao rezultat vraća primjerak klase kojoj pripada. Da bismo vidjeli zbog čega je ovo korisno, pretpostavimo da imamo neku funkciju nazvanu "NekaFunkcija" koja prima objekat tipa "Datum" kao parametar:

```
void NekaFunkcija(const Datum &d) {
  ...                              // Nebitno je šta funkcija radi
}
```

Pretpostavimo sada da ovoj funkciji treba proslijediti neki konkretan datum, recimo 12. 5. 2007. To bismo mogli uraditi recimo ovako:

```
Datum dat(12, 5, 2007);
NekaFunkcija(dat);
```

Mada ovaj isječak nesumnjivo radi, njegov nedostatak je što smo morali deklarirati neku promjenljivu tipa "Datum" (u ovom primjeru nazvanu "dat") samo da bismo mogli formirati neophodne informacije koje treba proslijediti funkciji. Međutim, zahvaljujući činjenici da se konstruktori mogu pozivati kao funkcije, istu stvar možemo uraditi i ovako, bez kreiranja ikakve pomoćne promjenljive:

```
NekaFunkcija(Datum(12, 5, 2007));
```

Naime, poziv "Datum(12, 5, 2007)" kreiraće pomoćni bezimeni objekat tipa "Datum" koji će biti inicijaliziran na 12. 5. 2007. i koji će nakon toga biti proslijeđen funkciji "NekaFunkcija". Činjenica da novostvoreni objekat nema imena nimalo ne smeta, jer je njegova jedina uloga da bude proslijeđen funkciji kao parametar. Treba ipak obratiti pažnju na jedan sitni detalj po kojem prvi isječak koji kreira konkretnu imenovanu promjenljivu "dat" i drugi isječak koji izbjegava kreiranje imenovane promjenljive ipak nisu posve ekvivalentni. Naime, bezimeni privremeni objekti koji se vraćaju kao rezultat u slučaju da se konstruktor pozove kao funkcija *nemaju status l-vrijednosti* (kao što status l-vrijednosti nema niti jedna vrijednost vraćena iz neke funkcije osim u slučaju da funkcija vraća referencu kao rezultat). Posljedica toga je da se obične (nekonstantne) reference *ne mogu vezati na bezimene objekte vraćene kao rezultat iz konstruktora*, dok se konstantne reference mogu vezati na njih (kao što se, uostalom, mogu vezati i na bilo koji izraz, bio on l-vrijednost ili ne). Drugim riječima, kada bi formalni parametar "d" u funkciji "NekaFunkcija" bio obična (nekonstantna) referenca, prvi isječak koji koristi konkretnu promjenljivu "dat" radio bi konkretno, dok bi drugi isječak doveo do prijave sintaksne greške.

Činjenica da se konstruktori mogu eksplicitno pozvati kao funkcije, daju mogućnost da 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, primjerci klasa se, poput primjeraka struktura, također mogu dodjeljivati jedan drugom pomoću operatora "=". Pri tome se, kao i kod struktura, svi atributi objekta sa desne strane prosto kopiraju u odgovarajuće atribute objekta sa lijeve strane, ukoliko nije drugačije rečeno (naime, kasnije ćemo vidjeti da je, u slučaju potrebe, moguće definirati i drugačije ponašanje operatora dodjele "="). 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, njegovim kopiranjem u objekat koji želimo postaviti i, na kraju, njegovim uništavanjem kada više nije potreban.

Sada smo u stanju u cijelosti objasniti šta je loše u implementaciji metode "Postavi" koja eksplicitno poziva konstruktor klase "Datum". Podsjetimo se da je ta implementacija izgledala ovako:

```
class Datum {
  ...
  void Postavi(int d, int m, int g) { Datum(d, m, g); }
  ...
};
```

Problem je u tome što se ovaj eksplicitni poziv konstruktora iz metode "`Postavi`" ne izvodi nad istim objektom nad kojim se poziva metoda "`Postavi`". Tačnije, poziv konstuktora se uopće ne izvodi *ni nad kakvim objektom*! Zaista, u skladu sa gore opisanim objašnjenjem, eksplicitni poziv konstruktora "`Datum(d, m, g)`" kreiraće *posve novi* (bezimeni) objekat (koji nema nikakve veze sa objektom nad kojim je funkcija članica "`Postavi`" pozvana) i inicijalizirati ga u skladu sa vrijednostima parametara. Kako se sa tim novostvorenim objektom nakon toga ne radi ništa, on će odmah potom biti i uništen, bez ikakvog utjecaja na objekat nad kojim je pozvana funkcija članica "`Postavi`". Drugim riječima, ovako napisana funkcija članica "`Postavi`" radi niz beskorisnih stvari i na kraju ne proizvodi nikakav vidljiv efekat (odnosno, efektivno gledano, ne radi *ama baš ništa*). Doduše, postoji ipak jedan način da ispravno napišemo funkciju članicu "`Postavi`" a da se u njoj iskoristi eksplicitan poziv konstruktora. Taj način bi mogao izgledati recimo ovako:

```
class Datum {
  ...
  void Postavi(int d, int m, int g) { *this = Datum(d, m, g); }
  ...
};
```

Ukoliko ste razumjeli do sada izložene koncepte, ne bi trebalo predstavljati problem shvatiti kako ova konstrukcija radi. Međutim, ovom rješenju moglo bi se dosta toga prigovoriti sa aspekta efikasnosti (zbog razloga koji bi također trebali biti jasni), tako da se ono definitivno *ne preporučuje*.

Da bismo bolje shvatili suštinu inicijalizacije primjeraka klasa, 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 sljedeći (ukoliko Vam sintaksne konstrukcije koje ovdje koristimo djeluju odnekud poznate, to je zbog toga što tip "`complex`" koji smo koristili ranije nije ništa drugo nego klasa definirana u istoimenoj biblioteci):

```
Kompleksni a(1, 2);
```

Međutim, postoje i drugi načini da se u konačnici postigne isti efekat. Da bismo to shvatili, istaknimo da se većina objekata može inicijalizirati pomoću nekog objekta (ili općenitije, izraza) *istog tipa*. Pri tome dolazi do *kopiranja* objekta kojim vršimo inicijalizaciju u objekat koji inicijaliziramo. Za takvu inicijalizaciju se može koristiti bilo konstruktorska sintaksa (sa zagradama), bilo sintaksa koja koristi znak "`=`". Iznimka od ovog pravila nastaje samo ukoliko je klasa koja opisuje objekat specijalno dizajnirana tako da se njeni primjerci ne mogu kopirati (kasnije ćemo vidjeti kako se to može postići). U tom slučaju, opisani način inicijalizacije nije moguć. Kao ilustraciju inicijalizacije nekog objekta drugim objektom istog tipa, pretpostavimo da je data deklaracija

```
Kompleksni b(1, 2);
```

koja inicijalizira promjenljivu "`b`" na kompleksnu vrijednost "`(1,2)`". Sada promjenljivu "`a`" možemo inicijalizirati na istu vrijednost pomoću jedne od sljedeće dvije konstrukcije:

```
Kompleksni a(b);
Kompleksni a = b;
```

Mada postoji način da se klasa dizajnira tako da druga od ove dvije inicijalizacije (pomoću znaka "`=`") budu zabranjene, takva zabrana se gotovo nikada ne uvodi (barem ne kod ovog tipa inicijalizacije, gdje se objekat inicijalizira drugim objektom istog tipa). Stoga se ove dvije inicijalizacije mogu smatrati ekvivalentnim (istine radi, treba reći da postoje izvjesni prljavi trikovi pomoću kojih se može kreirati takva klasa kod koje bi obje gore prikazane inicijalizacije bile sintaksno ispravne a proizvodile različito dejstvo, mada teško da postoji ikakav razuman razlog zbog kojeg bi projektant klase izveo tako nešto).

Nakon što smo se upoznali sa mogućnošću da se neki objekat inicijalizira drugim objektom ili izrazom istog tipa, postaje jasno da inicijalizaciju promjenljive "a" na vrijednost "(1,2)" možemo izvesti i na jedan od sljedeća dva načina:

```
Kompleksni a(Kompleksni(1, 2));
Kompleksni a = Kompleksni(1, 2);
```

Naime, ovdje se prvo konstrukcijom "Kompleksni(1, 2)", koja predstavlja poziv konstruktora kao funkcije, kreira privremeni bezimeni objekat tipa "Kompleksni" inicijaliziran na vrijednost "(1,2)" koji se potom koristi za inicijalizaciju promjenljive "a". Mada bi ovakve inicijalizacije u principu trebale biti manje efikasne nego direktna inicijalizacija promjenljive "a" (s obzirom da se prvo kreira *privremeni objekat* koji se inicijalizira na vrijednost "(1,2)" koji se zatim *kopira* u objekat "a"), većina kompajlera će ove konstrukcije optimizirati i izbjeći nepotrebno kopiranje, tako da će se one efektivno svesti na konstrukciju u kojoj se promjenljiva "a" inicijalizira direktno (treba istaći da kompajler ima samo *pravo* ali ne i *obavezu* da takve optimizacije zaista sprovede). Ipak, treba znati da su ovakve "indirektne" inicijalizacije dozvoljene samo kod klasa čiji se primjerci mogu kopirati (a takve su sve klase kod kojih dizajner nije eksplicitno rekao drugačije), bez obzira da li će do kopiranja zaista doći ili će ga kompajler izbjeći. U suprotnom, kompajler će odmah prijaviti grešku da kopiranje primjeraka te klase nije dozvoljeno, prije nego što uopće pokuša primijeniti ikakvu optimizaciju!

Kako se primjerci klasa mogu međusobno dodjeljivati (osim ukoliko projektant klase nije odredio drugačije), postavljanje promjenljive "a" na vrijednost "(1,2)" načelno je moguće izvesti i pomoću sljedeće konstrukcije, u kojoj se koristi *naknadna dodjela*:

```
Kompleksni a;
a = Kompleksni(1, 2);
```

Međutim, bez obzira na "inteligenciju" upotrijebljenog kompajlera, ovakva konstrukcija je uvijek osjetno manje efikasna od prethodnih. Naime, ovdje se prvo kreira objekat "a" koji se inicijalizira na kompleknu vrijednost "(0,0)" posredstvom *konstruktora bez parametara* (stoga ovakva konstrukcija ne bi bila moguća da konstruktor bez parametara ne postoji), nakon čega se kreira privremeni bezimeni objekat tipa "Kompleksni" koji se inicijalizira na kompleksni broj "(1,2)" i koji se na kraju kopira u objekat "a" (uništavajući pri tome njegov prethodni sadržaj). Ovim se jasno uočava razlika između *inicijalizacije* i *dodjele*. Razlika u efikasnosti između neposredne inicijalizacije i konstrukcija u kojima se koristi naknadna dodjela osjetno se povećava sa porastom složenosti objekata.

Konstruktor bez parametara se također može pozvati kao funkcija (uz obavezno navođenje para praznih zagrada, kao i kod poziva bilo koje druge funkcije bez parametara). 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, iznenadićemo se kada primijetimo da ova sintaksna konstrukcija radi čak i kod klasa koje *uopće nemaju konstruktore*. Suština je u tome što *konstruktore zapravo ima svaka klasa*, pri čemu ukoliko mi eksplicitno ne napišemo konstruktor za neku klasu, kompajler automatski generira jedan konstruktor bez parametara za tu klasu, čije je tijelo prazno. Takav automatski generirani konstruktor naziva se ***podrazumijevani konstruktor*** (engl. *default constructor*).

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:

```
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 atribute. Bolje bi bilo odmah primijeniti konstruktor sa dva parametra da inicijalizira objekat "c" na željeni sadržaj:

```
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:

```
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. Mada je kod iole inteligentnijih kompajlera efikasnost ovog i prethodnog rješenja praktično ista, vidljivo je da je ovo rješenje jednostavnije i prirodnije, jer se ne koristi pomoćna promjenljiva.

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, uz dodatnu činjenicu da objekti nazvani "privremeni_1" i "privremeni_2" prestaju postojati čim se završi inicijalizacija objekta "c":

```
Kompleksni privremeni_1(3, 12), privremeni_2(5, -1);
Kompleksni c(ZbirKompleksnih(privremeni_1, privremeni_2));
c.Ispisi();
```

Na osnovu onoga što smo vidjeli na prethodnom predavanju 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, nije na odmet još jednom napomenuti da je direktni prenos objekta koji je vraćen kao rezultat iz konstuktora u neku drugu funkciju moguć samo u slučaju da se odgovarajući parametar prenosi *po vrijednosti*, ili *po referenci na konstantni objekat* (kao što jeste u slučaju funkcije "ZbirKompleksnih"), s obzirom da privremeni bezimeni objekti nemaju status l-vrijednosti. Ukoliko bi odgovarajući formalni parametar bio *obična referenca* (na nekonstantni objekat) tada bi odgovarajući stvarni parametar morao biti *promjenljiva* ili neka druga *l-vrijednost* (npr. element niza ili dereferencirani pokazivač).

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 kompajler 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 uspjeti...

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*, kompajler 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 gdje kompajler očekuje da zatekne primjerak neke klase pojavi neki drugi tip podataka (uključujući eventualno i primjerak neke druge klase), kompajler 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 upotrijebljen umjesto onoga čiji tip nije bio odgovarajući. Tek ukoliko takav konstruktor *ne postoji*, biva prijavljena greška. Pojednostavljeno rečeno, možemo reći da se konstruktor neke klase sa jednim parametrom koristi za *automatsku 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 kasnije 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 konstruktorskom 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 konstrukor 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 sljedećih konstrukcija:

```
Kompleksni a(5);
Kompleksni a = 5;
Kompleksni a = Kompleksni(5);
Kompleksni a(Kompleksni(5));
```

Prva konstrukcija direktno inicijalizira objekat "a" pozivom konstruktora sa jednim parametrom. Druga konstrukcija se automatski interpretira kao treća, osim ukoliko bismo konstruktor klase "Kompleksni" sa jednim parametrom deklarirali kao *eksplicitni konstruktor* (tada ova konstrukcija *ne bi bila uopće dozvoljena*). Što se tiče treće i četvrte konstrukcije, ranije smo vidjeli da je njihov smisao praktično identičan (osim kod nekih patološki dizajniranih klasa, na šta smo već ukazali). Principijelno gledano, ove konstrukcije prvo kreiraju privremeni objekat koji se inicijalizira pomoću konstruktora sa jednim parametrom i koji se nakon toga kopira u objekat "a", mada će većina kompajlera optimizirati ove konstrukcije da generiraju posve isti izvršni kôd kao i prva konstrukcija u kojoj se objekat "a" kreira neposredno.

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)". Isto vrijedi i za konverzije tipa pomoću operatora "**static_cast**" poput "**static_cast**<Kompleksni>(x)". Ovo je u skladu sa već uobičajenim tretmanom po kojem su izrazi poput "(**int**)x", "**int**(x)" i "**static_cast**<int>(x)" ekvivalentni.

Sasvim je moguće napraviti *nizove čiji su elementi primjerci neke klase*, potpuno analogno nizovima čiji su elementi strukturnog 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 primjerci klasa (često se ovakvi nizovi pogrešno nazivaju *nizovi klasa*, iako je jasno da se od klasa koje su *tipovi* a ne *objekti* ne mogu praviti nizovi). Nizove čiji su elementi primjerci neke klase moguće je bez problema deklarirati jedino ukoliko ta 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]);
```

Svi elementi tako stvorenog dinamičkog niza također bi bili inicijalizirani konstruktorom bez parametara.

Situacija je znatno složenija ukoliko klasa *ima konstruktore*, *ali nema konstruktor bez parametara*. U tom slučaju, neposredne deklaracije nizova (kako statičkih, tako i dinamičkih) čiji su elementi primjerci te klase *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 klasa*, pri čemu se inicijalizacija elemenata niza vrši kao i statička inicijalizacija ma kojeg drugog niza (navođenjem željenih elemenata niza unutar vitičastih zagrada). Pri tome se moraju navesti *svi elementi niza*. Međutim, kako su u ovom slučaju elementi niza *primjerci klase*, to i elementi u vitičastim zagradama moraju biti također primjerci iste klase (alternativno bi se mogli navesti i elementi koji se pomoću neeksplicitnog konstruktora sa jednim parametrom mogu automatski pretvoriti u primjerke te

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), ili neki izrazi koji su nekog tipa koji se može automatski pretvoriti u tip te klase. 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 pomoću operatora "**new**" (strogo uzevši, biblioteka "memory" sadrži tzv. *alokatore* pomoću kojih se može zaobići operator "**new**" i izvršiti dinamičko kreiranje nizova čak i u ovim slučajevima, ali u pitanju su napredne specijalističke tehnike o kojima nećemo govoriti). 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 osobito dobrim*, s obzirom da ne postoje uvijek smislene podrazumijevane vrijednosti koje bi trebao 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 primjeraka neke klase koristiti *nizove pokazivača na primjerke klase* (preciznije, nizove čiji su elementi pokazivači na dinamički kreirane primjerke klase). Ovo rješenje biće demonstrirano malo kasnije.

Nešto je jednostavnija situacija prilikom kreiranja *vektora čiji su elementi primjerci neke klase*, s obzirom da se prilikom deklaracije vektora može zadati parametar koji služi za inicijalizaciju *svih elemenata vektora*. Tako, vektor od 1000 elemenata čiji su elementi tipa "Datum", možemo deklarirati recimo ovako (prisustvo drugog parametra u deklaraciji je u ovom slučaju *obavezno*):

```
vector<Datum> vektor_datuma(1000, Datum(1, 1, 2000));
```

Naravno, svi elementi ovakvog vektora biće inicijalizirani na 1. 1. 2000.

Statički inicijalizirani niz čiji su elementi primjerci 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 primjeraka neke klase u praksi ne predstavlja osobit problem, s obzirom da se u objektno orijentiranom programiranju kao elementi nizova mnogo češće koriste *pokazivači na primjerke klasa* nego sami *primjerci klasa*. Jedan od razloga za to je što često ne znamo unaprijed na koje vrijednosti treba inicijalizirati elemente klase, tako da pojedine primjerke klasa često kreiramo *dinamički*, i to tek onda kada saznamo čime ih treba inicijalizirati. Pored toga, kasnije ćemo vidjeti da su nizovi (ili vektori) 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. Pretpostavimo, 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 (slične primjedbe vrijede i u slučaju da koristimo vektor čiji su elementi tipa "`Datum`"). Stoga je mnogo bolje deklarirati niz (ili vektor) čiji su elementi *pokazivači na tip* "`Datum`", a same datume kreirati dinamički *onog trenutka kada podaci o njima budu poznati*. Nakon toga ćemo pokazivač na novostvoreni objekat, smjestiti u niz (ili vektor), a samom objektu ćemo *pristupati indirektno preko pokazivača*. Ilustrirajmo ovo na konkretnom primjeru. Deklariraćemo niz pokazivača na objekte tipa "`Datum`", koji ćemo, jednostavnosti radi, 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 klasa 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`**" (sve ovdje izložene primjedbe vrijede i za slučaj ukoliko se umjesto nizova koriste vektori). 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] = {};        // Svi pokazivači su inicijalizirani
cout << "Unesi 10 datuma:\n";      //   na "0"...
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 < 10; i++) {
    niz_datuma[i]->Ispisi();
    cout << endl;
  }
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
for(int i = 0; i < 10; 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 klasa mogu biti (i često jesu) također primjerci 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 usljed njihovog korištenja veoma se lako rješavaju, samo treba objasniti kako). Naime, pošto klasa "Datum" ima konstruktor, niti jedan objekat tipa "Datum" *ne može ostati neinicijaliziran*! Stoga svako stvaranje bilo kojeg objekta tipa "Student" mora 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 zagrade 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" (atribut "ime_i_prezime" inicijaliziramo pozivom funkcije "strcpy", s obzirom da se radi o *klasičnom nizu znakova*, a ne objektu tipa "string", koji bi se mogao inicijalizirati pomoću operatora dodjele "="):

```
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) :
    datum_rodjenja(d, m, g) {
```

```
      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 nije dopušteno 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 konstruktorskom 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 d, int m, int g) :
    datum_rodjenja(d, m, g), 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 prosto 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 primjerci 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 primjerci klase mogli izvršiti njegovu inicijalizaciju na različite vrijednosti. Oni se mogu inicijalizirati na već opisani način kako se inicijaliziraju statički atributi. Međutim, oni se mogu inicijalizirati i *odmah prilikom deklaracije samog atributa*, na isti način kao što se vrši i deklaracija bilo koje konstante, kao na primjer u sljedećoj deklaraciji:

```
class Student {
  static const int broj_studenata = 100;
  char ime_i_prezime[50];
  const int indeks;
  Datum datum_rodjenja;
public:
  Student(const char ime[], int indeks, int d, int m, int g) :
    datum_rodjenja(d, m, g), indeks(indeks) {
      strcpy(ime_i_prezime, ime);
    }
  // Ovdje bi trebalo definirati ostatak interfejsa klase
};
```

Ovo je *jedini izuzetak* u kojem se inicijalizacija nekog atributa smije navesti *odmah prilikom njegove deklaracije*. Ipak, bitno je naglasiti da se njegova inicijalizacija može izvesti *jedino sintaksom u kojoj se koristi znak jednakosti*, a ne pomoću sintakse u kojoj se vrijednost navodi unutar zagrada (ova anomalija uvedena je da se spriječi mogućnost inicijalizacije statičkih konstantnih atributa pomoću konstruktora). Također, ovom sintaksom je moguće inicijalizirati isključivo atribute koji su na fundamentalnom nivou *cjelobrojni* (recimo tipovi poput "**int**" ili "**char**", kao i pobrojani tipovi). Stoga se statički konstantni atributi tipa poput "**double**" ili "string" ne mogu inicijalizirati na ovaj način, nego isključivo na način kao i svi drugi statički atributi (razlog za ovo ograničenje je tehničke prirode i vezan je za mogućnost da necjelobrojni atributi imaju različitu internu reprezentaciju na različitim računarskim arhitekturama).

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, prerušeni pokazivač), kao u sljedećoj deklaraciji:

```
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
};
```

Korištenjem reference umjesto pokazivača 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 (poput inicijalizacije konstantnih atributa). 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 referenca (najbolje konstantna) na neki objekat tipa "`Student`".

Pristup u kojem se umjesto pokazivača na studenta koji je zadužio knjigu koristi referenca ipak posjeduje i jedan nedostatak. Naime, postavlja se pitanje kako modelirati knjigu (tj. objekat tipa "`Knjiga`") *koju niko nije zadužio*. U slučaju da je atribut "`kod_koga_je`" pokazivač, prirodno rješenje u tom slučaju je postaviti taj atribut na nulu (tj. na nul-pokazivač). Međutim, ta mogućnost otpada ukoliko se umjesto pokazivača koriste reference. Jedan od načina da se riješi taj problem je da se uvede *fiktivni student* (tj. objekat tipa "`Student`") za kojeg se pretpostavlja da *posjeduje one i samo one knjige koje niko nije zadužio* (tog studenta možemo nazvati recimo "`niko`"). Tako se knjiga koju nije zadužio niko modelira kao da ju je zadužio "`niko`" (tj. knjiga koja se ne nalazi *ni kod koga* modelira se kao da se nalazi *kod nikoga* – obratite pažnju na malu jezičku zavrzlamu).

U vezi sa problematikom inicijalizacije atributa klase, prirodno je 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 redoslijed izvršavanja tzv. *destruktora*, o kojima 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*).

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 zakomplicirali život programerima, nego upravo suprotno – da olakšaju razvoj programa. Stoga, prihvatite 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...

Predavanje 10.

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). Neka na primjer želimo napraviti klasu nazvanu "`VektorNd`" 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 "`dimenzija`" i "`koordinate`", koji respektivno predstavljaju dimenziju vektora i dinamički alociran niz komponenti vektora (preciznije, pokazivač na prvi element dinamički alociranog niza). Radi jednostavnosti, ovu klasu ćemo napisati samo sa minimalnim interfejsom, koji čine konstruktor bez parametara koji inicijalizira dimenziju vektora na 0 a pokazivač na koordinate na *nul-pokazivač*, zatim metoda "`PostaviDimenziju`" koja vrši dinamičku alokaciju, metoda "`OslobodiMemoriju`" koja oslobađa alocirani prostor, metoda "`Ispisi`" koja u vitičastim zagradama ispisuje kooordinate vektora međusobno razdvojene zarezom, kao i dvije metode nazvane "`Koordinata`" (jedna je konstantna, a druga nije) koje omogućavaju pristup koordinatama vektora:

```
class VektorNd {
  int dimenzija;
  double *koordinate;
public:
  VektorNd() : dimenzija(0), koordinate(0) {}
  void PostaviDimenziju(int n) {
    dimenzija = n; koordinate = new double[n];
  }
  void OslobodiMemoriju() { delete[] koordinate; dimenzija = 0; }
  void Ispisi() const;
  double Koordinata(int n) const;
  double &Koordinata(int n);
};
```

Slijede implementacije metoda "`Ispisi`" i "`Koordinata`" koje nisu izvedene unutar deklaracije klase:

```
void VektorNd::Ispisi() const {
  cout << "{";
  for(int i = 0; i < dimenzija - 1; i++)
    cout << koordinate[i] << ",";
  cout << koordinate[dimenzija - 1] << "}";
}

double VektorNd::Koordinata(int n) const {
  if(n < 1 || n > dimenzija) throw "Pogrešan indeks!\n";
  return koordinate[n - 1];
}

double &VektorNd::Koordinata(int n) {
  if(n < 1 || n > dimenzija) throw "Pogrešan indeks!\n";
  return koordinate[n - 1];
}
```

Implementacije dvije metode nazvane "`Koordinata`" 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 "`Koordinata`" vraća *vrijednost* odgovarajućeg elementa niza "`koordinate`", 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. Naime, 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 spriječeno da se poziv metode "`Koordinata`" nađe sa lijeve strane jednakosti ukoliko je

pozvana nad konstantnim objektom i na taj način omogući promjena niza na koji pokazuje pokazivač "`koordinate`". Slijedi jednostavan primjer koji demonstrira upotrebu napisane klase:

```
VektorNd v1, v2;
try {
  v1.PostaviDimenziju(5); v2.PostaviDimenziju(3);
  v1.Koordinata(1) = 3; v1.Koordinata(2) = 5; v1.Koordinata(3) = -2;
  v1.Koordinata(4) = 0; v1.Koordinata(5) = 1;
  v2.Koordinata(1) = 3; v2.Koordinata(2) = 0; v2.Koordinata(3) = 2;
  v1.Ispisi(); v2.Ispisi();
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
v1.OslobodiMemoriju(); v2.OslobodiMemoriju();
```

Pojasnimo još malo zbog čega smo uveli dvije verzije metode "`Koordinata`". Da smo definirali samo jednu verziju ove metode, koja nije deklarirana kao konstantna, ne bismo mogli pristupati koordinatama konstantnih vektora. Da smo definirali samo konstantnu metodu "`Koordinata`" 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 "`Koordinata`" koja vraća referencu na odgovarajući element niza, mogli bismo promijeniti elemente niza, *čak i ukoliko je objekat tipa* "`VektorNd`" *prethodno deklariran kao konstantan*! Na prvi pogled ovo izgleda nemoguće, s obzirom da je metoda "`Koordinata`" deklarirana kao konstantna. Međutim, metoda "`Koordinata`" zaista *ne mijenja niti jedan od atributa klase* "`VektorNd`"! Ona samo vraća referencu na element dinamički kreiranog niza, a *sama izmjena obavlja se potpuno izvan same metode*. Opisano rješenje sa dvije verzije metode "`Koordinata`" je *zaista jedino* koje omogućava konzistentno ponašanje u svim situacijama!

Alternativno smo umjesto napisanih metoda nazvanih "`Koordinata`" mogli napisati dvije posebne metode "`DajKoordinatu`" i "`PostaviKoordinatu`", od kojih bi metoda "`DajKoordinatu`" bila deklarirana kao konstantna. Tada bismo za čitanje koordinata koristili metodu "`DajKoordinatu`", kao na primjer u konstrukciji "`cout << v1.DajKoordinatu(1)`", dok bismo za postavljanje koordinata umjesto konstrukcija poput "`v1.Koordinata(1) = 3`" pisali "`v1.PostaviKoordinatu(1, 3)`". Mada je takvo rješenje više u duhu do sada razmatranih rješenja, gore prikazano rješenje sa dvije verzije metode "`Koordinata`" je univerzalnije (razlog zbog čega smo ovu metodu nazvali prosto "`Koordinata`" a ne recimo "`DajKoordinatu`" je upravo mogućnost da se ona koristi za obostrani pristup koordinatama vektora, tj. i za čitanje i za upis). U svakom slučaju, ovo je samo privremeno rješenje dok ne naučimo kako da klasu "`VektorNd`" proširimo tako da može direktno podržavati indeksiranje, tj. da možemo direktno pisati konstrukcije poput "`cout << v1[1]`" i "`v1[1] = 3`" sa ciljem pristupa elementima pridruženog dinamičkog niza (uskoro ćemo vidjeti da je i ovo moguće).

Bitno je primijetiti da neće nastati nikakvi problemi zbog pozivanja metode "`OslobodiMemoriju`" 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..

Mada prikazano rješenje radi lijepo, u njemu se javlja jedna nedoslijednost. Primijetimo da se pri deklaraciji objekta tipa "`VektorNd`" on prvo inicijalizira na prazan vektor, a zatim mu dimenzionalnost određuje *naknadno* (pozivom metode "`PostaviDimenziju`"). Mnogo je prirodnije 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 "`PostaviDimenziju`". Ona 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 "`VektorNd`" u kojoj je dinamička alokacija memorije povjerena konstruktoru:

```
class VektorNd {
  int dimenzija;
  double *koordinate;
public:
  explicit VektorNd(int n) : dimenzija(n), koordinate(new double[n]) {}
  void OslobodiMemoriju() { delete[] koordinate; }
  void Ispisi() const;
  double Koordinata(int n) const;
  double &Koordinata(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 koordinata vektora, ali njegovi elementi ostaju nedefinirani (zapravo, ista stvar je vrijedila i za prethodno napisanu metodu "`PostaviDimenziju`"). Međutim, nikakav problem nije prepraviti konstruktor da inicijalizira i koordinate kreiranog vektora na neku željenu inicijalnu vrijednost (recimo, na nulu):

```
explicit VektorNd(int n) : dimenzija(n), koordinate(new double[n]) {
  for(int i = 0; i < n; i++) koordinate[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 {
  VektorNd v1(5), v2(3);
  v1.Koordinata(1) = 3; v1.Koordinata(2) = 5; v1.Koordinata(3) = -2;
  v1.Koordinata(4) = 0; v1.Koordinata(5) = 1;
  v2.Koordinata(1) = 3; v2.Koordinata(2) = 0; v2.Koordinata(3) = 2;
  v1.Ispisi(); v2.Ispisi();
  v1.OslobodiMemoriju(); v2.OslobodiMemoriju();
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
```

Primijetimo da je sintaksa kojom se dimenzioniraju objekti tipa "`VektorNd`" 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* eksplicitno pozvati metodu za oslobađanje memorije (u našem primjeru metodu "`OslobodiMemoriju`") 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 tokom svog života, recimo prilikom poziva konstruktora

ili neke metode koja vrši zauzimanje dodatnih resursa (ti resursi su obično *dinamički alocirana memorija*, ali mogu biti i otvorene datoteke na disku i razne druge stvari). Slično konstruktorima, ni destruktori nemaju povratni tip, ali za razliku od konstruktora oni *ne mogu imati parametre*, a ime im je isto kao ime klase u kojoj se nalaze, samo sa prefiksom "~" (tilda) ispred imena. Slijedi primjer klase "`VektorNd`" u kojoj smo definirali destruktor (čime je otpala potreba za metodom "`OslobodiMemoriju`"):

```
class VektorNd {
  int dimenzija;
  double *koordinate;
public:
  explicit VektorNd(int n) : dimeznija(n), koordinate(new double[n]) {}
  ~VektorNd() { delete[] koordinate; }
  void Ispisi() const;
  double Koordinata(int n) const;
  double &Koordinata(int n);
};
```

Sada upotreba klase "`VektorNd`" postaje mnogo jednostavnija, jer se više ne moramo eksplicitno brinuti o oslobađanju dinamički alocirane memorije koja je zauzeta prilikom stvaranja objekata tipa "`VektorNd`":

```
try {
  VektorNd v1(5), v2(3);
  v1.Koordinata(1) = 3; v1.Koordinata(2) = 5; v1.Koordinata(3) = -2;
  v1.Koordinata(4) = 0; v1.Koordinata(5) = 1;
  v2.Koordinata(1) = 3; v2.Koordinata(2) = 0; v2.Koordinata(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 osloboditi memoriju. Također, lijepa je stvar što se destruktori pozivaju samo nad objektima koji su *zaista stvoreni* (pod stvorenim objektom smataramo 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 destruktor nad objektom "`v1`", koji će ga uništiti. Destruktor 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 oslobađaju nas mnogih problema, a njihova deklaracija je sasvim jednostavna. Zbog toga se može smatrati pravilom da svaka klasa koja vrši dodatno zauzimanje računarskih resursa bilo iz konstruktora bilo iz neke druge metode (tipičan primjer je dinamička alokacija memorije) obavezno mora imati destruktor, koji će osloboditi sve resurse koji su dodatno zauzeti tokom života objekata te klase. Ipak, veoma je bitno naglasiti da zbog jednog ozbiljnog problema uzrokovanog plitkim kopiranjem, svaka klasa koja posjeduje destruktore po pravilu mora posjedovati i tzv. *konstruktor kopije* i *preklopljeni operator dodjele*. O ovome ćemo detaljnije govoriti nešto kasnije, nakon što se još malo upoznamo sa funkcioniranjem destruktora. Nažalost, ovoj *izuzetno važnoj problematici* se u literaturi posvećuje *premalo pažnje* (često i nimalo).

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 ugradi "**`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.

Bitno je napomenuti da *bacanje izuzetaka iz destruktora treba izbjegavati po svaku cijenu* (srećom, ovo je rijetko potrebno). Naime, bacanje izuzetaka iz destruktora može u nekim situacijama izazvati veoma čudne efekte (koji obično završavaju krahom programa), zbog čega je najbolje bacanje izuzetaka iz destruktora potpuno izbjeći. 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 destruktor kao funkciju, nego definirajte *pomoćnu funkciju* (tipično u privatnoj sekciji klase) unutar koje ćete smjestiti te naredbe, a zatim tu pomoćnu funkciju pozovite iz destruktora, i sa bilo kojeg drugog mjesta 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 konstruktore i destruktore. Na primjer, ukoliko smo *dinamički* stvorili neki objekat tipa "VektorNd" naredbom poput

```
VektorN *pok(new VektorN(5));
```

tada će njegovo brisanje pomoću naredbe

```
delete pok;
```

izazvati dva efekta: prvo će nad dinamički stvorenim objektom "*pok" biti pozvan destruktor (koji će osloboditi memoriju koja je zauzeta u konstruktoru objekta), pa će tek tada biti oslobođena memorija koji je sam objekat "*pok" zauzimao (tj. prostor za njegove atribute "dimenzija" i "koordinate"). 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 isključivo za ono za šta je namijenjen!

Ostali smo dužni da objasnimo zbog čega smo konstruktor klase "VektorNd" definirali kao eksplicitni 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 = VektorNd(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 nešto kasnije). Slična neočekivana situacija nastala bi ukoliko bi nekoj funkciji koja očekuje objekat tipa "VektorN" kao parametar bio proslijeđen broj (taj broj bi bio proslijeđen konstruktoru sa jednim parametrom, nakon čega bi konstruisani objekat bio proslijeđen funkciji, a to sigurno nije željeno ponašanje). Ovako, označavanjem konstruktora sa "**explicit**", zabranjuje se njegovo korištenje za automatsku pretvorbu tipova iz cjelobrojnog u tip "VektorNd", tako da ovakve besmislene konstrukcije neće biti ni dozvoljene (tj. dovešće do prijave greške od strane kompajlera).

Većinu stvari o kojima smo do sada govorili ilustrira sljedeći program, koji obavlja istu funkciju kao program za obradu rezultata učenika u razredu koji smo prikazali na nekom od ranijih predavanja, samo što je napisan u duhu objektno zasnovanog programiranja i bez korištenja specijalnih tipova podataka

definiranih u standardnim bibliotekama kao što su "vector" itd. 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;
class Datum {
  int dan, mjesec, godina;
public:
  Datum(int d, int m, int g);
  void Ispisi() const {
    cout << dan << "." << mjesec << "." << godina;
  }
};
class Ucenik {
  static const int BrojPredmeta = 12;      // Pri testiranju smanjite
  char ime[20], prezime[20];               //   broj predmeta...
  Datum datum_rodjenja;
  int ocjene[BrojPredmeta];
  double prosjek;
  bool prosao;
public:
  Ucenik(const char ime[], const char prezime[], int d, int m, int g);
  void PostaviOcjenu(int predmet, int ocjena);
  static int DajBrojPredmeta() { return BrojPredmeta; }
  double DajProsjek() const { return prosjek; }
  bool DaLiJeProsao() const { return prosao; }
  void Ispisi() const;
};
class Razred {
  const int kapacitet;
  int broj_evidentiranih;
  Ucenik **ucenici;
  static bool BoljiProsjek(const Ucenik *u1, const Ucenik *u2) {
    return u1->DajProsjek() > u2->DajProsjek();
  }
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() {
  try {
    int broj_ucenika;
    cout << "Koliko ima ucenika: ";
    cin >> broj_ucenika;
    if(!cin) throw 0;                  // Ovdje je nebitno šta bacamo...
    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";
  }
  return 0;
}
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!";
  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), prosao(false) {
    if(strlen(ime) > 19 || strlen(prezime) > 19)
      throw "Predugacko ime ili prezime!";
    strcpy(Ucenik::ime, ime); strcpy(Ucenik::prezime, prezime);
    for(int i = 0; i < BrojPredmeta; i++) ocjene[i] = 1;
}
void Ucenik::PostaviOcjenu(int predmet, int ocjena) {
  if(ocjena < 1 || ocjena > 5) throw "Pogresna ocjena!";
  if(predmet < 1 || predmet > BrojPredmeta)
    throw "Pogresan broj predmeta!";
  ocjene[predmet - 1] = ocjena;
  prosjek = 1; prosao = 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; prosao = true;
}
void Ucenik::Ispisi() const {
  cout << "Ucenik " << ime << " " << prezime << " rodjen ";
  datum_rodjenja.Ispisi();
  if(DaLiJeProsao())
    cout << " ima prosjek " << setprecision(3) << DajProsjek() << endl;
  else
    cout << " mora ponavljati razred\n";
}
Razred::~Razred() {
  for(int i = 0; i < broj_evidentiranih; i++) delete ucenici[i];
  delete[] ucenici;
}
void Razred::EvidentirajUcenika(Ucenik *ucenik) {
  if(broj_evidentiranih >= kapacitet) throw "Previse ucenika!";
  ucenici[broj_evidentiranih++] = ucenik;
}
void Razred::UnesiNovogUcenika() {
  bool pogresan_unos(true);
  while(pogresan_unos) {
    try {
      char ime[20], prezime[20];
      int d, m, g;
      char znak1, znak2;
      cout << "  Ime: "; cin >> setw(20) >> ime;
      cout << "  Prezime: "; cin >> setw(20) >> prezime;
      cout << "  Datum rodjenja (D/M/G): ";
      cin >> d >> znak1 >> m >> znak2 >> g;
```

```
        if(!cin || znak1 != '/' || znak2 != '/') throw "Pogresan datum!";
        Ucenik *ucenik(new Ucenik(ime, prezime, d, m, g));
        for(int pr = 1; pr <= Ucenik::DajBrojPredmeta(); pr++) {
          int ocjena;
          cout << "  Ocjena iz " << pr << ". predmeta: ";
          cin >> ocjena;
          if(!cin) throw "Pogresna ocjena!";
          ucenik->PostaviOcjenu(pr, ocjena);
        }
        EvidentirajUcenika(ucenik);
        pogresan_unos = false;
      }
      catch(const char greska[]) {
        cout << "Greska: " << greska << "\nMolimo, ponovite unos!\n";
        cin.clear(); cin.ignore(10000, '\n');
      }
    }
  }
  void Razred::IspisiIzvjestaj() const {
    cout << endl;
    for(int i = 0; i < broj_evidentiranih; i++)
      ucenici[i]->Ispisi();
  }
```

Listing ovog programa može se naći na web stranici kursa pod imenom "ucenici_obp.cpp". U njemu 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 konstantni statički atribut "BrojPredmeta" kao i obične atribute "ime", "prezime", "datum_rodjenja", "ocjene", "prosjek" i "prosao", pri čemu atribut "ocjene" predstavlja klasični niz čiji je broj elemenata određen konstantnim statičkim atributom "BrojPredmeta". U interfejsu klase "Ucenik" nalazi se konstruktor koji inicijalizira atribute "ime", "prezime" i "datum_rodjenja" u skladu sa proslijeđenim parametrima, zatim sve ocjene inicijalizira na 1 (tako da se smatra da učenik nije zadovoljio predmet sve dok ne dobije pozitivnu ocjenu iz njega), dok atribute "prosjek" i "prosao" inicijalizira respektivno na vrijednosti "1" i "**false**" (atribut "prosao" sadrži vrijednost "**true**" ukoliko je učenik prošao a "**false**" ako nije). Dalje su deklarirane i implementirane trivijalne metode "DajProsjek" i "DaLiJeProsao" koje vraćaju respektivno vrijednosti atributa "prosjek" i "prosao", zatim statička metoda "DajBrojPredmeta" koja vraća vrijednost statičkog atributa "BrojPredmeta" (ova metoda je statička jer je nebitno nad kojim se konkretno primjerkom klase "Ucenik" poziva), kao i metoda "Ispis" koja ispisuje podatke o učeniku. Pored toga, imamo i metodu "PostaviOcjenu" 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 "prosjek" i "prosao" u skladu sa novonastalom situacijom.

Interesantno je napomenuti da su se atributi "prosjek" i "prosao" mogli izostaviti, pri čemu bi se tada metode "DajProsjek" i "DaLiJeProsao" trebale izmijeniti tako da prilikom poziva *računaju* prosjek i indikator prolaznosti, umjesto da prosto vrate vrijednosti odgovarajućih atributa. Naravno, metoda "PostaviOcjenu" tada ne bi trebala računati prosjek 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, ovakva implementacija je nešto efikasnija, jer se prosjek i indikator prolaznosti računaju samo prilikom upisa nove ocjene, a uz izmijenjenu implementaciju, oni bi se iznova računali pri svakom pozivu metode "DajProsjek" odnosno "DaLiJeProsao". Ovo poboljšanje naročito dolazi do izražaja prilikom sortiranja spiska učenika, u kojem se veoma često poziva metoda "DajProsjek".

Klasa "Razred" je nešto složenija, i nju ćemo razmotriti malo detaljnije. Njeni atributi 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 čiji su elementi primjerci neke klase i konstruktorima. Konstruktor klase "Razred" postavlja atribut "kapacitet" na vrijednost zadanu parametrom, atribut "broj_evidentiranih" postavlja na nulu, te obavlja 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 popunjen. 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 "PostaviOcjenu") i, konačno, evidentira učenika (tj. upisuje pokazivač na njega u dinamički kreirani niz pokazivača na učenike) 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. U slučaju da ulazni tok dospije u neispravno stanje zbog besmislenog unosa vršimo ručno bacanje izuzetka, tako da sve tipove neispravnog unosa obrađujemo na jednoobrazan način. Metoda "IspisiIzvjestaj" ispisuje izvještaj o svim upisanim učenicima u razredu prostim pozivom metode "Ispisi" nad svakim od upisanih učenika. Primijetimo da se metode "PostaviOcjenu" 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 proslijeđena 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 čiji je formalni parametar funkcija (odnosno 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. Naravno, kao alternativu, funkciju kriterija smo mogli definirati i kao običnu funkciju, ali je ovakvo rješenje bolje, s obzirom da se ona 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 destruktoru 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 (pri čemu se baca izuzetak ukoliko unesemo besmislice), 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 mjesta ukoliko neka od dinamičkih alokacija memorije ne uspije.

Treba naglasiti da bez obzira što je ovako napisan program skoro dvostruko duži od sličnog programa koji je koristio samo strukture i bio napisan u čisto proceduralnom duhu, uloženi trud se, dugoročno gledano, svakako *isplati*, 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.

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 *preklopljenog operatora dodjele* (engl. *overloaded assignment operator*). Konstruktor kopije i preklopljeni operator dodjele su toliko važni za ispravno funkcioniranje klase da se kao pravilo može uzeti da *svaka klasa koja posjeduje destruktor*, *obavezno mora posjedovati i konstruktor kopije i preklopljeni operator dodjele*. Stoga iznenađuje da mnoge knjige o jeziku C++ konstruktore kopije spominju više uzgredno ili čak i nikako. Preklopljenom operatoru 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 da omogući upravljanje 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 istog tipa, kada se vrši *dodjeljivanje* nekog objekta drugom objektu istog tipa, 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 atribute objekta B. Međutim, kada god objekti sadrže atribute koji su *pokazivači na dinamički alocirane resurse*, 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 destruktorima, plitke kopije mogu biti fatalne. Na primjer, 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 destruktor, 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 ranije napisanu klasu "`VektorNd`" proširiti funkcijom "`ZbirVektora`" koja vraća kao rezultat zbir dva *n*-dimenzionalna vektora koji su joj proslijeđeni 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 VektorNd {
  int dimenzija;
  double *koordinate;
public:
  explicit VektorNd(int n) : dimenzija(n), koordinate(new double[n]) {}
  ~VektorNd() { delete[] koordinate; }
  void Ispisi() const;
  double Koordinata(int n) const;
  double &Koordinata(int n);
  friend VektorNd ZbirVektora(VektorNd v1, VektorNd v2);
};
```

Implementacije metoda "`Ispisi`" i "`Koordinata`" su iste kao i ranije, 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):

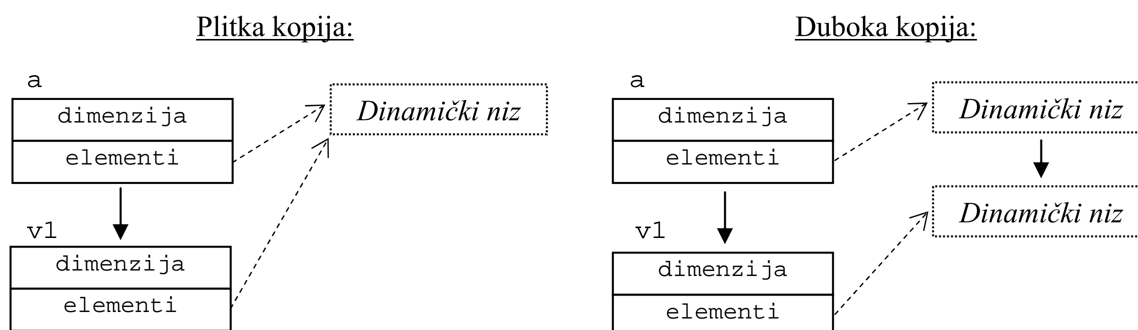
```
VektorNd ZbirVektora(VektorNd v1, VektorNd v2) {
  if(v1.dimenzija != v2.dimenzija)
    throw "Vektori koji se sabiraju moraju biti iste dimenzije!\n";
  VektorNd v3(v1.dimenzija);
  for(int i = 0; i < v1.dimenzija; i++)
    v3.koordinate[i] = v1.koordinate[i] + v2.koordinate[i];
  return v3;
}
```

Na prvi pogled je sve u redu, tako da je, uz pretpostavku da su "`a`" i "`b`" dva vektora iste dimenzije, moguće napisati naredbu poput

```
VektorNd 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 "`VektorNd`" nad svakim od ova tri objekta, koji će osloboditi memoriju koju su zauzimali dinamički alocirani nizovi 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 izbjeći ukoliko bismo parametre prenosili po referenci na konstantne objekte (tj. ako bismo deklarirali da su "`v1`" i "`v2`" reference na konstantne objekte tipa "`VektorNd`"). Kao što znamo, ovo je svakako i preporučeni način prenosa primjeraka klasa u funkcije. Međutim, kopiranje rezultata koji se nalazi u vektoru "`v3`" nije moguće izbjeći čak ni vraćanjem reference na njega kao rezultata (ne smijemo vratiti referencu na objekat koji prestaje postojati). 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 destruktor kopiranog objekta uništiti "svoj" a ne i "tuđi" dinamički niz. Sljedeća slika ilustrira razliku između plitke i potpune (duboke) kopije:



Neko se može zapitati zbog čega jezik C++ prilikom kopiranja objekata automatski ne vrši duboku nego plitku kopiju. Odgovor je veoma jednostavan: kompajler *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 (kompajler može znati samo *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, kompajler automatski generira ***podrazumijevani konstruktor kopije*** (engl. *default copy constructor*) 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 "`VektorNd`" 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 "`VektorNd`"). Nova deklaracija klase "`VektorNd`" izgledaće ovako:

```
class VektorNd {
  int dimenzija;
  double *koordinate;
public:
  explicit VektorNd(int n) : dimenzija(n), koordinate(new double[n]) {}
  VektorNd(const VektorNd &v);
  ~VektorNd() { delete[] koordinate; }
  void Ispisi() const;
  double Koordinata(int n) const;
  double &Koordinata(int n);
  friend VektorNd ZbirVektora(VektorNd v1, VektorNd v2);
};
```

S obzirom da konstruktor kopije nije posve kratak, njegovu implementaciju ćemo izvesti izvan deklaracije klase. Razmotrimo šta on zapravo treba da obavi. Atribut "`dimenzija`" svakako treba kopirati, ali atribut-pokazivač "`koordinate`" 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 "`VektorNd`" mogla izgledati ovako:

```
VektorNd::VektorNd(const VektorNd &v) {
  dimenzija = v.dimenzija; koordinate = new double[v.dimenzija];
  for(int i = 0; i < dimenzija; i++) koordinate[i] = v.koordinate[i];
}
```

Radi bolje efikasnosti, bolje je sve atribute koji se mogu inicijalizirati u konstruktorskoj inicijalizacijskoj listi inicijalizirati unutar konstruktorske inicijalizacijske liste. Također, umjesto

kopiranja element po element, možemo koristiti i funkciju "copy" iz biblioteke "algorithm", što može biti efikasnije. Tako dolazimo do sljedeće optimizirane implementacije konstruktora kopije:

```
VektorNd::VektorNd(const VektorNd &v) : dimenzija(v.dimenzija),
  koordinate(new double[v.dimenzija]) {
    copy(v.koordinate, v.koordinate + v.dimenzija, koordinate);
}
```

Treba napomenuti da su konstruktori kopije "nužno zlo" i da kompajleri imaju pravo (ali ne i obavezu) da izbjegnu njihovo pozivanje kada je god to moguće. Na primjer, ukoliko kompajler primijeti da se objekat A kopira u objekat B, koji se zatim kopira u objekat C i koji se na kraju kopira u objekat D pri čemu se nakon toga objekti B i C ne koriste nizašta (recimo, prestaju postojati), kompajler ima pravo direktno kopirati objekat A u objekat D uz samo jedan poziv konstruktora kopije (umjesto tri). Dalje, ukoliko kompajler primijeti da se konstruira objekat A koji se nakon toga kopira u objekat B a zatim u objekat C pri čemu se nakon toga objekti A i B ne koriste nizašta, kompajler ima pravo da u potpunosti izbjegne kopiranja (i pozive konstruktora kopije) i da direktno konstruira objekat C na isti način kako bi konstruirao objekat A. Stoga konstruktori kopije nikada ne bi trebali obavljati ništa što izlazi iz okvira postupaka za kopiranje objekata, jer nikada nismo sigurni koliko će se puta i kada konstruktori kopije zaista pozvati (s obzirom da kompajler ima pravo izbjeći njihov poziv kad god je to moguće). Posebno, logika programa ne bi smjela ovisiti od toga koliko će se puta konstruktor kopije pozvati!

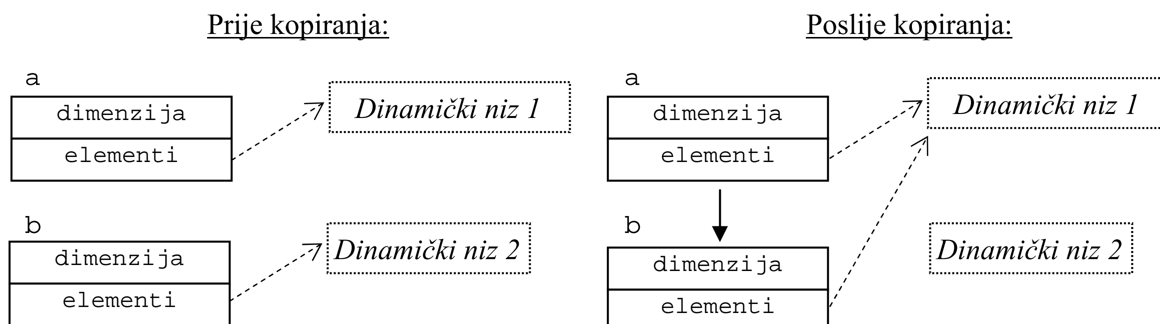
Prethodno definirani konstruktori kopije obezbjeđuju duboko kopiranje, čime ćemo biti sigurni da će svi različiti objekti imati svoje neovisne primjerke dinamičkih nizova, odnosno da se neće "jedan drugom petljati u posao". Ipak, konstruktor kopije se eventualno poziva samo u tri od četiri situacije u kojima bi mogle nastati plitke kopije. Naime, konstruktor kopije se eventualno poziva pri *inicijalizaciji* novostvorenog objekta drugim objektom istog tipa, pri *prenosu po vrijednosti* objekata u funkcije i pri *vraćanju objekata kao rezultata iz* funkcije. Prilikom *dodjeljivanja* nekog objekta drugom objektu koji od ranije postoji, *konstruktor kopije se nikada ne poziva*! To znači da ukoliko bismo izvršili naredbe poput

```
b = a;
c = ZbirVektora(a, b);
```

pri čemu sva tri vektora (tj. objekta tipa "VektorNd") "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). Pretpostavimo, na primjer, da su vektori "a" i "b" prethodno deklarirani deklaracijom

```
VektorNd 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:



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. Na ovom mjestu postaje posve jasno zbog čega je potrebno praviti oštru razliku između *inicijalizacije* i *dodjele* (na ovu razliku ćemo mnogo lakše misliti ukoliko se od samog početka naviknemo da za inicijalizaciju nikada ne koristimo sintaksu koja koristi znak dodjele "=").

Opisani problem bi se mogao riješiti kada bi se prilikom dodjele poput "`b = a`" prvo izvršio *destruktor* 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. Stoga, 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 atribute objekta "`b`", jedan po jedan (isto kao i kod podrazumijevanog konstruktora kopije). Međutim, projektant klase može definirati drugačije ponašanje *preklapanjem operatora dodjele*. Mada je preklapanje operatora dodjele specijalan slučaj općeg preklapanja operatora o kojem ćemo govoriti kasnije, očekivano ponašanje operatora dodjele je veoma tijesno 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 kasnije 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 "`VektorNd`" treba imati sljedeći prototip:

```
VektorNd &operator =(const VektorNd &v);
```

Nakon deklaracije funkcije članice koja realizira preklopljeni operator dodjele, sve dodjele poput "`a = b`" interpretiraće se kao izraz oblika "`a.`**`operator`** `=(b)`". Sada bi kompletna deklaracija klase "`VektorNd`" mogla izgledati ovako:

```
class VektorNd {
  int dimenzija;
  double *koordinate;
public:
  explicit VektorNd(int n) : dimenzija(n), koordinate(new double[n]) {}
  VektorNd(const VektorNd &v);
  VektorNd &operator =(const VektorNd &v);
  ~VektorNd() { delete[] koordinate; }
  void Ispisi() const;
  double Koordinata(int n) const;
  double &Koordinata(int n);
  friend VektorNd ZbirVektora(VektorNd v1, VektorNd 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 (čak ni ukoliko se inicijalizacija vrši pomoću sintakse koja koristi znak dodjele "="), 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:

```
VektorNd &VektorNd::operator =(const VektorNd &v) {
  if(dimenzija < v.dimenzija) {
    delete[] koordinate;
    koordinate = new double[v.dimenzija];
  }
  dimenzija = v.dimenzija;
  copy(v.koordinate, v.koordinate + v.dimenzija, koordinate);
  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.

Mnogi 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. Ovim se oponaša efekat kao da je nad objektom sa lijeve strane operatora dodjele izvršen destruktor, a zatim izvršen konstruktor kopije. Međutim, na taj način se bespotrebno vrši dealokacija i ponovna alokacija memorije čak i u slučajevima kad je ona potpuno nepotrebna (recimo, realokacija je bez ikakve sumnje posve neopravdana 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):

```
VektorNd &VektorNd::operator =(const VektorNd &v) {
  if(&v == this) return *this;
  delete[] koordinate;
  dimenzija = v.dimenzija; koordinate = new double[dimenzija];
  copy(v.koordinate, v.koordinate + v.dimenzija, koordinate);
  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 izbjeći (takav slučaj smo imali kod ranije napisane klase "`Razred`"). Preklopljeni operator dodjele također bismo mogli izbjeći 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). Deklariranje je potrebno izvesti u privatnom a ne u javnom dijelu klase, jer će tada kompajler prijaviti grešku već pri prvom pokušaju kopiranja objekta. Kada bi deklaracija bila izvedena u javnom dijelu klase, kompajler ne bi prijavio grešku pri pokušaju kopiranja objekta, već tek na *samom kraju prevođenja* (u fazi povezivanja), kada ustanovi da konstruktor kopije nije nigdje implementiran, a iz tako prijavljene greške ne bismo mogli znati gdje je zapravo nastupio problem. Stoga bi, iz razloga sigurnosti, ranije napisanu klasu "`Razred`" bilo bolje 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, on 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. Kao zaključak, ukoliko neka klasa posjeduje

destruktor, 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: *neznanje* 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.

Veoma je interesantno razmotriti kako bi se trebao implementirati konstruktor kopije ranije napisane klase "Razred" u slučaju da se odlučimo da ga implementiramo. Da bismo odgovorili na ovo pitanje, razmotrimo kakva je *logička struktura* ove klase, s obzirom da je cilj konstruktora kopije da napravi tačnu *logičku kopiju* objekta, a ne puku kopiju njenih atributa, kao i kakva je veza te logičke strukture sa atributima klase (tj. njenom *fizičkom strukturom*). Logički, objekat klase "Razred" predstavlja kolekciju podataka o učenicima, ali fizički gledano, ti podaci se ne sadrže u samom objektu, nego u posebnim dinamički alociranim objektima. Tim objektima se pristupa preko niza pokazivača koji na njih pokazuju, ali ni taj niz pokazivača nije sadržan u samom objektu klase "Razred", nego mu se pristupa preko drugog pokazivača (koji *jeste* sadržan u objektu klase "Razred"). Sad ako malo razmislimo, vidjećemo da trebamo napraviti kopije svih objekata koji sadrže podatke o učenicima, ali da ne trebamo kopirati niz pokazivača na njih, s obzirom da oni svakako trebaju pokazivati na novostvorene kopije, a ne na izvorne podatke o učenicima. Jedan (ne osobito dobar) način da se to izvede je sljedeći:

```
Razred::Razred(const Razred &r) : ucenici(new Ucenik*[r.kapacitet]),
  kapacitet(r.kapacitet), broj_evidentiranih(r.broj_evidentiranih) {
    for(int i = 0; i < broj_evidentiranih; i++) {
      ucenici[i] = new Ucenik("", "", 1, 1, 2000);
      *ucenici[i] = *r.ucenici[i];
    }
}
```

U ovoj izvedbi, prvo se unutar "**for**" petlje alocira memorija za kopiju podataka za svakog od učenika. Pri tome se konstruktoru klase "Ucenik" predaju neki proizvoljni, ali legalni parametri (tj. koji neće dovesti do bacanja izuzetka). Naime, klasa "Ucenik" je izvedena tako da svako kreiranje nekog primjerka te klase mora biti praćeno pozivom konstruktora sa 5 parametara, što ne možemo ni u kom slučaju izbjeći. Nakon toga se pomoću konstrukcije "*ucenici[i] = *r.ucenici[i]" vrši stvarno kopiranje aktuelnih podataka o učeniku. Ovaj pristup ima dva nedostatka. Prvo, svako kreiranje novog primjerka klase "Ucenik" praćeno je nepotrebnom inicijalizacijom na besmislene podatke, koji će svakako već u sljedećoj naredbi biti prebrisani aktuelnim podacima. Drugo, konstrukcija poput "*ucenici[i] = *r.ucenici[i]" uopće ne bi bila dozvoljena da je klasa "Ucenik" bila napravljena tako da je međusobna dodjela primjeraka te klase zabranjena. Oba nedostatka mogu se izbjeći ukoliko na neki način postignemo da se prilikom dinamičkog kreiranja nekog objekta tipa "Ucenik" on odmah inicijalizira sadržajem nekog drugog objekta čiju kopiju on treba da predstavlja. Ovo se može učiniti tako da prilikom dinamičke alokacije objekta *eksplicitno pozovemo upravo konstruktor kopije klase* "Ucenik", navodeći mu kao parametar *objekat čija se kopija kreira*. Mada takva konstrukcija može izgledati neobično, ona je posve legalna. Naime, konstruktor kopije je konstruktor kao i svaki drugi, jedino je specifičan po tome kakav parametar prima, i po tome što se automatski poziva u svim slučajevima kada je potrebno (automatsko) kreiranje kopije nekog objekta. Inače, sve što vrijedi za ma koji drugi konstruktor, vrijedi i za konstruktor kopije. Na osnovu ovih razmatranja, slijedi da bi dobro napisan konstruktor kopije klase "Razred" trebao izgledati ovako:

```
Razred::Razred(const Razred &r) : ucenici(new Ucenik*[r.kapacitet]),
  kapacitet(r.kapacitet), broj_evidentiranih(r.broj_evidentiranih) {
    for(int i = 0; i < broj_evidentiranih; i++)
      ucenici[i] = new Ucenik(*r.ucenici[i]);
}
```

Kao nešto složeniju ilustraciju većine 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 (koji se može naći na web stranici kursa pod imenom "matrica_obp.cpp") slijedi odmah nakon njegovog prikaza:

```
#include <iostream>
#include <iomanip>
#include <cstring>

using namespace std;

template <typename TipEl>
class Matrica {
  int br_redova, br_kolona;
  TipEl **elementi;
  char ime_matrice;
  void AlocirajMemoriju(int br_redova, int br_kolona);
  void DealocirajMemoriju();
public:
  Matrica(int br_redova, int br_kolona, char ime = 0);
  Matrica(const Matrica &m);
  ~Matrica() { DealocirajMemoriju(); }
  Matrica &operator =(const Matrica &m);
  void Unesi();
  void Ispisi(int sirina_ispisa) const;
  template <typename Tip2>
  friend Matrica<Tip2> ZbirMatrica(const Matrica<Tip2> &m1,
    const Matrica<Tip2> &m2);
};

template <typename TipEl>
void Matrica<TipEl>::AlocirajMemoriju(int br_redova, int br_kolona) {
  elementi = new TipEl*[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 TipEl[br_kolona];
  }
  catch(...) {
    DealocirajMemoriju();
    throw;
  }
}

template <typename TipEl>
void Matrica<TipEl>::DealocirajMemoriju() {
  for(int i = 0; i < br_redova; i++) delete[] elementi[i];
  delete[] elementi;
}

template <typename TipEl>
Matrica<TipEl>::Matrica(int br_redova, int br_kolona, char ime) :
  br_redova(br_redova), br_kolona(br_kolona), ime_matrice(ime) {
    AlocirajMemoriju(br_redova, br_kolona);
}

template <typename TipEl>
Matrica<TipEl>::Matrica(const Matrica<TipEl> &m) :
  br_redova(m.br_redova), br_kolona(m.br_kolona),
  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 TipEl>
Matrica<TipEl> &Matrica<TipEl>::operator =(const Matrica<TipEl> &m) {
  if(br_redova < m.br_redova || br_kolona < m.br_kolona) {
    DealocirajMemoriju(); AlocirajMemoriju(m.br_redova, m.br_kolona);
  }
  else if(br_redova > m.br_redova)
    for(int i = m.br_redova; i < br_redova; i++) delete elementi[i];
```

```
    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 TipEl>
  void Matrica<TipEl>::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 TipEl>
  void Matrica<TipEl>::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 TipEl>
  Matrica<TipEl> ZbirMatrica(const Matrica<TipEl> &m1,
    const Matrica<TipEl> &m2) {
      if(m1.br_redova != m2.br_redova || m1.br_kolona != m2.br_kolona)
        throw "Matrice nemaju jednake dimenzije!\n";
      Matrica<TipEl> 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";
    }
    return 0;
  }
```

Mada do sada nije eksplicitno rečeno da klase također mogu biti generičke, to nije neočekivano 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 konkretan 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. Svaka od njih je uklopljena u šablon, tako da po formi sve implementacije metoda podsjećaju na implementacije običnih generičkih funkcija.

Pogledajmo sada implementaciju ove klase. Vidimo da pored konstruktora, destruktora i preklopljenog operatora dodjele, interfejs ove klase sadrži metode "Unesi" i "Ispisi", koje su dovoljno jednostavne, tako da se o njima nema mnogo toga reći (metoda "Ispisi" zahtijeva kao parametar željenu širinu ispisa koji će zauzeti svaki element matrice). Definirana je i prijateljska generička 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". Obratimo pažnju kako je uspostavljeno "prijateljstvo" između generičke klase "Matrica" i generičke funkcije "ZbirMatrica". Unutar deklaracije klase moramo naglasiti da je "ZbirMatrica" generička funkcija, pri čemu moramo upotrijebiti neko drugo ime parametra šablona ("Tip2" u našem primjeru) da bismo izbjegli konflikt sa imenom parametra šablona koji definira klasu "Matrica". Da nismo unutar deklaracije klase naglasili da je "ZbirMatrica" generička funkcija, tada bi prijateljstvo bilo uspostavljeno između svake od konkretnih specifikacija generičke klase "Matrica" i odgovarajuće *obične* (negeneričke) funkcije "ZbirMatrica" čiji parametri po tipu odgovaraju konkretnoj specifikaciji šablona (na primjer, između klase "Matrica<**double**>" i obične funkcije "ZbirMatrica" čiji su parametri i povratna vrijednost tipa "Matrica<**double**>"), što vjerovatno nije ono što želimo.

Konstruktor klase "Matrica" ima tri parametra: dimenzije matrice i znak koji se koristi pri ispisu prilikom unosa elemenata matrice (ovaj parametar je opcionalan, a u slučaju izostavljanja podrazumjieva se nul-karakter). 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". Međutim, kako je sama dinamička alokacija dvodimenzionalnih nizova nešto složeniji postupak, 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 na predavanjima na kojima smo govorili 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 destruktor. 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" povjereno je funkciji "DealocirajMemoriju", koja se poziva iz konstruktora (u slučaju greške), iz destruktora (destruktor 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 relizira 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). Treba primijetiti da je u slučaju kada je odredišna matrica prije dodjele imala veći broj redova nego izvorna matrica potrebno osloboditi memoriju koju su zauzimali dodatni redovi. Ukoliko to ne učinimo, imaćemo curenje memorije. Zaista, nakon obavljene dodjele, odredišna matrica gubi informaciju o tome koliko je redova imala prije dodjele. Stoga kada dođe vrijeme da se odredišna matrica briše, destruktor neće imati nikakvu informaciju o tome da su prije postojali dodatni redovi, tako da ih neće moći ni obrisati (drugim riječima, ukoliko prethodno ne obrišemo dodatne redove, kasnije to neće učiniti niko drugi). Ostaje još glavni program (odnosno funkcija "main") koji je dovoljno jednostavan da ne zahtijeva nikakva posebna objašnjenja.

Definiranje konstruktora kopije i preklopljenog operatora 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 usljed 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 destruktor, onog trenutka kada više nije potrebna. Kao ilustraciju ove tehnike, izmijenimo deklaraciju prethodno razvijene klase "VektorNd" na sljedeći način:

```
class VektorNd {
  int dimenzija;
  double *koordinate;
  int *pnb;
public:
  explicit VektorNd(int n) : dimenzija(n), koordinate(new double[n]),
    pnb(new int(1)) {}
  VektorNd(const VektorNd &v) : dimenzija(v.dimenzija),
    koordinate(v.koordinate), pnb(v.pnb) { (*pnb)++; }
  VektorNd &operator =(const VektorNd &v);
  ~VektorNd();
  void Ispisi() const;
  double Koordinata(int n) const;
  double &Koordinata(int n);
  friend VektorNd ZbirVektora(VektorNd v1, VektorNd 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:

```
VektorNd::~VektorNd() {
  if(--(*pnb) == 0) {
    delete[] koordinate; delete pnb;
  }
}
```

Ostaje još implementacija preklopljenog 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), izbjeći ćemo i probleme usljed eventualne samododjele, tako da taj slučaj ne moramo posebno razmatrati:

```
VektorNd &VektorNd::operator =(const VektorNd &v) {
  (*v.pnb)++;
  if(--(*pnb) == 0) delete[] koordinate;
  dimenzija = v.dimenzija; koordinate = v.koordinate; 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 "`VektorNd`"), nove implementacije konstruktora, konstruktora kopije, destruktora i preklopljenog operatora dodjele mogle bi izgledati recimo ovako:

```
template <typename TipEl>
Matrica<TipEl>::Matrica(int br_redova, int br_kolona, char ime) :
  br_redova(br_redova), br_kolona(br_kolona), ime_matrice(ime),
  pnb(new int(1)) {
    AlocirajMemoriju(br_redova, br_kolona);
}

template <typename TipEl>
Matrica<TipEl>::Matrica(const Matrica<TipEl> &m) :
  br_redova(m.br_redova), br_kolona(m.br_kolona), elementi(m.elementi),
  pnb(m.pnb), ime_matrice(m.ime_matrice) {
    (*pnb)++;
}

template <typename TipEl>
Matrica<TipEl>::~Matrica() {
  if(--(*pnb) == 0) {
    DealocirajMemoriju();
    delete pnb;
  }
}

template <typename TipEl>
Matrica<TipEl> &Matrica<TipEl>::operator =(const Matrica<TipEl> &m) {
  (*m.pnb)++;
  if(--(*pnb) == 0) DealocirajMemoriju();
  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 modificiraju pripadni dinamički niz. S druge strane, efikasna realizacija ove tehnike može postati veoma komplicirana u slučajevima kada postoje metode koje se, zavisno od situacije, mogu koristiti kako za čitanje, tako i za modificiranje elemenata dinamičkog niza (poput metode "`Koordinata`" iz klase "`VektorNd`"). Stoga, u detalje ove tehnike nećemo ulaziti. Cilj je bio samo da se ukaže na osnovne ideje 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 podxataka "`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 preklopljene operatore dodjele ("`vector`" je, zapravo, generička klasa). 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 preklopljenim 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 zakačena na objekat. Jedini način kojim se korisnik može uvjeriti u ovu činjenicu je primjena operatora "**`sizeof`**" kojeg je nemoguće "prevariti". Naime, ovaj operator 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 šta je eventualno "prikačeno" 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.

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 preklopljeni operator dodjele za bezbjedno kopiranje objekata koji sadrže npr. atribute tipa "`vector`" ili "`string`" (pa ni atribute tipa "`VektorNd`" i "`Matrica`", koje smo sami razvili), jer će se za ispravno kopiranje objekata pobrinuti konstruktori kopije (odnosno preklopljeni 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. Ova činjenica pruža mogućnost da se u velikom broju praktičnih slučajeva u potpunosti izbjegne potreba za definiranjem konstruktora kopije i preklopljenog 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 iznenađenje, 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 "`VektorNd`" koristeći tip "`vector`" umjesto dinamičke alokacije memorije:

```
class VektorNd {
  int dimenzija;
  vector<double> koordinate;
public:
  explicit VektorN(int n) : dimenzija(n), koordinate(n) {}
  void Ispisi() const;
  double Koordinata(int n) const;
  double &Koordinata(int n);
  friend VektorNd ZbirVektora(const VektorNd &v1, const VektorNd &v2);
};
```

Konstruktor kopije i preklopljeni operator dodjele više nisu potrebni. Obratimo pažnju kako je konstruktor klase "`VektorNd`" iskorišten za inicijalizaciju broja elemenata atributa "`koordinate`" tipa "`vector<double>`". Naravno, sada je potpuno jasno da atribut "`koordinate`" nismo mogli deklarirati deklaracijom poput

```
vector<double> koordinate(n);
```

pa čak ni sličnom konstruktcijom 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 na prethodnim predavanjima. Implementacije metoda "`Ispisi`" i "`Koordinata`" 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, destruktorima, 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 rekvizita (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 ovih predavanja trebamo naglasiti da se tipovi podataka koji unutar sebe sadrže *sve informacije o sebi*, odnosno koji nemaju "prikolice" prikačene na njih nazivaju *POD tipovi* (od engl. Plain Old Data, odnosno "stari dobri podaci"). To su tipovi podataka za čije bezbjedno kopiranje nije potreban konstruktor kopije, i koji pored toga ne sadrže atribute koji posjeduju vlastiti konstruktor kopije. Svi prosti tipovi podataka naslijeđeni iz jezika C spadaju u POD tipove podataka. Također, u POD tipove podataka spadaju i neki novi tipovi podataka definirani u standardnoj biblioteci jezika C++, kao što je, recimo, tip "`complex`" (koji je također generička klasa). S druge strane, tipovi podataka za čije bezbjedno kopiranje je potreban konstruktor kopije nisu POD tipovi. Takvi su, recimo, svi tipovi podataka koji interno sadrže pokazivače na dinamički alocirane resurse (poput tipova "`vector`" i "`string`", kao i tipova poput "`VectorN`", "`Razred`" i "`Matrica`" koje smo razvili u ovom poglavlju). 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. Sa tipovima podataka koji nisu POD tipovi ne smije se bezbjedno manipulirati bajt po bajt pristupajući samo njihovoj internoj strukturi, s obzirom da njihova interna struktura ne sadrži sve neophodne informacije. Stoga se sa njima ne smiju koristiti pojedine funkcije naslijeđene iz jezika C poput "`memcpy`", "`memset`" i "`qsort`", koje manipuliraju sa podacima bajt po bajt, i to isključivo nad njihovom internom strukturom. Umjesto njih treba koristiti odgovarajuće funkcije iz biblioteke "`algorithm`", o čemu smo već govorili ranije. Također, pri korištenju tipova podataka koji nisu POD tipovi podataka potreban je priličan oprez kod rada sa datotekama (pogotovo pri njihovom upisu u binarne datoteke), o čemu ćemo kasnije detaljno govoriti.

Predavanje 11.

Vidjeli smo da strukture i klase, omogućavaju dizajniranje 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 "`Vektor3d`" ili "`VektorNd`" 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 tipovima "`Vektor3d`" odnosno "`VektorNd`" 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 ovakvu 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. Na prethodnom predavanju upoznali smo se sa preklapanjem operatora dodjele, koje predstavlja specijalni slučaj općeg postupka preklapanja operatora, koji ćemo sada detaljno razmotriti.

Preklapanje operatora ostvaruje se uz pomoć *operatorskih funkcija*, koje izgledaju kao i klasične funkcije, samo što umjesto imena imaju ključnu riječ "**`operator`**" iza koje slijedi oznaka nekog od postojećih operatora. Za sada ćemo pretpostaviti da su operatorske funkcije izvedene kao *obične funkcije*, dok ćemo izvedbu operatorskih funkcija kao *funkcija članica* razmotriti nešto kasnije. Operatorske funkcije izvedene kao obične 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`**"), što uključuje i predefinirane tipove definirane u standardnoj biblioteci jezika C++ (kao što su "`vector`", "`string`", itd.). Na primjer, neka je data sljedeća deklaracija klase "`Vektor3d`", u kojoj su, radi jednostavnosti, definirani samo konstruktor i trivijalne metode za pristup atributima klase:

```
class Vektor3d {
  double x, y, z;
public:
  Vektor3d(double x, double y, double z) : x(x), y(y), z(z) {}
  double DajX() const { return x; }
  double DajY() const { return y; }
  double DajZ() const { return z; }
};
```

Za ovakvu klasu možemo definirati sljedeću operatorsku funkciju koja obavlja sabiranje dva vektora (razmak iza riječi "**`operator`**" može se izostaviti):

```
Vektor3d operator +(const Vektor3d &v1, const Vektor3d &v2) {
  return Vektor3d(v1.DajX() + v2.DajX(), v1.DajY() + v2.DajY(),
    v1.DajZ() + v2.DajZ());
}
```

Operatorsku funkciju principijelno je moguće pozvati kao i svaku drugu funkciju. Na primjer, ukoliko su "`a`", "`b`" i "`c`" objekti tipa "`Vektor3d`", sljedeća konstrukcija je sasvim ispravna:

```
c = operator +(a, b);
```

Međutim, uvođenje operatorskih funkcija omogućava da umjesto toga možemo prosto pisati

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

Razmotrimo ovu mogućnost malo općenitije. Neka je "⊗" 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 ⊗ y" prvo pokušava interpretirati kao izraz "**operator** ⊗(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, kao i korisnički definirane pretvorbe, koje se mogu ostvariti konstruktorima sa jednim parametrom (o čemu smo već govorili ranije), 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 načina, jer je tada nejasno koje pretvorbe treba primijeniti.

Slično vrijedi i za slučaj unarnih operatora. Ukoliko je "⊛" neki unarni operator, tada se izraz "⊛x" prvo pokušava interpretirati kao izraz "**operator** ⊛(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 operatoru 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 "Vektor3d", koja omogućava da se unarna varijanta operatora "-" može primijeniti na objekte tipa "Vektor3d" (sa značenjem obrtanja smjera vektora, kao što je uobičajeno u matematici):

```
Vektor3d operator -(const Vektor3d &v) {
  return Vektor3d(-v.DajX(), -v.DajY(), -v.DajZ());
}
```

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 "Vektor3d" možemo definirati i binarni operator "–" za oduzimanje na sljedeći način:

```
Vektor3d operator -(const Vektor3d &v1, const Vektor3d &v2) {
  return Vektor3d(v1.DajX() - v2. DajX(), v1.DajY() - v2.DajY(),
    v1.DajZ() - v2.DajZ());
}
```

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 Vektor3d operator -(const Vektor3d &v1, const Vektor3d &v2) {
  return v1 + -v2;
}
```

Ovdje je iskorištena činjenica da smo prethodno definirali operator "unarni minus" za tip "Vektor3d" (tako da je kompajler "naučio" šta znači konstrukcija "-v2"), kao i da smo definirali binarni operator "+" za isti tip (tako da kompajler 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 operatora "+" i unarnog minusa za objekte tipa "Vektor3d". 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 interpretirana kao da ova operatorska funkcija treba pozvati samu sebe (tj. kao rekurzija, 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 prijatelji 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 "`Vektor3d`" operatorske funkcije koje smo razmatrali deklarirati kao prijateljske funkcije:

```
class Vektor3d {
  double x, y, z;
public:
  Vektor3d(double x, double y, double z) : x(x), y(y), z(z) {}
  double DajX() const { return x; }
  double DajY() const { return y; }
  double DajZ() const { return z; }
  friend Vektor3d operator +(const Vektor3d &v1, const Vektor3d &v2);
  friend Vektor3d operator -(const Vektor3d &v);
  friend Vektor3d operator -(const Vektor3d &v1, const Vektor3d &v2);
};
```

Uz ovakvu deklaraciju, definicije pomenutih operatorskih funkcija mogle bi se pojednostaviti:

```
Vektor3d operator +(const Vektor3d &v1, const Vektor3d &v2) {
  return Vektor3d(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z);
}
Vektor operator -(const Vektor &v) {
  return Vektor(-v.x, -v.y, -v.z);
}
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 obzirom da tada u njoj uopće ne pristupamo atributima 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 kompajler 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:

```
Vektor3d operator *(double d, const Vektor3d &v) {
  return Vektor3d(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 dovešće do prijave greške, odnosno prethodnom definicijom kompajler je "naučio" kako se množi broj sa vektorom, ali ne i vektor sa brojem! Naime, po konvenciji, 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

```
Vektor3d operator *(const Vektor3d &v, double d) {
  return Vektor3d(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 Vektor3d operator *(const Vektor3d &v, double d) {
  return d * v;
}
```

Na ovaj način smo eksplicitno 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 "Vektor3d". Za tu svrhu potrebno je definirati još jednu operatorsku funkciju za binarni operator "*", čija će oba parametra biti tipa "Vektor3d". 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 "·" označava skalarni, a operator "×" 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 Vektor3d &v1, const Vektor3d &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 "*". 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 "×" sa ciljem da nam "a × 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 zbunjujuće, 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 "×" od znaka "/"):

```
Vektor3d operator %(const Vektor3d &v1, const Vektor3d &v2) {
  return Vektor3d(v1.y * v2.z - v1.z * v2.y, v1.z * v2.x - v1.x * v2.z,
    v1.x * v2.y - v1.y * v2.x);
}
```

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 "`*`".

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 ovakvu mogućnost. Pored toga, takva mogućnost bi veoma lako mogla dovesti do formiranja posve neočekivanih i nekontroliranih neželjenih rekurzivnih poziva. S druge strane, operatorske funkcije je *sasvim moguće* definirati za tipove uvedene u standardnoj biblioteci jezika C++, s obzirom da su oni gotovo bez izuzetka realizirani kao klase (preciznije, kao generičke klase). Na primjer, vrlo je lako omogućiti sabiranje primjeraka standardne generičke klase "`vector`" definirane u istoimenoj standardnoj biblioteci sintaksom poput "`v3 = v1 + v2`" prostim definiranjem (generičke) operatorske funkcije poput sljedeće:

```
template <typename Tip>
  vector<Tip> operator +(const vector<Tip> &v1, const vector<Tip> &v2) {
    if(v1.size() != v2.size()) throw "Različite dimenzije!"
    vector<Tip> v3(v1.size());
    for(int i = 0; i < v1.size(); i++) v3[i] = v1[i] + v2[i];
    return v3;
  }
```

Gotovo svi operatori ugrađeni u jezik C++ mogu se preklopiti (tj. dodefinirati). Jedini operatori koje nije moguće preklopiti su operatori "`::`", "`.`", "`.*`", "`?:`" i operatori "**`sizeof`**" i "**`typeof`**", 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:

```
+    -    *    /    %    <<   >>   <    <=   >    >=
==   !=   &    ^    |    &&   ||   =    +=   -=   *=
/=   %=   &=   ^=   |=   <<=  >>=  ->*  ,
```

zatim sljedeće unarne operatore:

```
+    -    !    ~    &    *    ++   --
```

kao i sljedeće specijalne operatore koje je teško svrstati među gore prikazane operatore:

```
[]   ()   ->   new  new[] delete delete[]
```

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 oni imaju kako prefiksni, tako i postfiksni oblik). O preklapanju operatora dodjele smo već govorili (mada ćemo kasnije dati još nekoliko napomena vezanih za njegovo preklapanje), dok ćemo o preklapanju specijalnih operatora govoriti nešto kasnije. Treba naglasiti 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. Jasno je 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 nekoliko riječi kako bi trebalo definirati izvjesne operatore sa ciljem da se održi konzistencija sa načinom djelovanja pojedinih operatora na različite proste ugrađene tipove.

Na prvom mjestu, neki operator je za neki tip potrebno definirati 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 "`Vektor3d`":

```
void operator +=(Vektor &v1, const Vektor &v2) {
  v1.x += v2.x; v1.y += v2.y; v1.z += v2.z;
}
```

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).

Mada navedena definicija operatorske funkcije za operator "`+=`" radi posve dobro, ona nije u potpunosti saglasna sa načinom 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`" (što ima identično značenje kao sekvenca instrukcija "`b += c`" i "`a = b`"). Sa ovako napisanom operatorskom funkcijom i tipom "`Vektor3d`" 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):

```
Vektor &operator +=(Vektor &v1, const Vektor &v2) {
  v1.x += v2.x; v1.y += v2.y; v1.z += v2.z;
  return v1;
}
```

Potreba da se operatori sa pridruživanjem implementiraju odvojeno od odgovarajućih operatora bez pridruživanja otežava održavanje programa, s obzirom da se svaka eventualna izmjena u implementaciju jednog operatora mora izvesti i u implementaciji drugog. Zbog toga je prirodno postaviti pitanje može li se nekako implementacija jednog od njih prosto osloniti na implementaciju drugog, tako da se eventualne prepravke vrše samo na jednom mjestu. Na prvi pogled, djeluje logično implementaciju operatora poput "`+=`" zasnovati na implementaciji operatora "+". Recimo, mogli bismo uraditi nešto poput sljedeće definicije:

```
inline Vektor3d &operator +=(Vektor3d &v1, const Vektor3d &v2) {
  return v1 = v1 + v2;
}
```

Mada nema nikakve sumnje da ovakva definicija radi ispravno, ona je *izrazito nepreporučljiva*, s obzirom na neefikasnost. Naime, ona izraz oblika "`v1 += v2`" efektivno svodi na prividno ekvivalentan, ali mnogo neefikasniji izraz "`v1 = v1 + v2`". Zaista, dok se izraz "`v1 += v2`" može izvesti direktnom izmjenom sadržaja objekta "`v1`", izraz "`v1 = v1 + v2`" prvo zahtijeva da se izvrši *konstrukcija pomoćnog objekta* koji sadrži vrijednost "`v1 + v2`" (što se izvodi pozivom operatorske funkcije za operator "`+`"), nakon čega se tako stvoreni objekat *dodjeljuje* objektu "`v1`" (što tipično zahtijeva *uništavanje* prethodnog sadržaja objekta i *kopiranje* novog sadržaja na mjesto starog sadržaja), dok se kreirani pomoćni objekat *uništava*. Ovo može biti posebno neefikasno ukoliko je objekat takav da se u igru moraju uključiti destruktori, konstruktori kopije i preklopljeni operator dodjele. Zbog toga, operatorsku funkciju za operator "`+=`" *nikada ne treba implementirati na ovaj način* (analogna priča vrijedi i za sve ostale operatore sa pridruživanjem). Umjesto toga, mnogo je bolje neposredno izvesti operatorsku funkciju za operator "`+=`" (na ranije opisani način), a operatorsku funkciju za operator "`+`" izvesti *indirektno* preko već implementiranog operatora "`+=`". Mada ovo djeluje malo neobično, postoji mnogo jednostavnih načina da se ovo izvede. Vjerovatno najbolji način je ovako:

```
inline Vektor operator +(const Vektor &v1, const Vektor &v2) {
  Vektor v3(v1); v3 += v2; return v3;
}
```

Ovdje smo kreirali pomoćni objekat "`v3`" koji vraćamo kao rezultat funkcije. Razlog zbog kojeg nismo operator "`+=`" primijenili direktno nad argumentom "`v1`" leži u činjenici da on ne smije biti izmijenjen. Ako pažljivo analiziramo ovu izvedbu, primijetićemo da ovako definirana operatorska funkcija za operator "`+`" vrši isti broj kreiranja, kopiranja i uništavanja objekata kao i direktna izvedba, tako da nije ništa manje efikasna od direktne izvedbe. Interesantni su pokušaji da se ova izvedba skrati. Naime, dosta autora preporučuju da se posljednje dvije naredbe spoje u jednu, kao u sljedećem primjeru:

```
inline Vektor operator +(const Vektor &v1, const Vektor &v2) {
  Vektor v3(v1); return v3 += v2;
}
```

Međutim, pokazuje se da znatan broj kompajlera nisu osobito dobri sa optimizacijom vraćanja vrijednosti iz funkcije u slučaju kada se iza naredbe "**`return`**" nalazi neki izraz koji mijenja promjenljive koje u njemu učestvuju, tako da ova izvedba tipično generira lošiji mašinski kôd od prethodne verzije (ovo je činjenica koju je pokazala praksa). Tačnije, prva verzija tipično uspijeva eliminirati veći broj nepotrebnih kopiranja u slučaju složenijih objekata. Neki preporučuju i ovakvu izvedbu:

```
inline Vektor operator +(Vektor v1, const Vektor &v2) {
  v1 += v2; return v1;
}
```

Ova izvedba se može još više skratiti, što daje izuzetno kompaktnu izvedbu:

```
inline Vektor operator +(Vektor v1, const Vektor &v2) {
  return v1 += v2;
}
```

Ideja ovih izvedbi je da se prvi parametar prenese *po vrijednosti*, čime će "`v1`" svakako biti *kopija* odgovarajućeg stvarnog parametra, tako da primjena operatora "`+=`" na njega neće uticati na stvarni parametar. Nevolja ove izvedbe je u tome što će se ovo kopiranje zaista i izvršiti. Neko će reći da se i u prvoj prikazanoj izvedbi svakako formalni parametar "`v1`" kopira u pomoćni objekat "`v3`". U načelu je to tačno, ali veliki broj kompajlera je u stanju da izvrši takvu optimizaciju da u potpunosti izbjegne stvaranje objekta "`v3`" i da umjesto toga kopiranje direktno vrši na odredište koje treba da prihvati rezultat operacije "`+`". Istina je da postoje kompajleri koji su u stanju optimizirati kopiranje i u posljednje dvije prikazane verzije, ali su takvi kompajleri znatno rjeđi. Da rezimiramo, prva prikazana verzija generira najbolji mašinski kôd na najvećem broju raspoloživih kompajlera.

Sljedeće pravilo preporučuje da ukoliko smo se uopće odlučili da definiramo operatore za neku klasu, treba definirati i smisao operatora "`==`" i "`!=`". Ovi operatori bi, ukoliko poštujemo "zdravu logiku", trebali testirati jednakost odnosno različitost dva objekta. Za klasu "`Vektor3d`", odgovarajuće operatorske funkcije bi mogle izgledati ovako:

```
bool operator ==(const Vektor3d &v1, const Vektor3d &v2) {
  return v1.x == v2.x && v1.y == v2.y && v1.z == v2.z;
}
bool operator !=(const Vektor3d &v1, const Vektor3d &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 "`!`" primijenjenog na cjelobrojne tipove:

```
inline bool operator !=(const Vektor3d &v1, const Vektor3d &v2) {
  return !(v1 == v2);
}
```

Na ovaj način smo istakli da želimo da značenje izraza "`v1 != 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 njoj 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 "`Vektor3d`" 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 "`Vektor3d`", 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 "`Vektor3d`". 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 atribute 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 prenositi po vrijednosti (primjenom tehnika opisanih na prethodnim predavanjima), tako da će nam kompajler 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 preklopljeni 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 Vektor3d &v) {
  cout << "{" << v.x << "," << v.y << "," << v.z << "}";
  return cout;
}
```

Uzmemo li u obzir da operator "<<" primijenjen na proste ugrađene tipove vraća kao rezultat sam *objekat izlaznog toka* (preciznije, referencu na njega), prethodnu funkciju možemo kraće napisati ovako:

```
ostream &operator <<(ostream &cout, const Vektor3d &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:

```
ostream &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 direktan 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 &cin, Vektor3d &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 koncepata, 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 Kompleksni operator +(const Kompleksni &a) { return a; }
  friend Kompleksni operator -(const Kompleksni &a) {
    return Kompleksni(-a.re, -a.im);
  }
  friend Kompleksni operator +(const Kompleksni &a,
    const Kompleksni &b) {
      return Kompleksni(a.re + b.re, a.im + b.im);
  }
  friend Kompleksni operator -(const Kompleksni &a,
    const Kompleksni &b) {
      return Kompleksni(a.re - b.re, a.im - b.im);
  }
  friend 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 Kompleksni operator /(const Kompleksni &a,
    const Kompleksni &b) {
      double pom(b.re * b.re + b.im * b.im);
      return Kompleksni((a.re * b.re + a.im * b.im) / pom,
        (a.im * b.re - a.re * b.im) / pom);
  }
  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) {
    a.re += b.re; a.im += b.im; return a;
  }
  friend Kompleksni &operator -=(Kompleksni &a, const Kompleksni &b) {
    a.re -= b.re; a.im -= b.im; return a;
  }
  friend Kompleksni &operator *=(Kompleksni &a, const Kompleksni &b) {
    double pom(a.re * b.im + a.im * b.re);
    a.re = a.re * b.re - a.im * b.im; a.im = pom; return a;
  }
  friend Kompleksni &operator /=(Kompleksni &a, const Kompleksni &b) {
    double pom1(a.im * b.re - a.re * b.im),
      pom2(b.re * b.re + b.im * b.im);
    a.re = (a.re * b.re + a.im * b.im) / pom2; a.im = pom1 / pom2;
    return a;
  }
  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.re, a.im); }
    friend Kompleksni conj(const Kompleksni &a) {
      return Kompleksni(a.re, -a.im);
    }
    friend 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 operatora "+", "–", "*" i "/", zatim odgovarajući pridružujući operatori "+=", "–=", "*=" 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 objasniti ulogu konstrukcije "cin.setstate(ios::failbit)" u operatorskoj funkciji za operator ">>". 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? Primijetimo 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 zagradu 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 operator ">>" 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.

Primijetimo da, zbog već opisanih razloga vezanih za efikasnost, operatorske funkcije za operatore poput "+=" nismo implementirali na trivijalan način konstrukcijama poput "**return** a = a + b". Ako po svaku cijenu želimo izbjeći dupliranje koda, onda je bolje direktno implementirati samo operatore poput "+=", a operatore poput "+" implementirati preko njih, na već opisani način.

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, kompajler 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 prirodnije pretvorbe (npr. ugrađene pretvorbe smatraju se prirodnijim od korisnički definiranih pretvorbi, kao što se i pretvorbe između jednog u drugi cjelobrojni tip smatraju prirodnijim od pretvorbe nekog cjelobrojnog u neki realni tip). Na primjer, pretpostavimo da smo definirali i sljedeću operatorsku funkciju:

```
Kompleksni operator *(double d, const Kompleksni &a) {
  return Kompleksni(d * a.re, d * a.im);
}
```

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 operatorska funkcija za množenje dva kompleksna broja (što je nedvojbeno mnogo efikasnije). 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 koja 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 "Vektor3d" 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 "Vektor3d". Stoga ovaj operator možemo definirati pomoću sljedeće operatorske funkcije:

```
Vektor3d &operator ++(Vektor3d &v) {
  v.x++; v.y++; v.z++;
  return v;
}
```

Vjerovatno vam je jasno zašto je formalni parametar "v" deklariran kao referenca na nekonstantni objekat. Također, kao rezultat je vraćena referenca na modificirani objekat, čime ne samo da sprečavamo nepotrebno kopiranje, nego omogućavamo i da vraćena vrijednost bude l-vrijednost, što i treba da bude. Naime, kako se operator "++" može primjenjivati samo na l-vrijednosti, legalni i smisleni izrazi poput "++(++a)" odnosno "++++a" ne bi bili mogući 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 "Vektor3d", 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 (zapravo, 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 "Vektor3d" 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):

```
Vektor3d operator ++(Vektor3d &v, int) {
  Vektor3d 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 operatora vraćaju vrijednost operanda kakva je bila *prije izmjene*), istu funkcionalnost smo simulirali i u prikazanoj izvedbi postfiksne verzije operatora "++" za klasu "Vektor3d". 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).

Operatorske funkcije se mogu definirati i za pobrojane tipove. Na primjer, neka je data sljedeća deklaracija pobrojanog tipa "Dani":

```
enum Dani {Ponedjeljak, Utorak, 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) {
  return d = Dani((int(d) + 1) % 7);
}
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 cijelim 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 slijedi "n" dana nakon 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 ovih operatorskih funkcija vješto iskorišten operator "%". Također, važno je uočiti da je

eksplicitna 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.

Sve do sada, operatorske funkcije smo definirali kao *obične funkcije*, koje nisu funkcije članice klase. Međutim, operatorske funkcije se mogu definirati i kao *funkcije članice klase*. U tom slučaju, interpretacija operatora se neznatno mijenja. Naime, neka je "⊗" neki binarni operator. Već smo vidjeli da se izraz "x ⊗ y" interpretira kao "**operator** ⊗(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 ⊗ y" interpretira kao "x.**operator** ⊗(y)", odnosno uzima se da operatorska funkcija *djeluje nad prvim operandom*, a drugi operand *prihvata kao parametar*. Slično, ukoliko je "✻" neki unarni operator, tada se izraz "✻x" interpretira kao "**operator** ✻(x)" u slučaju da je odgovarajuća operatorska funkcija definirana kao obična funkcija, a kao "x.**operator** ✻()" u slučaju da je definirana kao funkcija članica. Za slučaj operatora "++" 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 fiktivnim cjelobrojnim argumentom).

Pogledajmo kako bi mogle izgledati definicije operatorskih funkcija za binarni operator "+" i unarni minus za klasu "Vektor3d" 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 Vektor3d {
  double x, y, z;
public:
  Vektor3d(double x, double y, double z) : x(x), y(y), z(z) {}
  double DajX() const { return x; }
  double DajY() const { return y; }
  double DajZ() const { return z; }
  Vektor3d operator +(const Vektor3d &v) const;
  Vektor3d operator –() const { return Vektor3d(-x, -y, -z); }
};
```

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 zaglavlje operatorske funkcije članice klase u slučaju kada se ona implementira izvan klase (primijetimo da prva pojava riječi "Vektor3d" označava povratni tip funkcije, dok druga pojava riječi "Vektor3d" zajedno sa operatorom "::" označava da se radi o funkciji članici klase "Vektor3d"):

```
Vektor3d Vektor3d::operator +(const Vektor3d &v) const {
  return Vektor3d(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, s obzirom da se radi o *funkciji članici*, ova imena se odnose na imena *onog konkretnog objekta nad kojim je metoda primijenjena*. 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".

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. Pretpostavimo 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!

Generalno, 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*, čime se uvodi *izrazita asimetrija* 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*. Kako za proste ugrađene tipove većina binarnih operatora poput "+", "–", "*", "/", "%", "<", "<=", ">", ">=", "==", "!=", "&", "^", "|", "&&" i "||" tretira oba svoja operanda *ravnopravno*, ove operatore je 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 "Vektor3d" (pri tome pretpostavljamo da smo u deklaraciji klase "Vektor3d" dodali odgovarajući prototip):

```
Vektor3d &Vektor3d::operator +=(const Vektor3d &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, koji je moguće dohvatiti jedino preko pokazivača "**this**". Zapravo, sa ovakvom situacijom smo se susreli i ranije, u jednoj od ranijih definicija klase "Vektor3d", u kojoj smo imali funkciju članicu "SaberiSa", u kojoj smo koristili istu tehniku. Ovdje operatorska funkcija za operator "+=" nije ništa drugo nego prerušena verzija metode "SaberiSa".

S obzirom da već imamo definiranu operatorsku funkciju za operator "+", mnogi će doći u napast da operatorsku funkciju za operator "+="napišu 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;
}
```

Međutim, ovakva izvedba se nikako ne može preporučiti zbog neefikasnosti (mnoštvo bespotrebnih stvaranja objekata, kopiranja i uništavanja), o čemu smo već raspravljali. Ukoliko želimo iskoristiti zajednički kôd i na jednostavan način održati konzistenciju između definicija operatora "+" i "+=", već smo rekli da je mnogo bolje operator "+" definirati preko operatora "+=" nego obrnuto, što možemo uraditi na jedan od ranije opisanih načina (nema nikakve smetnje što je ovaj put operatorska funkcija za operator "+=" funkcija članica). Međutim, u slučaju kada je operatorska funkcija za operator "+=" funkcija članica, moguća je još jedna vrlo kompaktna izvedba operatorske funkcije za operator "+" preko operatora "+=", koja izgleda ovako:

```
inline Vektor3d operator +(const Vektor3d &v1, const Vektor3d &v2) {
  return Vektor3d(v1) += v2;
}
```

S obzirom da je ovdje operatorska funkcija za operator "+=" funkcija članica, izraz iza naredbe "**return**" interpretira se kao "`Vektor3d(v1).operator +=(v2)`". Ovdje je ideja da se pomoću konstrukcije "`Vektor3d(v1)`" kreira privremeni bezimeni objekat koji je kopija objekta "`v1`" (ovo je zapravo *poziv konstruktora kopije kao funkcije*), nakon čega se na njega *primjenjuje* operatorska funkcija članica za operator "+=" (rezultat te primjene se vraća kao konačni rezultat iz funkcije). Mada se ova izvedba može naći u mnogim dobrim knjigama o jeziku C++ (preporučuju je mnogi C++ "gurui"), veliki broj današnjih kompajlera nije u stanju dobro optimizirati ovako "prljave" konstrukcije, tako da ranije predložena rješenja tipično generiraju znatno efikasniji mašinski kôd na većini raspoloživih kompajlera.

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`", odgovarajuće operatorske funkcije bi trebale biti funkcije članice klasa "`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 klasa "`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 (naravno, druga situacija bi bila da je "`a`" *niz čiji su elementi primjerci neke klase*). Međutim, česta situacija je da imamo klase čiji se primjerci po svojoj prirodi ponašaju *poput nizova* (tipičan primjer su klase "`VektorNd`" i "`Razred`" koje smo ranije razvijali), s obzirom da u sebi čuvaju izvjesne elemente. Takve klase se nazivaju *kontejnerske klase*. Također, moguće je napraviti klase čiji se primjerci po svojoj prirodi ponašaju *poput funkcija* (primjer bi mogla biti klasa koja bi opisivala polinom čiji se koeficijenti mogu zadavati) i one se nazivaju *funkcijske klase*. Preklapanje operatora "`[]`" i "`()`" omogućava da se primjerci klasa čiji se primjerci ponašaju kao nizovi *koriste kao da se radi o nizovima*, odnosno da se primjerci klasa čiji se primjerci ponašaju poput funkcija *koriste kao da se radi o funkcijama*.

Razmotrimo jedan konkretan primjer. U klasi "`VektorNd`" koju smo definirali na prethodnim predavanjima imali smo funkciju članicu "`Koordinata`" koju smo koristili za pristup elementima n-dimenzionalnog vektora. Na primjer, ukoliko je "`v`" objekat klase "`VektorNd`", morali smo koristiti rogobatne konstrukcije poput "`v.Koordinata(3) = 10`" ili "`cout << v.Koordinata(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.Koordinata(i)`" možemo pisati prosto "`v[i]`", sve što treba uradti je promijeniti imena funkcija članica "`Koordinata`" u "**`operator`** `[]`", dok njihova definicija može ostati ista. Slijedi poboljšana izvedba klase "`VektorNd`", 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 VektorNd {
  int dimenzija;
  double *koordinate;
public:
  explicit VektorNd(int n) : dimenzija(n), koordinate(new double[n]) {}
  VektorNd(const VektorNd &v);
  ~VektorNd() { delete[] koordinate; }
  VektorNd &operator =(const VektorNd &v);
  friend ostream &operator <<(ostream &cout, const VektorNd &v);
  double operator [](int n) const;
  double &operator [](int n);
  friend VektorNd operator +(const VektorNd &v1, const VektorNd &v2);
};

VektorNd::VektorNd(const VektorNd &v) : dimenzija(v.dimenzija),
  koordinate(new double[v.dimenzija]) {
    copy(v.koordinate, v.koordinate + v.dimenzija, koordinate);
}

VektorNd &VektorNd::operator =(const VektorNd &v) {
  if(dimenzija < v.dimenzija) {
    delete[] koordinate;
    koordinate = new double[v.dimenzija];
  }
  dimenzija = v.dimenzija;
  copy(v.koordinate, v.koordinate + v.dimenzija, koordinate);
  return *this;
}

ostream &operator <<(ostream &cout, const VektorNd &v) {
  cout << "{";
  for(int i = 0; i < v.dimenzija - 1; i++)
    cout << v.koordinate[i] << ",";
  if(v.dimenzija > 0) cout << v.koordinate[v.dimenzija - 1];
  return cout << "}";
}

double VektorNd::operator [](int n) const {
  if(n < 1 || n > dimenzija) throw "Pogrešan indeks!\n";
  return koordinate[n - 1];
}

double &VektorNd::operator [](int n) {
  if(n < 1 || n > dimenzija) throw "Pogrešan indeks!\n";
  return koordinate[n - 1];
}

VektorNd operator +(const VektorNd &v1, const VektorNd &v2) {
  if(v1.dimenzija != v2.dimenzija)
    throw "Vektori koji se sabiraju moraju biti iste dimenzije!\n";
  VektorNd v3(v1.dimenzija);
  for(int i = 0; i < v1.dimenzija; i++)
    v3.koordinate[i] = v1.koordinate[i] + v2.koordinate[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*. Nije preteško zaključiti da je funkcioniranje tipa "`vector`" definiranog u istoimenoj biblioteci, kao i još nekih srodnih tipova, kao što su "`deque`" i "`valarray`" (o kojem nismo govorili) također zasnovano na preklapanju operatora "`[]`".

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 tabelarno 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 pretpostavke, 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 klasa kod kojih operator "`[]`" kao "indeks" prihvata string. Ovakve klase služe za implementaciju tzv. *asocijativnih nizova* (na primjer, asocijativni niz nazvan "`stanovnistvo`" mogao bi čuvati broj stanovnika pojedinih gradova pri čemu se ime grada koristi kao "indeks", tako da bi npr. izraz "`stanovnistvo["Tuzla"]`" predstavljao broj stanovnika Tuzle).

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 ona definirana 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*. Na primjer, zamislimo da naša klasa "`VektorNd`" 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 "`VektorNd`" (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 $v_1 x + v_2 x^2 + ... + v_N x^N$, pri čemu su $\{v_1, v_2, ... v_N\}$ 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 izmijeniti). Ovo ćemo omogućiti upravo preklapanjem operatora "`()`" za klasu "`VektorNd`". Slijedi definicija izmijenjene klase "`VektorNd`" u kojoj je navedena samo implementacija ove operatorske funkcije, dok implementacije ostalih elemenata klase ostaju iste kao i ranije:

```
class VektorNd {
  int dimenzija;
  double *koordinate;
public:
  ...                                   // Sve ostaje isto kao i ranije
  double operator ()(double x) const;
};

double VektorNd::operator ()(double x) const {
  double suma(0);
  for(int i = dimenzija - 1; i >= 0; i--)
    suma = suma * x + koordinate[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 operatora:

```
VektorNd 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, znamo da se 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. 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 "`VektorNd`" 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]`".

Primjerci klasa koje imaju preklopljen 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 destruktorima, 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 funktor koji predstavlja neku matematičku funkciju, a vraća kao rezultat funktor koji predstavlja *njen izvod*, tako da ukoliko je "`f`" neki funktor, 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, neka je potrebno definirati funkciju "`KumulativnaSuma`" sa jednim parametrom koja vraća kao rezultat ukupnu sumu svih dotada zadanih vrijednosti njenih stvarnih argumenata. Na primjer, želimo da naredba poput

```
for(int i = 1; i <= 5; i++) cout << KumulativnaSuma(i) << endl;
```

ispiše sekvencu brojeva 1, 3, 6, 10 i 15 ($1+2=3$, $1+2+3=6$, $1+2+3+4=10$, $1+2+3+4+5=15$). Njena definicija mogla bi izgledati recimo ovako:

```
int KumulativnaSuma(int n) {
  static int suma(0);
  return suma += n;
}
```

Vidimo da ova funkcija svoje stanje (dotatašnju sumu) čuva u statičkoj promjenljivoj "suma" unutar definicije funkcije (promjenljiva je označena kao statička da bi njena vrijednost ostala nepromijenjena između dva poziva funkcije, a statičke promjenljive se inicijaliziraju samo po prvom nailasku na inicijalizaciju). 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 pristupu 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 zasnivati 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 koliko 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 funkcijsku 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 funktor možemo deklarirati prosto kao instancu ove klase:

```
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 ispisaće slijed brojeva 1, 3, 6, 10, 15, 1, 3, 6 (bez poziva metode "Resetiraj" ispisani slijed bio bi 1, 3, 6, 10, 15, 16, 18, 21).

Treba napomenuti da sve funkcije iz standardne biblioteke "algorithm", koje kao parametre prihvataju funkcije, također prihvataju i *funktore odgovarajućeg tipa*, što omogućava znatno veću fleksibilnost. Biblioteka "functional" sadrži, između ostalog, nekoliko jednostavnih funkcijskih objekata koji obavljaju neke standardne operacije. Tako je, na primjer, definirana generička funkcijska klasa "greater", koja je definirana otprilike ovako:

```
template <typename UporediviTip>
  class greater {
  public:
    bool operator ()(UporediviTip x, UporediviTip y) { return x > y; }
  };
```

Na primjer, ukoliko izvršimo deklaraciju

```
greater<int> veci;
```

tada će izraz "veci(x, y)" za cjelobrojne argumente "x" i "y" imati isto značenje kao izraz "x > y", s obzirom da se on interpretira kao "veci.operator()(x, y)". Isto značenje ima i izraz poput "greater<int>()(x, y)" (koji se interpretira kao "greater<int>().operator()(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 pojednostavljuju korištenje mnogih funkcija iz biblioteke "algorithm". Na primjer, ukoliko želimo sortirati niz "niz" od 100 realnih brojeva u *opadajući poredak*, 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 funkcijske klase "greater", postoje i generičke funkcijske klase "less", "equal_to", "not_equal_to", "greater_equal" i "less_equal", koje respektivno odgovaraju relacionim operatorima "<","==","!=",">=" i "<=", zatim generičke funkcijske klase "plus", "minus", "multiplies", "divides", "modulus", "logical_and" i "logical_or", koje respektivno odgovaraju binarnim operatorima "+", "–", "*", "/", "%", "&&" i "||", i konačno, generičke funkcijske klase "negate" i "logical_not". Kao primjer primjene ovih funkcijskih klasa, navedimo da naredba

```
transform(niz1, niz1 + 10, niz2, niz3, multiplies<int>());
```

prepisuje u niz "niz3" produkte odgovarajućih elemenata iz nizova "niz1" i "niz2" (pretpostavljajući 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 funkcijske klase.

Standardna biblioteka "functional" također sadrži nekoliko vrlo neobičnih ali interesantnih funkcija koje olakšavaju upotrebu funkcija iz biblioteke "algorithm". Te funkcije *primaju funktore kao parametre* i vraćaju *druge funktore kao rezultat*. Najvažnije funkcije iz ove biblioteke su "bind1st" i "bind2nd". Funkcija "bind1st" prima dva parametra, nazovimo ih f i x_0, pri čemu je f funktor koji prima dva parametra (nazovimo ih x i y). Ova funkcija vraća kao rezultat novi funktor, nazovimo ga g, koji prima samo jedan parametar (nazovimo ga y) takav da vrijedi $g(y) = f(x_0, y)$. Na primjer, ako je "f" neki konkretan funktor, izraz "bind1st(f, 5)(y)" ima isto dejstvo kao izraz "f(5, y)". Slično, funkcija "bind2nd" prima dva parametra f i y_0, i vraća kao rezultat novi funktor g, takav da vrijedi $g(x) = f(x, y_0)$, odnosno izraz poput "bind2nd(f, 5)(x)" ima isto dejstvo kao izraz "f(x, 5)". Kao primjer primjene, neka imamo niz "niz" od 100 cjelobrojnih elemenata, i neka želimo prebrojati i ispisati koliko u njemu ima elemenata čija je vrijednost manja od 30. To možemo izvesti pomoću samo jedne naredbe na sljedeći način:

```
cout << count_if(niz, niz + 100, bind2nd(less<int>(), 30));
```

Naime, poziv "less<int>()" kreira funktor (nazovimo ga f) takav da $f(x, y)$ ima vrijednost "**true**" ako i samo ako je $x < y$. Stoga će izraz "bind2nd(less<int>(), 30)" kreirati novi funktor (nazovimo ga g) takav da je $g(x) = f(x, 30)$, odnosno $g(x)$ ima vrijednost "**true**" ako i samo ako je $x < 30$. Tako kreirani funktor prosljeđuje se funkciji "count_if" kao kriterij, a dalje se sve dešava u skladu sa već opisanim dejstvom funkcije "count_if". Treba napomenuti da funkcije "bind1st" i "bind2nd" primaju kao parametre isključivo *funktore*, tako da im se ne može kao parametar poslati klasična funkcija. Međutim, u biblioteci "functional" postoji i funkcija "ptr_fun" koja kao parametar prima *proizvoljnu funkciju* od jednog ili dva argumenta, a vraća kao rezultat *funktor* potpuno identičnog dejstva kao i funkcija zadana kao parametar, tako da se rezultat funkcije "ptr_fun" može iskoristiti kao argument u funkcijama poput "bind1st" i "bind2nd". Inače, funkcije poput "bind1st" i "bind2nd"

u teoriji programiranja nazivaju se *veznici* (engl. *binders*), dok je "`ptr_fun`" tipičan primjer funkcije *adaptera* (engl. *adapters*).

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`**" i "**`delete`**". Operator "`->`" preklapa se operatorskom funkcijom članicom "**`operator`** `->`" bez parametara, pri čemu se izraz oblika "`x->y`" u slučaju kada je "`x`" primjerak neke klase interpretira kao "`(x.operator->())->y`". Preklapanje ovog operatora omogućava korisniku da kreira objekte koje se ponašaju *poput pokazivača na primjerke drugih struktura ili klasa* i koji se obično nazivaju *iteratori* (takvi objekti tipično imaju preklopljen i *operator dereferenciranja*, tj. unarni operator "`*`"). S druge strane, preklapanje operatora "**`new`**" 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 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 kursa.

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 ugrađeni 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 "`VektorNd`" 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 VektorNd {
  ...
public:
  ...
  operator double() const;
};
```

```
VektorNd::operator double() const {
  double suma(0);
  for(int i = 0; i < dimenzija; i++)
    suma += koordinate[i] * koordinate[i];
  return sqrt(suma);
}
```

Ovim postaju moguće sljedeće stvari: eksplicitna konverzija objekata tipa "`VektorN`" u tip "**`double`**" pomoću operatora za pretvorbu tipa (type-castinga), zatim neposredna dodjela promjenljive ili izraza tipa "`VektorNd`" promjenljivoj tipa "**`double`**", kao i upotreba objekata tipa "`VektorNd`" u izrazima na mjestima gdje bi se normalno očekivao operand tipa "**`double`**" (pri tome bi u sva tri slučaja vektor prvobitno bio pretvoren u realni broj). Na primjer, ukoliko je "`v`"promjenljiva tipa "`VektorNd`" a "`d`" promjenljiva tipa "**`double`**", sljedeće konstrukcije su sasvim legalne:

```
cout << (double)v;
cout << double(v);
cout << static_cast<double>(v);
d = v;
cout << sin(v);
```

Napomenimo da je funkcijska 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 kompajler. 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, kompajler ne može razriješiti situaciju, i prijavljuje se greška. Do greške neće doći jedino ukoliko eksplicitno specificiramo koju konverziju želimo, konstrukcijama poput "`Tip2(x)`" (odnosno nekim analognim konstrukcijama poput "`(Tip2)x`" ili "`static_cast<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. Pretpostavimo, 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). Pretpostavimo 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.

Predavanje 12.

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. Da bi program bio *objektno orjentiran* (a ne samo *objektno zasnovan*), u njemu se moraju koristiti i ova dva načela. 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 klasa 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*, *inherited 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 *jeste* 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 ima smisla klasu "`DiplomiraniStudent`" 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 potvrdni, 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 potvrdan, 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 "`Vektor2d`" i "`Vektor3d`" koje predstavljaju respektivno dvodimenzionalni odnosno trodimenzionalni vektor. Očigledno će klasa "`Vektor3d`" sadržavati sve atribute i metode kao i klasa "`Vektor2d`" (i jednu koordinatu više), tako da na prvi pogled djeluje prirodno koristiti nasljeđivanje i definirati klasu "`Vektor3d`" kao izvedenu klasu iz klase "`Vektor2d`". 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 primjerci naslijeđene klase koriste u svim kontekstima u kojem se mogu koristiti i primjerci bazne klase. To znači da bi se primjerci klase "`Vektor3`" mogli koristiti svugdje gdje i primjerci klase "`Vektor2`". Jasno je da to ne mora biti opravdano, s obzirom da trodimenzionalni vektori *nisu* dvodimenzionalni vektori, tako da je sasvim moguće zamisliti operacije koje su definirane za dvodimenzionalne vektore a nisu za trodimenzionalne vektore. Stoga je najbolje definirati klase "`Vektor2d`" i "`Vektor3d`" posve neovisno jednu od druge, bez korištenja ikakvog nasljeđivanja. Možda djeluje pomalo apsurdno, ali ukoliko baš želimo da koristimo nasljeđivanje, tada je

bolje *klasu* "Vektor2d" *naslijediti iz klase* "Vektor3d"! Zaista, dvodimenzionalni vektori *jesu* specijalni slučaj trodimenzionalnih vektora i mogu se koristiti u svim kontekstima gdje i trodimenzionalni vektori. Na prvi pogled djeluje dosta neobično da je ovdje nasljeđivanje opravdano, 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 klasa "Vektor2d" i "Vektor3d" 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 postojeća 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 naslijeđena iz klase "A". Umjesto toga, klasa "B" treba da sadrži *atribut koji je tipa* "A". Na primjer, klasa "Datum" sigurno će imati atribute koji čuvaju pripadni dan, mjesec i godinu. Klasa "Student" mogla bi imati te iste atribute (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 atribute 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 DajIme() const { return ime; }
  int DajIndeks() const { return indeks; }
  void Ispisi() const {
    cout << "Student " << ime << " ima indeks " << indeks;
  }
};
```

Ukoliko želimo da definiramo klasu "DiplomiraniStudent" koja je nasljeđena 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 DajGodinuDiplomiranja() const { return godina_diplomiranja; }
};
```

Nasljeđivanje se u jeziku C++ 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 atribute ili metode koje dodajemo (ili metode koje *mijenjamo*, što ćemo uskoro demonstrirati). Svi ostali atributi i metode prosto se nasljeđuju iz bazne klase. Međutim, ovdje postoji jedan važan izuzetak: *konstruktori se nikada ne nasljeđuju*. Ukoliko je bazna klasa posjedovala konstruktore, izvedena klasa ih mora ponovo

definirati. Razlog za ovo je činjenica da su konstruktori namijenjeni da *inicijaliziraju* elemente objekta, a objekti nasljeđene klase gotovo uvijek imaju dodatne atribute 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 atribute koji postoje u baznoj klasi, dok će novododani atributi ostati neinicijalizirani. U svim ostalim slučajevima kompajler 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. Stoga konstruktor izvedene klase gotovo po pravilu *mora pozvati konstruktor bazne klase* (ovo pozivanje se može izostaviti jedino ukoliko bazna klasa posjeduje konstruktor bez parametara, koji će tada automatski biti pozvan iz konstruktora izvedene klase). Poziv konstruktora bazne klase izvodi se isključivo 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 atribute "`ime`" i "`indeks`", a pored toga (također u konstruktorskoj inicijalizacijskoj listi) inicijalizira i svoj specifični atribut "`godina_diplomiranja`". Napomenimo da ovo pravilo vrijedi i za konstruktor kopije. Ukoliko bazna klasa ima definiran konstruktor kopije, mora ga imati i izvedena klasa, pri čemu će konstruktor kopije izvedene klase obavezno pozvati konstruktor kopije bazne klase i još eventualno odraditi ono što je potrebno da bi se dobila korektna kopija objekta izvedene klase.

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 direktan 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!

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`**". Metode svih klasa naslijeđenih iz neke klase *mogu pristupiti* onim atributima i metodama *objekta nad kojim su pozvani* koji imaju zaštićeno pravo pristupa (tj. čija su prava pristupa označena sa "**`protected`**"), što ne bi bilo moguće kada bi oni imali privatno pravo pristupa. U svim ostalim aspektima, atributi i metode sa zaštićenim pravom pristupa ponašaju se kao da imaju privatno pravo pristupa. Stoga, ukoliko bismo atributima "`ime`" i "`indeks`" dali zaštićeno pravo pristupa, sve metode klase "`DiplomiraniStudent`" (uključujući i konstruktor) mogle bi im u slučaju potrebe pristupiti, dok bi pristup i dalje bio onemogućen u ostalim dijelovima programa. 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 DajIme() const { return ime; }
  int DajIndeks() const { return indeks; }
  void Ispisi() const {
    cout << "Student " << ime << " ima indeks " << indeks;
  }
};
```

Često se može čuti prilično neprecizno tumačenje prema kojem su atributi i metode sa zaštićenim pravom pristupa *dostupni metodama svih klasa nasljeđenih iz klase u kojoj su definirani*. Međutim, ukoliko pažljivo razmotrimo šta zapravo znači zaštićeno pravo pristupa, vidjećemo da je ovo tumačenje samo djelimično tačno. Na primjer, sve metode ma koje klase naslijeđene iz klase "Student" (recimo klase "DiplomiraniStudent") zaista će moći pristupiti atributima "ime" i "indeks" koji se odnose na onaj objekat nad kojima su pozvane (tj. na onaj objekat na koji pokazuje pokazivač "**this**"), ali *neće moći pristupiti* atributima "ime" i "indeks" koji se odnose *na neki drugi objekat* tipa "Student" (ili tipa neke klase naslijeđene iz klase "Student") koji nije onaj objekat nad kojim je metoda pozvana. Na primjer, niti jedna metoda klase "DiplomiraniStudent" ne može pristupiti atributima "ime" i "indeks" neke promjenljive "s" tipa "Student" definirane negdje drugdje, bez obzira što oni imaju zaštićena prava pristupa a ta metoda pripada klasi naslijeđenoj iz klase "Student". To vrijedi čak i ukoliko je "s" lokalna promjenljiva tipa "Student" deklarirana unutar te metode. Mogli bismo reći da metode klasa naslijeđenih iz neke klase "A" imaju pravo pristupa samo *svojim naslijeđenim atributima i metodama sa zaštićenim pravima pristupa*, ali ne i atributima i metodama sa zaštićenim pravima pristupa *drugih objekata* tipa "A". Mada ovo pravilo djeluje pomalo neobično i isuviše restriktivno, postoje jaki razlozi zašto je tako. Zaista, pretpostavimo da nije tako, i da je "a" neki objekat tipa "A" koji posjeduje zaštićeni atribut "x". Tada bi zlonamjerni programer mogao bez problema pristupiti atributu "x" objekta "a" tako što bi kreirao neku klasu naslijeđenu iz klase "A" i u njoj definirao neku metodu koja bi direktno pristupila atributu "x" objekta "a" (bez prethodno opisanih restrikcija, to bi bilo moguće). Drugim riječima, bez ovakvih restrikcija, mehanizam zaštite bi se mogao veoma lako zaobići, tako da zaštita u suštini ne bi bila nikakva zaštita!

U nastavku ćemo podrazumijevati da je atributima "ime" i "indeks" dato zaštićeno pravo pristupa (mada mnoge preporuke govore da zaštićeno pravo pristupa treba koristiti sa velikom rezervom i da je bolje držati što je god moguće više stvari u privatnom dijelu klase ukoliko je ikako moguće). Ipak, bez obzira na ova nešto slobodnija prava pristupa, bitno je naglasiti da se u konstruktorskoj inicijalizacionoj listi mogu inicijalizirati *samo atributi koji su neposredno deklarirani u toj klasi*, bez obzira na prava pristupa atributa u baznoj klasi.

Nad objektima nasljeđene 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 nasljeđenoj klasi *promijene* definicije nekih metoda koje su definirane u baznoj klasi. Na primjer, u klasi "DiplomiraniStudent" ima smisla promijeniti definiciju metode "Ispisi" tako da ona uzme u obzir i godinu diplomiranja studenta. Izmijenjena 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 DajGodinuDiplomiranja() 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 "`DajIme`" i "`DajIndeks`" 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 naslijeđena iz klase "`Student`". Naslijeđena 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 operatora "`::`". Tako, ukoliko bismo nad objektom "`s2`" željeli da pozovemo metodu "`Ispisi`" naslijeđenu 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 naslijeđenoj klasi pozvati istoimenu metodu naslijeđenu iz bazne klase. Na primjer, u sljedećoj definiciji klase "`DiplomiraniStudent`" metoda "`Ispisi`" poziva istoimenu metodu naslijeđenu iz njene bazne klase:

```
class DiplomiraniStudent : public Student {
  int godina_diplomiranja;
public:
  DiplomiraniStudent(string ime, int ind, int god_dipl) :
    Student(ime, ind), godina_diplomiranja(god_dipl) {}
  int DajGodinuDiplomiranja() 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 odrednica "`Student::`" ispred poziva metode "`Ispisi`" *veoma bitna*, jer bi bez nje kompajler 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 definirala). Zbog toga se ne nasljeđuju ni automatske pretvorbe tipova koje se ostvaruju eventualnim konstruktorima sa jednim parametrom (koji nisu označeni sa "**`explicit`**"), nego ih naslijeđena 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*. Preklopljeni 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`" naslijeđene iz klase "`A`". Ovdje ne treba krivo pomisliti da se prijateljske funkcije ne nasljeđuju. One se nasljeđuju, u smislu da su prijateljske funkcije bazne klase sasvim dostupne i svim objektima ma koje klase naslijeđene iz nje, ali te funkcije *nisu prijatelji* naslijeđene klase (tj. nemaju pravo pristupa njenim privatnim atributima). Ukoliko želimo da zadržimo prijateljstvo i u naslijeđenoj klasi, funkciju "`f`" ponovo treba proglasiti prijateljem klase unutar deklaracije klase "`B`". Osim konstruktora, operatora dodjele i relacije prijateljstva, svi ostali elementi bazne klase nasljeđuju se u izvedenoj klasi (uključujući i sve druge operatorske funkcije osim za operator dodjele).

Sasvim je moguće da više klasa bude naslijeđeno iz iste bazne klase. Pored toga, možemo imati i čitav lanac nasljeđivanja. Na primjer, klasa "C" može biti naslijeđena iz klase "B", koja je opet naslijeđena iz klase "A". Konačno, moguće je da neka klasa naslijedi više klasa, tj. da bude naslijeđena iz više od jedne bazne klase. Na primjer, sljedeća konstrukcija

```
class C : public A, public B {
  ...
};
```

deklarira klasu "C" koja je naslijeđena iz baznih klasa "A" i "B". U ovom slučaju se radi o tzv. *višestrukom nasljeđivanju*. Mada višestruko nasljeđivanje u nekim specifičnim primjenama može biti korisno, ono sa sobom vuče mnoge nedoumice koje se moraju posebno razmotriti (npr. šta se dešava ukoliko više od jedne bazne klase imaju neke atribute 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.). U doba kada jezik C++ nije posjedovao generičke klase, višestruko nasljeđivanje se intenzivno koristilo za postizanje nekih funkcionalnosti, koje se danas mnogo prirodnije rješavaju uz pomoć generičkih klasa. Stoga se višestruko nasljeđivanje danas znatno manje koristi, tako da se njegovom opisivanju nećemo zadržavati, tim prije što nisu rijetka mišljenja da višestruko nasljeđivanje često unosi više zbrke nego što donosi koristi.

Već je rečeno da se svaki objekat naslijeđene klase može koristiti u kontekstima u kojima se može koristiti objekat bazne klase. Tako se objekat bazne klase može inicijalizirati objektom naslijeđene klase, zatim može se izvršiti dodjela objekta naslijeđene 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 naslijeđene klase. Ovo je dozvoljeno zbog toga što svaki objekat naslijeđene klase sadrži sve elemente koje sadrže i objekti bazne klase. Međutim, da bi ovakva mogućnost imala smisla, veoma je važno se svaki objekat naslijeđene 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). Ovakvo ponašanje naziva se *odsjecanje* (engl. *slicing*).

Veoma je važno shvatiti da je odnos između bazne i naslijeđene klase *strogo jednosmjeran*, tako da se u općem slučaju objekti bazne klase *ne mogu koristiti* u kontekstima u kojima se mogu koristiti objekti naslijeđene 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 "s2 = s1" gdje su "s1" i "s2" promjenljive iz prethodnog primjera nije legalna, iako dodjela "s1 = s2" jeste. Ovo je sasvim razumljivo, jer promjenljiva "s2" ima dodatni atribut koji promjenljiva "s1" nema (godinu diplomiranja), pa je nejasno šta bi trebalo dodijeliti atributu "godina_diplomiranja" objekta "s2". 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 (ili ukoliko bazna klasa posjeduje *operator konverzije u tip izvedene klase*). U tom slučaju, takav konstruktor (ili operator konverzije) će nedvosmisleno odrediti kako treba tretirati atribute 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 javno pravo pristupa zadržavaju to pravo pristupa i u izvedenoj klasi, kod privatnog odnosno zaštićenog izvođenja elementi osnovne klase koji su imali javno pravo pristupa u izvedenoj klasi će imati privatno odnosno zaštićeno pravo 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 eksplicitno definirane kao metode sa javnim pristupom unutar izvedene klase (koje, naravno, mogu pozivati neke od metoda bazne klase). Alternativno je moguće pomoću ključne riječi "**using**" specificirati da neka od metoda koja je činila interfejs bazne klase treba (neizmijenjena) da bude sadržana i u interfejsu privatno izvedene klase. Na primjer, ukoliko klasa "A" sadrži metodu "NekaMetoda" u svom interfejsu, i ukoliko je klasa "B" privatno izvedena iz klase "A", tada deklaracijom "**using** A::NekaMetoda" unutar interfejsa klase "B" naglašavamo da metoda "NekaMetoda" treba da čini i interfejs klase "B".

Iz izloženog neposredno slijedi da se privatno izvedene klase ne mogu smatrati kao specijalni slučajevi bazne klase, niti se primjerci tako izvedenih klasa mogu koristiti u istim kontekstima kao i primjerci bazne klase. Stoga je jasno da se privatnim ili zaštićenim nasljeđivanjem *ne realizira nasljeđivanje*. Primjerci klase koja je privatno odnosno zaštićeno izvedena iz bazne klase tretiraju se posve neovisno od primjeraka 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 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 objekte bazne klase može dodijeliti adresa nekog objekta nasljeđene klase ili neki drugi pokazivač na objekat nasljeđene klase (ovo vrijedi samo za *nasljeđivanje* odnosno *javno izvođenje* – prilikom privatnog odnosno zaštićenog izvođenja ova konvencija ne vrijedi). Također, svaka funkcija koja prima kao parametar ili vraća kao rezultat pokazivač na objekte bazne klase može primiti kao parametar ili vratiti kao rezultat pokazivač na neki objekat izvedene klase (recimo 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 inicijalizaciju prilikom deklariranja ovih pokazivača, tako da bismo isti efekat postigli deklaracijom

```
Student *pok1(&s1), *pok2(&s2);
```

S obzirom da se pri upotrebi nasljeđivanja tip objekta na koji pokazivač pokazuje može razlikovati od tipa koji je naveden pri deklaraciji pokazivača, uvodi se pojam *statičkog* i *dinamičkog tipa* nekog pokazivača. Statički tip je određen deklaracijom, tj. to je onaj tip koji je naveden prilikom deklaracije pokazivača. U gornjem primjeru, statički tip oba pokazivača "`pok1`" i "`pok2`" je pokazivač na objekte tipa "`Student`" (tj. "`Student *`"). Statički tip nekog pokazivača se ne može mijenjati tokom života pokazivača. S druge strane, dinamički tip pokazivača je određen tipom objekta na koji pokazivač *u tom trenutku pokazuje* i on se može mijenjati tokom života pokazivača. U gornjem primjeru, dinamički tip pokazivača "`pok1`" je i dalje pokazivač na objekat tipa "`Student`", dok je dinamički tip pokazivača "`pok2`" pokazivač na objekat tipa "`DiplomiraniStudent`" (tj. "`DiplomiraniStudent *`").

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 proizvesti sljedeći ispis:

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 zaista pokazuju. Drugim riječima, odluka je donesena na osnuvu *statičkog tipa* ovih pokazivača. Ovakva strategija naziva se *rano povezivanje* (engl. *early binding*), s obzirom da kompajler 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 pokazivaču 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 moralo doći*. Zaista, bez obzira što je "`pok2`" pokazivač na tip "`Student`", on je usmjeren da pokazuje na objekat tipa "`DiplomiraniStudent`", koji *postoji* i već se nalazi 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* (odnosno, na osnovu njegovog *dinamičkog tipa*). 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*. Drugim riječima, ukoliko je funkcija članica virtualna, odluka o tome koja će se verzija funkcije članice pozvati (u slučaju kada ih ima više) donosi se na osnovu *dinamičkog tipa pokazivača*, a ne *statičkog tipa* kao što je to slučaj kod običnih funkcija članica. 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) {}
  string DajIme() const { return ime; }
  int DajIndeks() 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 naslijeđene 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 sintaksno prerušeni pokazivači (tako da i kod referenci možemo razlikovati njihov statički i dinamički tip, kao i kod pokazivača). Stoga, mehanizam poziva virtualnih funkcija radi i sa referencama (virtualne funkcije se pozivaju na osnovu dinamičkog a ne statičkog tipa reference). Pretpostavimo, na primjer, da imamo deklariranu sljedeću funkciju, čiji je formalni parametar referenca na konstantni objekat tipa "`Student`":

```
void NekaFunkcija(const Student &s) {
  s.Ispisi();
}
```

Pretpostavimo 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, ukoliko je metoda "`Ispisi`" unutar klase "`Student`" deklarirana kao virtualna metoda, kasno povezivanje će obezbijediti da iz funkcije "`NekaFunkcija`" bude pozvana metoda "`Ispisi`" iz klase "`DiplomiraniStudent`" bez obzira što je formalni parametar "`s`" deklariran kao referenca na objekat tipa "`Student`" (njegov *dinamički tip* će biti referenca na objekat "`DiplomiraniStudent`"). 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. Također, do ovoga neće doći ukoliko formalni parametar "`s`" nije referenca (nego prosto objekat tipa "`Student`"), bez obzira da li je metoda "`Ispisi`" virtuelna ili ne. Naime, u tom slučaju se pri pozivu funkcije objekat "`s2`" *kopira* u parametar "`s`" pri čemu se, kako smo već vidjeli, *gube sve specifičnosti* objekta "`s2`"! Zapravo, mehanizam virtualnih funkcija i kasnog povezivanja nikada ne djeluje kada se neki objekat naslijeđene 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 naslijeđene klase skoro sigurno oslanja na specifičnosti naslijeđene klase). Možemo reći i ovako: samo kod pokazivača i referenci ima smisla govoriti o statičkom i dinamičkom tipu, dok svi drugi tipovi objekata imaju samo statički tip (mada je moguće simulirati postojanje dinamičkog tipa i kod nekih drugih vrsta objekata, ali ono ne dolazi samo po sebi kao u slučaju pokazivača i referenci, već ga je potrebno implementirati).

Kasno povezivanje i virtualne metode omogućavaju mnogobrojne interesantne primjene. Međutim, prije nego što nastavimo dalje, načinićemo još jednu izmjenu u klasi "`Student`" tako što ćemo joj dodati *virtualni destruktor* sa praznim tijelom, čija će uloga biti uskoro razjašnjena:

```
class Student {
protected:
  string ime;
  int indeks;
public:
  Student(string ime, int ind) : ime(ime), indeks(ind) {}
  virtual ~Student() {}
  string DajIme() const { return ime; }
  int DajIndeks() const { return indeks; }
  virtual void Ispisi() const {
    cout << "Student " << ime << " ima indeks " << indeks;
  }
};
```

Razmotrimo sada sljedeći programski isječak:

```
Student *s;
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 "`.`"). Promjenljiva "`s`" je zapravo promijenila svoj *dinamički tip*. 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 prosto 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 eksplicitno ne naglasimo drugačije.

U prethodnom slučaju, "`s`" je tipičan primjer *polimorfne promjenljive*. U jeziku C++, jedino se pokazivačke promjenljive i reference mogu direktno ponašati kao polimorfne promjenljive. Pri tome, 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. Pored pokazivača i referenci, polimorfno se mogu ponašati još jedino promjenljive tipa neke klase koja unutar sebe sadrži neki atribut koji je pokazivač ili referenca koji se ponašaju polimorfno. Na taj način je moguće kreirati polimorfne objekte, čija je pokazivačka priroda sakrivena od korisnika u implementaciji klase. Primjer ovakve tehnike razmotrićemo u Predavanju 14.

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 predavanja).

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 naslijeđenih 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();
```

Ostaje još da razmotrimo zbog čega smo u baznoj klasi "`Student`" definirali virtualni destruktor. Ne smijemo zaboraviti da kada god pomoću operatora "**`delete`**" uklanjamo neki dinamički alocirani objekat, nad tim objektom se izvršava njegov destruktor. Međutim, ukoliko destruktor nije virtualan, odluka o tome koji se destruktor poziva donosi se na osnovu statičkog tipa pokazivača na koji je operator "**`delete`**" primijenjen. To znači da će ukoliko je "`s`" deklariran kao pokazivač na tip "`Student`", nakon što izvršimo "**`delete`** `s`" biti pozvan destruktor iz klase "`Student`", neovisno od toga na šta zaista pokazuje pokazivač "`s`". Ukoliko u nekoj klasi nije uopće definiran destruktor, kompajler će sam generirati podrazumijevani destruktor (sa praznim tijelom), ali koji *nije virtualan* (razlog za to je što virtualnost ima i svoju cijenu, a filozofija jezika C++ je da nikada ne treba plaćati za ono što se neće koristiti). Deklariranjem virtualnog destruktora u baznoj klasi garantiramo da će prilikom poziva operatora "**`delete`**" biti uvijek pozvan ispravan destruktor. Neko će se vjerovatno zapitati zašto je ovo potrebno čak i u slučaju kada i bazna i sve izvedene klase uopće nemaju destruktor, tj. kada imaju samo podrazumijevani destruktor koji autmatski generira kompajler. Može izgledati da je svejedno koji će se od njih pozvati, jer niti jedan od njih ne radi ništa. Međutim, bitno je znati da destruktori nisu funkcije članice (iako liče na njih), tako da činjenica da je destruktor sa praznim tijelom ipak ne znači da on ne radi ništa. Postoje neke podrazumijevane akcije koje destruktor izvršava čak i ako mu je tijelo prazno, a mi unutar tijela destruktora samo dopisujemo *dodatne akcije* koje ne spadaju u okvir podrazumijevanih akcija. Stoga, čak i ukoliko su automatski generirani, nema garancije da će konstruktor bazne i izvedenih klasa raditi istu stvar, tako da pozivanje krivog destruktora može imati fatalne posljedice. Da rezimiramo, kada god želimo koristiti polimorfizam bazna klasa *mora imati virtualni destruktor*, makar sa praznim tijelom. U suprotnom, posljedice su posve nepredvidljive (ukoliko Vam izgleda da u nekim situacijama program radi ispravno i bez virtualnog destruktora, ne mora značiti da će tako biti i sa nekim drugim kompajlerom u odnosu na onaj koji trenutno koristite). Preciznije, virtualni destruktor u baznoj klasi je neophodan *kad god imamo potrebu da uklanjamo neki objekat izvedene klase preko pokazivača na baznu klasu* (a to je praktično uvijek kada želimo koristiti polimorfizam).

Jedna od tipičnih primjena kasnog povezivanja i polimorfizma je kreiranje *heterogenih nizova* (ili *heterogenih vektora*, ili, još općenitije, *heterogenih kontejnerskih objekata*) koji kao svoje elemente mogu prividno sadržavati objekte *različitih tipova*. Striktno posmatrano, njihovi elementi zapravo *pokazuju na objekte različitih tipova*, ali u većini konceptualnih razmatranja, ovaj implementacioni detalj možemo ignorirati (možemo reći da oni sadrže pokazivače *različitih dinamičkih 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 naslijeđene 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 primjerci te klase, samo što za pristup atributima i metodama umjesto operatora "." koristimo operator "->". Dalje, kako je metoda "Ispisi" virtualna, njena primjena nad pojedinim pokazivačima dovodi do različitog dejstva u zavisnosti od toga na kakav objekat konkretan 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 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. Primijetimo 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 naslijeđenoj klasi a ne i u baznoj klasi, čak i ukoliko taj pokazivač trenutno pokazuje na objekat naslijeđene 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 "DajGodinuDiplomiranja", bez obzira što je on inicijaliziran tako da pokazuje na objekat klase "DiplomiraniStudent":

```
Student *s(new DiplomiraniStudent("Miki Maus", 3412, 2004));
cout << s->DajGodinuDiplomiranja();
```

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 "DajGodinuDiplomiranja" 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 naslijeđenoj klasi, a *sigurni smo* da u posmatranom trenutku taj pokazivač *zaista pokazuje na objekat izvedene klase* (odnosno, ukoliko smo sigurni da je u posmatranom trenutku njegov dinamički tip takav da poziv željene metode ima smisla) možemo na pokazivač primijeniti eksplicitnu konverziju tipa u tip pokazivača na izvedenu klasu pomoću operatora 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)->DajGodinuDiplomiranja();
```

Alternativno, umjesto prethodne konstrukcije koja koristi sintaksu za pretvorbu tipa u C stilu, možemo koristiti i sintaksu preporučenu u jeziku C++, koja za tu svrhu koristi ključnu riječ "**static_cast**":

```
cout << static_cast<DiplomiraniStudent *>(s)->DajGodinuDiplomiranja();
```

Međutim, bez obzira na korištenu sintaksu, ovakve konstrukcije preduzimamo *na vlastitu odgovornost*, s obzirom da posljedice mogu biti posve nepredvidljive ukoliko "s" ne pokazuje na objekat koji je tipa "DiplomiraniStudent" (tj. ukoliko mu dinamički tip nije odgovarajući). 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 << static_cast<DiplomiraniStudent *>(s)->DajGodinuDiplomiranja();
else
  cout << "Žalim, s ne pokazuje na diplomiranog studenta!";
```

Operator "**typeid**" može se primijeniti na neki izraz ili ime tipa, a kao rezultat daje izvjesnu kolekciju informacija o tipu onoga na šta je primijenjen (tačna priroda tih informacija uglavnom nije bitna, niti je posve precizno specificirana). Alternativno, može se vršiti i tzv. *dinamička pretvorba pokazivača* uz pomoć ključne riječi "**dynamic_cast**". Ona se sintaksno vrši identično kao i klasična pretvorba pomoću ključne riječi "**static_cast**" (tzv. *statička pretvorba*), samo je razlika u tome što se kao rezultat dinamičke pretvorbe dobija vrijednost 0 (*nul-pokazivač*) u slučaju da se pretvorba ne može obaviti zbog toga što izvorni pokazivač ne pokazuje na objekat koji odgovara željenom tipu pretvorbe (tj. ukoliko dinamički tip pokazivača nije u skladu sa tipom u koji želimo izvršiti pretvorbu). Sljedeći primjer ilustrira dinamičku pretvorbu pokazivača:

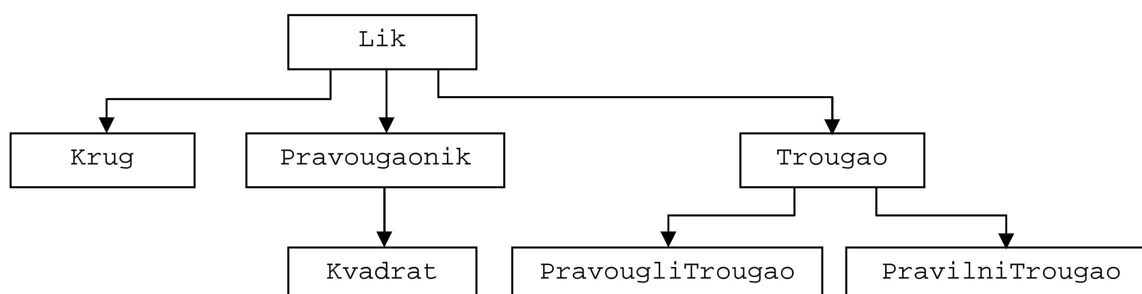
```
DiplomiraniStudent *s1(dynamic_cast<DiplomiraniStudent *>(s));
if(s1 != 0) s1->DajGodinuDiplomiranja();
else cout << "Žalim, s ne pokazuje na diplomiranog studenta!";
```

U principu, korištenje dinamičke pretvorbe je mnogo moćnije (i manje efikasno) od upotrebe operatora "**typeid**". Zaista, pretpostavimo da imamo još neku klasu naslijeđenu iz klase "DiplomiraniStudent" (recimo klasu "StudentDoktorant") i da pokazivač "s" koji je deklariran kao pokazivač na tip "Student" u nekom trenutku pokazuje na objekat tipa "StudentDoktorant". Pretpostavimo dalje da je potrebno pokazivač "s" pretvoriti u pokazivač na tip "DiplomiraniStudent". Ovakva pretvorba je posve sigurna, s obzirom da objekat tipa "StudentDoktorant" (na koji "s" pokazuje) sadrži sve što sadrže i objekti tipa "DiplomiraniStudent". Stoga će "**dynamic_cast**" uspješno obaviti takvu pretvorbu. S druge strane, tu činjenicu nećemo moći saznati primjenom "**typeid**" operatora, s obzirom da će izraz "**typeid**(*s)" utvrditi činjenicu da "s" pokazuje na objekat tipa "StudentDoktorant" a ne na objekat tipa "DiplomiraniStudent" (tako da nećemo znati da li je pretvorba sigurna). Bez obzira na sve, mada je činjenica 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 naslijeđene klase odnosno adresu nekog objekta naslijeđene klase, obrnuta dodjela *nije moguća*. Tako, pokazivaču na objekat naslijeđene 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 naslijeđene klase ne može prihvatiti 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 naslijeđenu 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 naslijeđenu klasu 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 izbjeći 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 upotrebom polimorfizma. Pretpostavimo da želimo deklarirati nekoliko klasa koje 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 "`DajObim`" i "`DajPovrsinu`", kao 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 klase "`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`" naslijeđenu iz klase "`Pravougaonik`", koja odražava činjenicu da su kvadrati specijalna forma pravougaonika, kao i klase "`PravougliTrougao`" i "`PravilniTrougao`" naslijeđene 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 naslijeđena iz klase "`Pravougaonik`". S jedne strane, neosporna je činjenica da kvadrati *jesu pravougaonici*, tako da je prvi uvjet za svrsihodnost 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 zasada da ove metode za klasu "`Lik`" prosto vraćaju nulu kao rezultat, a u svakoj od naslijeđenih 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:
  string naziv;
public:
  Lik(string naziv) : naziv(naziv) {}
  double DajObim() const { return 0; }
  double DajPovrsinu() const { return 0; }
  void Ispisi() const {
    cout << "Lik: " << naziv << endl << "Obim: " << DajObim() << endl
      << "Površina: " << DajPovrsinu() << endl;
  }
};
```

Sada možemo definirati i klasu "`Krug`", naslijeđenu iz klase "`Lik`". U njoj ćemo deklarirati atribut "`r`" koji čuva poluprečnik kruga, konstruktor koji inicijalizira ovaj atribut, kao i metode "`DajObim`" i

"DajPovrsinu" koje računaju obim i površinu kruga, a koje treba da zamijene istoimene metode iz bazne klase (primijetimo da je u njima je iskorištena konstanta "M_PI" iz biblioteke "cmath", koja predstavlja vrijednost konstante π sa onom preciznošću koliko to tip "**double**" dopušta). Atribut "naziv" i metodu "Ispisi" ova klasa nasljeđuje iz bazne klase "Lik":

```
class Krug : public Lik {
  double r;
public:
  Krug(double r) : Lik("Krug"), r(r) {}
  double DajObim() const { return 2 * M_PI * r; }
  double DajPovrsinu() const { return r * r * M_PI; }
};
```

Na sličan način ćemo definirati i klase "Pravougaonik" i "Trougao" (atributima klase "Trougao" dali smo zaštićeno pravo 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) : Lik("Pravougaonik"), a(a), b(b) {}
  double DajObim() const { return 2 * (a + b); }
  double DajPovrsinu() const { return a * b; }
};

class Trougao : public Lik {
protected:
  double a, b, c;
public:
  Trougao(double a, double b, double c) : Lik("Trougao"), a(a), b(b),
    c(c) {}
  double DajObim() const { return a + b + c; }
  double DajPovrsinu() 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, i promijeniti naziv objekta na "Kvadrat" (s obzirom da će konstruktor klase "Pravougaonik", koji svakako moramo pozvati iz konstruktora klase "Kvadrat", postaviti naziv objekta na "Pravougaonik"). Svi ostali elementi (uključujući i metode za računanje obima i površine) mogu se prosto naslijediti, jer se obim i površina kvadrata mogu računati na isti način kao i za pravougaonik (uz jednake dužine stranica):

```
class Kvadrat : public Pravougaonik {
public:
  Kvadrat(double a) : Pravougaonik(a, a) { 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 pravougaonik 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 atribute 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 klasa "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 kako na efikasnosti, tako i na tačnosti (s obzirom da Heronova formula može akumulirati greške uslijed zaokruživanja):

```
class PravougliTrougao : public Trougao {
public:
  PravougliTrougao(double a, double b)
    : Trougao(a, b, sqrt(a * a + b * b)) {
        naziv = "Pravougli trougao";
  }
  double DajPovrsinu() const { return a * b / 2; }
};

class PravilniTrougao : public Trougao {
public:
  PravilniTrougao(double a) : Trougao(a, a, a) {
    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.DajObim() << " P = " << p.DajPovrsinu() << endl;
k.Ispisi();
cout << "O = " << k.DajObim() << " P = " << k.DajPovrsinu() << endl;
```

Ove naredbe dovele bi do sljedećeg ispisa na ekranu:

Lik: Pravougaonik
Obim: 0
Površina: 0
O = 18 P = 20
Lik: Krug
Obim: 0
Površina: 0
O = 18.849556 P = 28.274334

Vidimo da metode "DajObim" i "DajPovrsinu" 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 naslijeđena još iz klase "Lik", ali je problem u tome što klasa "Lik" sadrži metode "DajObim" i "DajPovrsinu" koje vraćaju nulu kao rezultat, a kompajler je već u fazi prevođenja povezao metodu "Ispisi" sa metodama "DajObim" i "DajPovrsinu" iz iste klase. Stoga metoda "Ispisi" zapravo poziva metode "DajObim" i "DajPovrsinu" 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 "DajObim" i "DajPovrsinu" treba pozvati 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 "DajObim" i "DajPovrsinu" 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 "DajObim" i "DajPovrsinu" iz klase "Krug". Kasno povezivanje je neophodno jer se očigledno u vrijeme prevođenja programa ne može znati nad kojim će objektom metoda "Ispisi" biti pozvana, pa prema tome ni koje metode "DajObim" i "DajPovrsinu" treba da se iz nje pozovu. Prema tome, rješenje je da metode "DajObim" i "DajPovrsinu" u klasi "Lik" proglasimo za *virtualne*, kao u sljedećoj deklaraciji klase "Lik":

```
class Lik {
protected:
  string naziv;
public:
  Lik(string naziv) : naziv(naziv) {}
  virtual ~Lik() {}
  virtual double DajObim() const { return 0; }
  virtual double DajPovrsinu() const { return 0; }
  void Ispisi() const {
    cout << "Lik: " << naziv << endl << "Obim: " << DajObim() << endl
      << "Površina: " << DajPovrsinu() << endl;
  }
};
```

Uz ovakvu izmjenu sve će raditi u skladu sa očekivanjima. Ovdje smo iskoristili priliku i da definiramo virtualni destrtuktor, koji će nam kasnije biti neophodan za realizaciju polimorfizma. Međutim, ovdje je posebno interesantno da se potreba za kasnim povezivanjem i virtualnim metodama javila *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 "DajObim" i "DajPovrsinu", ali kako se radi o virtualnim metodama, unaprijed se ne zna na koje se metode "DajObim" i "DajPovrsinu" (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 naslijeđene 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 usljed 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). Objektno orijentirani programi obavezno se zasnivaju na *hijerarhiji klasa*, odnosno ideji da klase koje dijele neke zajedničke osobine (poput klasa "Krug" i "Pravougaonik" iz prethodnog primjera) trebaju obavezno biti izvedene iz neke osnovne klase koja sadži upravo ono što je zajedničko svim tim klasama (poput klase "Lik" iz prethodnog primjera, koja upravo opisuje činjenicu da je svim krugovima i pravougaonicima zajedničko to što su oni

likovi). Jedino uz poštovanje takve hijerarhijske organizacije možemo iskoristiti sve prednosti koje donosi metodologija objektno orjentiranog programiranja.

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 ostvaren je kroz *prenos funkcija kao parametara u druge funkcije*, odnosno, još generalnije, preko *pokazivača na funkcije*. Zaista, funkcija "NulaFunkcije" koju smo demonstrirali kada smo govorili o prenosu funkcija kao parametara u funkcije, pozivala je funkciju-parametar 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 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, funkcijski parametri i, općenitije, pokazivači na funkcije, također na neki način nude mogućnost da stari dijelovi programa bez ikakve izmjene pozivaju novonapisane dijelove programa, samo što je upotreba virtualnih funkcija jednostavnija. Ovo otkriće ne treba da nas pretjerano čudi. Naime, na kraju ovog predavanja ćemo vidjeti da su virtualne funkcije na neki način zapravo *prerušeni pokazivači na funkcije*, isto kao što su reference prerušeni pokazivači na objekte.

Neko 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 koje su izvedene iz klase "Lik" (tj. na bilo koji lik). Pored toga, kako su metode "DajObim" i "DajPovrsinu" 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;
lik = new Pravougaonik(5, 4);
lik->Ispisi();
cout << "O = " << lik->DajObim() << " P = " << lik->DajPovrsinu()
  << endl;
delete lik;
lik = new Krug(3);
lik->Ispisi();
cout << "O = " << lik->DajObim() << " P = " << lik->DajPovrsinu()
  << endl;
```

U navedenom primjeru promjenljiva "lik" je iskorištena kao polimorfna promjenljiva. Da metode "DajObim" i "DajPovrsinu" 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 kompajler zaključio da treba pozvati metode "DajObim" i "DajPovrsinu" iz klase "Lik" (s obzirom da je promjenljiva "lik" deklarirana kao pokazivač na "Lik"), odnosno odluka koje metode treba pozvati donijela bi se prema *statičkom tipu* promjenljive "lik", a ne njenom dinamičkom tipu. 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 prenositi objekte različitih tipova izvedenih iz klase "Lik", pri čemu bi se uvijek pozivale

odgovarajuće metode 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 (ili barem neka simulacija virtualnih metoda, koja se može postići pomoću pokazivača na funkcije), 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();
```

Strogo rečeno, "`likovi`" nije niz koji stvarno *sadrži objekte različitih tipova*, već niz čiji elementi *pokazuju na objekte različitih tipova*. Međutim, za nas je bitno da ovaj niz možemo koristiti *kao da se radi o nizu koji sadrži objekte različitih tipova*. Zapravo, *dinamički tipovi* elemenata ovog niza se razlikuju od elementa do elementa.

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, ali podrška polimorfizmu je jedan od glavnih razloga zbog kojeg se nasljeđivanje uopće i koristi). 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 izveden 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)! U navedenom primjeru, slučajno ćemo dobiti isti rezultat, s obzirom da bi obje metode "`Povrsina`" trebale dati isti rezultat za pravougli trougao (ako zanemarimo eventualne greške u zaokruživanju), ali takvo ponašanje svakako nije u skladu sa intuicijom.

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 kvadrati jesu pravougaonici, sve operacije koje se mogu primijeniti na pravougaonike ne moraju se nužno moći primijeniti na kvadrate tako da oni ostanu kvadrati. U navedenom primjeru, ne postoje operacije 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 primijenjena 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 naslijeđene 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 neizbježno dovelo do promjene njihove prirode (oni bi prestali biti kvadrati). Slična razmatranja

vrijede za odnose između klase "Trougao" i iz nje naslijeđenih klasa. O ovakvim detaljima treba dobro razmišljati prilikom donošenja odluke o hijerarhijskom dizajnu klasa i odnosa između njih. Teoretičari objektno orjentiranog programiranja ovu činjenicu su ilustrativno objasnili kroz hipotetičko pitanje da li klasa "Noj" treba da bude naslijeđena iz bazne klase "Ptica", s obzirom da noj definitivno *jeste* ptica. Odgovor je vjerovatno *negativan* s obzirom da klasa "Ptica" vjerovatno ima metodu "Leti" koju objekti klase "Noj" ne mogu podržavati. S obzirom da bi se u slučaju nasljeđivanja objekti klase "Noj" mogli koristiti u svim kontekstima u kojima i objekti klase "Ptica", do problema bi sigurno došlo ukoliko bi se objekat klase "Noj" upotrijebio u kontekstu u kojem se poziva metoda "Leti".

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 različitih, 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 primjerci apstraktne bazne klase, niti se primjerci bazne klase kreiraju dinamički pomoću operatora "**new**". Međutim, do sada nas niko nije sprečavao da uradimo tako nešto, npr. da izvršimo sljedeće deklaracije:

```
Lik l("Nešto");
Lik *pok_l = new Lik("Nešto");
```

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 zabrane*. Za tu svrhu dovoljno je u klasi "Lik" neke od virtualnih metoda (najbolje sve) *proglasiti apstraktnim* (to znači da je te metode besmisleno precizno specificirati unutar te klase). Da bismo to uradili, umjesto tijela metode trebamo prosto napisati oznaku "= 0", kao u sljedećoj deklaraciji klase "Lik":

```
class Lik {
protected:
  string naziv;
public:
  Lik(string naziv) : naziv(naziv) {}
  virtual ~Lik() {}
  virtual double DajObim() const = 0;
  virtual double DajPovrsinu() const = 0;
  void Ispis() const {
    cout << "Lik: " << naziv << endl << "Obim: " << DajObim() << endl
      << "Površina: " << DajPovrsinu() << endl;
  }
};
```

Ovom oznakom govorimo da metoda vjerovatno neće biti implementirana niti unutar klase, niti izvan klase, nego tek eventualno u nekoj od klasa koje nasljeđuju klasu "Lik" (mada se apstraktne metode mogu i implementirati, to se radi vrlo rijetko, s obzirom da se one ne mogu tek tako pozvati nad objektom klase, nego samo eventualno iz objekta neke od naslijeđenih klasa). Samo virtualne metode mogu ostati neimplementirane, s obzirom da se one pozivaju mehanizmom kasnog povezivanja (tako da se zapravo nikada neće ni pozvati virtuelna metoda iz bazne klase, nego neka metoda iz neke od naslijeđenih klasa, za koje pretpostavljamo da će biti implementirane). Stoga je jasno da ostavljanje virtualnih metoda neimplementiranim ima smisla samo u apstraktnim baznim klasama (koje će svakako biti naslijeđene). U skladu sa tim, često se uvodi i *formalna definicija apstraktne bazne klase* kao klase koja posjeduje barem jednu apstraktnu 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) kompajler 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 konkretne klase naslijeđene iz nje!

Zanimljivo je razmotriti kreiranje heterogenih, kontejnerskih klasa, odnosno kontejnerskih klasa koje mogu čuvati kolekciju objekata različitog tipa (naročito je interesantno pitanje kako izvesti *konstrukor kopije* i *preklopljeni operator dodjele* u takvim klasama). Pretpostavimo, na primjer, da želimo razviti klasu "`KolekcijaLikova`", koja može sadržavati kolekciju raznih likova, odnosno objekata tipova izvedenih iz apstraktne bazne klase "`Lik`". Kao demonstraciju, prikazaćemo kako bi mogla izgledati izvedba takve klase sa minimalističkim interfejsom, konceptualno slična klasi "`Razred`" koju smo ranije razvijali, a koja čuva kolekciju učenika (tj. objekata tipa "`Ucenik`"):

```
class KolekcijaLikova {
  int broj_likova, kapacitet;
  Lik **likovi;
public:
  KolekcijaLikova(int kapacitet) : broj_likova(0),
    kapacitet(kapacitet), likovi(new Lik*[kapacitet]) {}
  ~KolekcijaLikova();
  KolekcijaLikova(const KolekcijaLikova &k);
  KolekcijaLikova &operator =(const KolekcijaLikova &k);
  void DodajLik(Lik *lik);
  void DodajKrug(double r) { DodajLik(new Krug(r)); }
  void DodajPravougaonik(double a, double b) {
    DodajLik(new Pravougaonik(a, b));
  }
  void DodajTrougao(double a, double b, double c) {
    DodajLik(new Trougao(a, b, c));
  }
  void IspisiKolekciju() const;
};
```

Klasa "`KolekcijaLikova`" interno čuva dvojni pokazivač pomoću kojeg se pristupa dinamički alociranom nizu koji čuva pokazivače na objekte tipa "`Lik`" (jasno je da je interno čuvanje pokazivača umjesto samih objekata neophodno za realizaciju polimorfizma). Kao i obično atributi "`broj_likova`" i "`kapacitet`" predstavljaju respektivno broj likova u kolekciji i maksimalan broj likova koji kolekcija može primiti, a koji se zadaje putem konstruktora. Implementacija konstruktora je trivijalna, stoga razmotrimo implementaciju destruktora. Ukoliko pretpostavimo da ćemo klasi "`KolekcijaLikova`" povjeriti vlasništvo nad svim likovima koji su u njoj pohranjeni, ona je tada odgovorna i za njihovo brisanje, tako da bi destruktor mogao izgledati ovako:

```
KolekcijaLikova::~KolekcijaLikova() {
  for(int i = 0; i < broj_likova; i++) delete likovi[i];
  delete[] likovi;
}
```

Konstruktor kopije i preklopljeni operator dodjele su najproblematičniji dio ove klase, tako da ćemo njih razmotriti na kraju. Što se tiče metode "`DodajLik`", ona prosto prima pokazivač na lik koji želimo dodati u kolekciju i upisuje ga u kolekciju, osim ukoliko je ona popunjena (kako je brisanje likova povjereno samoj kolekciji, jasno je da pokazivači koje prosljeđujemo ovoj metodi moraju pokazivati na dinamički kreirane objekte, inače će brisanje uzrokovati probleme):

```
void KolekcijaLikova::DodajLik(Lik *lik) {
  if(broj_likova >= kapacitet) throw "Kolekcija popunjena!\n";
  likovi[broj_likova++] = lik;
}
```

Pored ove univerzalne metode za dodavanje novih likova u kolekciju, predviđene su i metode "DodajKrug", "DodajPravougaonik" i "DodajTrougao" koje na osnovu zadanih parametara "u hodu" dinamički kreiraju odgovarajući lik i upisuju ga u kolekciju (radi jednostavnosti, nismo predvidjeli metode poput "DodajKvadrat", koje je lako dodati u slučaju potrebe). Posljednja predviđena metoda u ovoj minimalističkoj klasi je metoda "IspisiKolekciju" koja ispisuje podatke o svim likovima u kolekciji, prostim pozivom metode "Ispisi" nad svakim objektom u kolekciji:

```
void KolekcijaLikova::IspisiKolekciju() const {
  for(int i = 0; i < broj_likova; i++) likovi[i]->Ispisi();
}
```

U slučaju potrebe, lako je ovu klasu proširiti novim metodama. Međutim, razmotrimo sada kako bismo izveli konstruktor kopije. Zadovoljimo li se plitkim kopijama, dovoljno bi bilo uvesti brojanje pristupa na način kako smo to već ranije opisali. Problemi nastaju ukoliko želimo da konstruktor kopije zaista formira potpunu kopiju čitave kolekcije. Mada je problem konceptualno sličan problemu kopiranja objekata tipa "Razred", ovdje dodatni problemi nastaju zbog činjenice da pokazivači unutar niza "likovi" mogu pokazivati na objekte *različitih tipova*. Zbog toga, sljedeće prepisivanje rješenja koje je radilo za klasu "Razred" ovdje *neće raditi*:

```
KolekcijaLikova::KolekcijaLikova(const KolekcijaLikova &k) :
  likovi(new Lik*[k.kapacitet]), kapacitet(k.kapacitet),
  broj_likova(k.broj_likova) {
    for(int i = 0; i < broj_likova; i++)
      likovi[i] = new Lik(*k.likovi[i]);
}
```

Zaista, mi ne možemo kreirati objekte tipa "Lik", a i da možemo, to nije ono što nam treba, s obzirom da pokazivači ne pokazuju na objekte tipa "Lik" nego na objekte nekog tipa izvedenog iz tipa "Lik". Neko bi mogao doći na ideju da iskoristi operator "**typeid**" za određivanje kojeg je zaista tipa objekat na koji razmatrani pokazivač pokazuje i da u skladu sa tim konstruira odgovarajući objekat koji je njegova kopija. Mada takvo rješenje radi, ono je vrlo rogobatno. Što je još gore, svaka eventualna dopuna hijerarhije likova nekim novim likovima (recimo, elipsama ili petouglovima) zahtijevala bi odgovarajuće dopune u konstruktoru kopije. Slijedi prikaz kako bi takvo rješenje moglo izgledati:

```
KolekcijaLikova::KolekcijaLikova(const KolekcijaLikova &k) :
  likovi(new Lik*[k.kapacitet]), kapacitet(k.kapacitet),
  broj_likova(k.broj_likova) {
    for(int i = 0; i < broj_likova; i++)
      if(typeid(*k.likovi[i]) == typeid(Krug))
        likovi[i] = new Krug(*(Krug*)k.likovi[i]);
      else if(typeid(*k.likovi[i]) == typeid(Pravougaonik))
        likovi[i] = new Pravougaonik(*(Pravougaonik*)k.likovi[i]);
      else if(typeid(*k.likovi[i]) == typeid(Trougao))
        ...
}
```

Ovako bi trebalo nastaviti za sve moguće tipove likova. Međutim, rješenja koja koriste "**typeid**" operator gotovo nikada nisu dobra rješenja (osim u rijetkim izuzecima) i tipično ukazuju da se problem može mnogo elegantnije riješiti *upotrebom virtualnih funkcija*. Tako se najčešće korišteno rješenje ovog problema izvodi dodavanjem jedne nove čisto virtuelne metode u apstraktnu baznu klasu koju možemo nazvati "DajKopiju" (u engleskoj literaturi ova metoda se tipično naziva *clone*):

```
class Lik {
  ...
  virtual Lik *DajKopiju() const = 0;
};
```

Uloga ove metode je da, kada se pozove nad nekim objektom, kreira identičnu kopiju tog objekta i vrati pokazivač na kreiranu kopiju. Naravno, ovu metodu treba dodati i implementirati u svakoj od klasa koje su naslijeđene iz apstraktne bazne klase. Srećom, te implementacije su veoma jednostavne, s

obzirom da svaki objekat vrlo lako može kreirati svoju kopiju pozivom svog vlastitog konstruktora kopije prilikom dinamičke alokacije. Recimo, pokazaćemo kako bi se ova metoda implementirala u klasama "Krug" i "Pravougaonik" (za ostale klase vrijedila bi analogna implementacija):

```
class Krug {
  ...
  virtual Lik *DajKopiju() const { return new Krug(*this); };
};
class Pravougaonik {
  ...
  virtual Lik *DajKopiju() const { return new Pravougaonik(*this); };
};
```

Ovim implementacija konstruktora kopije postaje trivijalna i, što je mnogo važnije, potpuno neovisna od toga kakvih uopće objekata imamo u hijerarhiji objekata izvedenih iz bazne klase:

```
KolekcijaLikova::KolekcijaLikova(const KolekcijaLikova &k) :
  likovi(new Lik*[k.kapacitet]), kapacitet(k.kapacitet),
  broj_likova(k.broj_likova) {
    for(int i = 0; i < broj_likova; i++)
      likovi[i] = k.likovi[i]->DajKopiju();
}
```

Jedina mana ovog rješenja je što sve klase u hijerarhiji moraju definirati metodu "DajKopiju". Postoje doduše neki trikovi zasnovani na generičkim mehanizmima kojima je ovo moguće izbjeći i automatizirati čitav proces, ali radi se o prilično naprednim tehnikama koje izlaze izvan okvira ovog kursa. Ovdje prikazano "školsko" rješenje uglavnom zadovoljava sve praktične potrebe.

Što se tiče preklopljenog operatora dodjele, on se može realizirati na analogan način, samo što je prilikom dodjele prethodno potrebno prvo uništiti sve objekte koji su bili u vlasništvu kolekcije kojoj se vrši dodjela (na način kao u destruktoru). Naravno, treba paziti da ne dođe do destruktivne samododjele. Implementaciju operatora dodjele možete uraditi sami kao korisnu vježbu, a kompletna implementacija razmatrane hijerarhije likova i klase "KolekcijaLikova" može se naći na web stranici kursa pod imenom "likovi.cpp".

Za kraj ostaje još da objasnimo kako zapravo radi mehanizam pozivanja virtualnih funkcija. Kao što smo već rekli, virtualne funkcije su zapravo prerušeni 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 naslijeđenih klasa koje modificiraju 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;
  int indeks;
  void (*pok_na_fn_za_ispis)(const Student *s);
  static void PomocnaZaIspis(const Student *s);
public:
  Student(string ime, int ind) : ime(ime), indeks(ind),
    pok_na_fn_za_ispis(PomocnaZaIspis) {}
  string DajIme() const { return ime; }
```

```
  int DajIndeks() const { return indeks; }
  void Ispisi() const { pok_na_fn_za_ispis(this); }
};
class DiplomiraniStudent : public Student {
  int godina_diplomiranja;
  static void PomocnaZaIspis(const Student *s);
public:
  DiplomiraniStudent(string ime, int ind, int god_dipl)
    : Student(ime, ind), godina_diplomiranja(god_dipl)
        { pok_na_fn_za_ispis = PomocnaZaIspis; }
  int DajGodinuDiplomiranja() const { return godina_diplomiranja; }
};
void Student::PomocnaZaIspis(const Student *s) {
  cout << "Student " << s->ime << " ima indeks " << s->indeks;
}
void DiplomiraniStudent::PomocnaZaIspis(const Student *s) {
  Student::PomocnaZaIspis(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. Razumjevanje ovog primjera 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. U suštini, kad koristimo virtualne funkcije, računar radi slične iste stvari kao u ovom primjeru, samo što zavrzlame oko upotrebe pokazivača na funkcije i njihove pravilne inicijalizacije umjesto nas obavlja sam kompajler, čime nas pošteđuje mnogih muka.

Strogo rečeno, mehanizam virtualnih funkcija zapravo uvodi *još jedan nivo indirekcije*. Naime, ukoliko postoji mnogo virtualnih metoda u nekoj klasi, neracionalno je u svakom primjerku te klase čuvati pokazivač za svaku od tih virtualnih metoda, s obzirom da svi primjerci iste klase dijele iste virtualne metode. Zbog toga se svi pokazivači na konkretne izvedbe virtualnih metoda neke klase čuvaju *izvan klase* u jednom nizu (tabeli) pokazivača koji se zove *tabela virtualnih metoda* (skraćeno *TVM*) koju automatski kreira kompajler, dok sami primjerci klasa sadrže samo pokazivač na tabelu virtualnih metoda za tu klasu. Prilikom poziva neke virtuelne metode, prvo se vrši dereferenciranje tog pokazivača da se izvrši pristup tabeli virtualnih metoda, nakon čega se u tabeli pronalazi odgovarajući pokazivač i indirektno poziva tražena metoda preko tog pokazivača. Zbog ove dvostruke indirekcije, pozivanje virtualnih metoda je nešto sporije nego kod metoda koje nisu virtualne. Ovo usporenje se posebno primijeti kod metoda sa kratkim tijelom čije je vrijeme izvršavanja zanemarljivo, tim prije što se virtuelne funkcije ne mogu *umetati* u prevedeni kôd (kao "**`inline`**" funkcije), nego se uvijek pozivaju (i to uz dvije dopunske indirekcije). Ovo je, pored nekih drugih manje bitnih razloga, glavni razlog zbog kojeg se sve metode ne tretiraju automatski kao virtualne, kao što je slučaj u čisto objektno orjentiranim jezicima, kao što je recimo Java. Drugi razlog je što je moguće konstruisati takve primjere nasljeđivanja u kojima se objekti izvedene klase u nekim kontekstima *neće ispravno ponašati* ukoliko su metode bazne klase deklarirane kao virtualne, mada svi takvi primjeri u manjoj ili većoj mjeri *narušavaju pravila* kada uopće treba koristiti nasljeđivanje, o kojima smo govorili na početku (tako da oni i nisu baš u duhu objektno orjentiranog programiranja).

Predavanje 13.

Svi programi koje smo do sada pisali posjedovali su suštinski nedostatak da se svi uneseni podaci gube onog trenutka kada se program završi. Poznato je da se ovaj problem rješava uvođenjem *datoteka* (engl. *files*), koje 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).

Poznato je da jezik C također podržava rad sa datotekama (koje inače mora podržavati svaki programski jezik koji sebe smatra upotrebljivim). Podsjetimo se da se u jeziku C rad sa datotekama zasniva na primjeni izvjesnih funkcija iz biblioteke sa zaglavljem "`stdio.h`" 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*. Radi kompatibilnosti sa jezikom C, ovaj način rada sa datotekama radi i u jeziku C++, samo je umjesto zaglavlja "`stdio.h`" potrebno koristiti zaglavlje "`cstdio`". Međutim, takav način rada sa datotekama ne uklapa se konceptualno u filozofiju objektno zasnovanog i objektno orjentiranog programiranja i može ponekad praviti probleme u kombinaciji sa objektno zasnovanim tehnikama. Zbog toga je u jeziku C++ uveden novi način rada sa datotekama, koji je ujedno i mnogo jednostavniji i fleksibilniji. Ovaj način rada zasnovan je na ulaznim i izlaznim tokovima, na način koji je veoma sličan načinu korištenja objekata ulaznog i izlaznog toka "`cin`" i "`cout`". U nastavku ćemo detaljnije razmotriti upravo taj način rada sa datotekama.

Već smo rekli da su "`cin`" i "`cout`" zapravo instance klasa 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 klasa 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 parametrom tipa znakovnog niza koji predstavlja ime datoteke sa kojom se tok povezuje. Pored toga, klasa "`ifstream`" je naslijeđena iz klase "`istream`", a klasa "`ofstream`" iz klase "`ostream`", tako da instance klasa "`ifstream`" i "`ofstream`" posjeduju sve metode i operatore koje posjeduju instance klasa "`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 iz 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 datoteka ne smiju sadržavati neki od znakova "\", "/", ":", "*", "?", "<", ">", "|", kao ni znak navoda. Također, izrazito je nepreporučljivo korištenje znakova izvan engleskog alfabeta 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 alfabeta 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 recimo 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 uređaja 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");
```

Podsjetimo 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`" tipa niza znakova, možemo kao stvarni parametar navesti bilo koji niz znakova, a ne samo stringovnu konstantu (tj. niz znakova između znakova navoda). 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 prihvatiti *dinamički string*, odnosno objekat tipa "`string`". Međutim, ranije smo govorili da se objekti tipa "`string`" uvijek mogu pretvoriti u klasične nul-terminirane nizove znakova primjenom metode "`c_str`" na stringovni objekat. 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 datoteka 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 (tzv. *kurzor*) 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 iščitati 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 iščitavanje podataka vrši "**for**" petljom, pri čemu unaprijed moramo znati koliko podataka sadrži datoteka. U praksi se obično javlja potreba za čitanjem datoteka 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 pretpostavlja 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), pomoću 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 neistinu (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 usljed 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 usljed 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 (ista primjedba vrijedi i za tzv. *binarne datoteke*, koje će biti opisane nešto kasnije). 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!

Ulazni 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 govora). Ovo je ujedno jedan od načina kako započeti čitanje datoteke ispočetka. Klase "`ifstream`" i "`ofstream`" također posjeduju i konstruktore 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 destruktor 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 (recimo u C-u), niti se moramo brinuti o tome šta će se desiti ukoliko dok je tok otvoren dođe do bacanja izuzetka (što je poseban problem u mnogim 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 klasa "`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`" (podsjetimo se da se binarna disjunkcija dobija 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 većini implementacija standardne biblioteke jezika C++) u slučaju da ulazni tok dospije u neispravno stanje (npr. zbog pokušaja čitanja iza kraja datoteke). Ovu osobinu smo iskoristili da skratimo program. S obzirom da se i sam izvorni program pisan u jeziku C++ također na disku čuva kao obična tekstualna datoteka, možemo prethodni program snimiti na disk recimo 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 klasa "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 novouneseni 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 novouneseni 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 dovešće 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`" mogle bi izgledati ovako:

STUDENTI.TXT:	OCJENE.TXT:
Pero Perić	1234 7
1234	4132 8
Huso Husić	1234 6
4132	2341 9
Marko Marković	1234 10
3142	4132 9
Meho Mehić	1234 8
2341	

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;
    studenti.getline(ime, sizeof ime);
    if(!studenti) break;
    studenti >> indeks;
    studenti.ignore(10000, '\n');
    ifstream ocjene("OCJENE.TXT");
    int suma_ocjena(0), broj_ocjena(0), tekuci_indeks, tekuca_ocjena;
    while(ocjene >> tekuci_indeks >> tekuca_ocjena)
      if(tekuci_indeks == indeks) {
        suma_ocjena += tekuca_ocjena;
        broj_ocjena++;
      }
    cout << ime << " ";
    if(suma_ocjena == 0) cout << "NEOCIJENJEN";
    else
      cout << setprecision(3) << double(suma_ocjena) / broj_ocjena;
    cout << 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 uzmemo 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 cjelobrojnoj promjenljivoj, ona će (uz pretpostavku da cjelobrojne promjenljive zauzimaju 32 bita) biti predstavljena kao odgovarajući binarni broj, tj. broj 00000010000101111111010001011110. 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, mnogi programski jezici, u koje spadaju i jezici C i C++, podržavaju 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 datoteka 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 tekstualnim 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). S druge strane, moramo imati u vidu da binarne datoteke ne možemo kreirati i pregledati 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. Ovo na prvi pogled nezgodno svojstvo 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.

Potrebno je još znati i da su binarne datoteke podložne velikim problemima *po pitanju prenosivosti*. Naime, zbog činjenice da tačan oblik zapisa podataka u računarskoj memoriji nije propisan standardima jezika nego je ostavljen implementatorima kompajlera na volju, može biti problematično pomoću nekog programa koji je kompajliran sa jednom verzijom kompajlera učitati neku binarnu datoteku koja je kreirana pomoću nekog programa koji je kompajliran sa drugom verzijom kompajlera (moguće čak i na drugom modelu računara). Zaista, sasvim je moguće da dva različita kompajlera ne koriste recimo isti broj bita za isti tip podataka (recimo, jedan kompajler može čuvati cijele brojeve u 32 bita, a drugi u 64

bita). Čak i kada se koristi isti broj bita za reprezentaciju nekog tipa podataka, sami biti mogu biti drugačije organizirani. Na primjer, kada se 32-bitni broj razlaže na 4 bajta, ti bajtovi na jednoj verziji kompajlera mogu biti poredani tako da bajtovi manje težine dolaze prije bajtova veće težine (tzv. *little endian* zapis), dok na drugoj verziji kompajlera može biti obrnuto (tzv. *big endian* zapis). U slučajevima kada *tačno znamo* kako su organizirani podaci pohranjeni u datoteci, načelno je (uz dosta muke) moguće pročitati sadržaj datoteke čak i kada se ta organizacija ne slaže sa načinom kako podatke organizira kompajler koji koristimo za kreiranje programa koji čita datoteku (naravno, pod uvjetom da znamo i tu organizaciju). Recimo, moguće je pročitati sadržaj datoteke *bajt po bajt* i ručno "presložiti" bajtove da se dobije potrebna organizacija. Međutim, za tako nešto potrebno je mnogo znanja i vještine. Stoga, kada god je potrebno ostvariti veliku prenosivost datoteka (pogotovo ukoliko se datoteka kreirana na jednom modelu računara treba pročitati na nekom drugom modelu računara), najpametnije je potpuno zaobići binarne datoteke i koristiti isključivo tekstualne datoteke.

Za rad sa binarnim datotekama namijenjene su metode "`read`" i "`write`" (koje su, zapravo, objektno orijentirani pandani funkcija "`fread`" i "`fwrite`" naslijeđenih iz jezika C). 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. Operator "**`sizeof`**" kao svoj parametar zahtijeva neki izraz ili ime tipa, i kao rezultat daje broj bajtova koje zauzima vrijednost tog izraza odnosno taj tip (u slučaju kada se kao parametar koristi ime tipa, parametar mora biti u zagradama, dok u drugim slučajevima zagrade nisu neophodne). Međutim, 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 (ovo je motivirano činjenicom da se jedino nizovi znakova mogu sigurno pohranjivati u binarne datoteke bez problema vezanih za prenosivost, s obzirom da jedan znak uvijek zauzima jedan bajt, bez obzira na korišteni kompajler i računarsku arhitekturu). Zbog toga će u praksi često biti potrebna konverzija stvarnog pokazivača koji sadrži adresu nekog objekta u memoriji u pokazivač na tip "**`char`**" primjenom operatora za konverziju tipova. Pri tome se, za konverziju pokazivača između *potpuno nesaglasnih tipova* ne može koristiti operator "**`static_cast`**" (koji zabranjuje konverzije pokazivača na neki tip u pokazivač na posve nekompatibilan tip), već se mora koristiti ili pretvorba tipova u stilu jezika C, ili operator "**`reinterpret_cast`**" koji je namijenjen upravo za međusobne konverzije između pokazivačkih tipova na međusobno nekompatibilne tipove (inače, upotreba operatora "**`reinterpret_cast`**" u ma kakvom kontekstu gotovo sigurno ukazuje da ćemo imati problema sa prenosivošću). Dakle, za konverziju nekog pokazivača "`p`" na neki proizvoljan tip u tip pokazivača na znakove možemo koristiti ili konstrukciju "`(char *)p`" naslijeđenu iz jezika C, ili nešto rogobatniju konstrukciju "`reinterpret_cast<char *>(p)`". Radi jednostavnosti, u nastavku ćemo koristiti prvu varijantu.

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 jednom potezu 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 izmijeni 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 operativim 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 ili Linux, opcija "`ios::binary`" nema nikakvog efekta i prosto se ignorira, dok kod drugih operativnih sistema (prvenstveno operativnih sistema iz MS DOS ili MS Windows serije) izostavljanje ove opcije može ponekad dovesti do neželjenih efekata pri radu sa binarnim datotekama. Stoga je veoma 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;
}
```

Primijetimo 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 promijenimo tip elemenata niza "niz".

Izuzetno je važno napomenuti da je u binarne datoteke dozvoljeno pohranjivati samo POD (Plain Old Data) tipove podataka. Recimo, nije dozvoljeno u binarnu datoteku pohraniti niz čiji su elementi tipa "string" (pa čak ni jedan objekat tipa "string"). Pokušaj da tako nešto uradimo dovešće do katastrofalnog gubitka podataka, dok pokušaj učitavanja iz takve datoteke može čak dovesti i do kraha. Također, potpuno je nesvrsihodno u binarnu datoteku pohranjivati *pokazivače*, s obzirom da pohranjivanje samog pokazivača neće ujedno sačuvati i objekat na koji on pokazuje (ovo ujedno i objašnjava zbog čega je besmisleno u binarnu datoteku pohranjivati podatke koji ne pripadaju POD tipovima podataka, s obzirom da se rad takvih tipova podataka gotovo uvijek interno zasniva na pokazivačima). Ukoliko niste sigurni šta jesu a šta nisu POD tipovi podataka, činjenica je da svi tipovi podataka naslijeđeni iz jezika C++ jesu POD tipovi podataka, dok tipovi podataka uvedeni u jeziku C++ uglavnom nisu POD tipovi podataka, mada ima i izuzetaka.

Kao opći zaključak, možemo istaći da da 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 predstavljali 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 zadanu 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 izuzetno 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 sekvencijalno (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)) {
    broj *= 2;
    datoteka.seekp(-int(sizeof broj), ios::cur);
```

```
    datoteka.write((char*)&broj, sizeof broj);
    datoteka.seekg(0, ios::cur);
  }
  return 0;
}
```

U ovom programu, metoda "seekp" je iskorištena da pomjeri kurzor za pisanje tako da se element koji se upisuje *prepiše* preko upravo pročitanog elementa. Neočekivana i na prvi pogled suvišna konverzija rezultata "**sizeof**" operatora u tip "**int**" neophodna je zbog činjenice da je tip rezultata koji vraća operator "**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). Također, razlog za prividno nepotrebno pozivanje metode "seekg" na kraju petlje postaće jasan iz izlaganja koje slijedi u nastavku.

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* (što mnogima nije poznato, s obzirom da se ova činjenica rijetko navodi u literaturi). Sa čitanja na pisanje smijemo preći samo nakon što *eksplicitno pozicioniramo kurzor za pisanje* pozivom metode "seekp" (što smo i radili u prethodnom primjeru). Analogno, sa pisanja na čitanje smijemo preći jedino nakon što *eksplicitno pozicioniramo kurzor za čitanje* pozivom metode "seekg", pa makar ne izvršili nikakvo pomjeranje kurzora (upravo je ovo razlog za prividno suvišno pozivanje metode "seekg" u prethodnom primjeru). Pored toga, sa pisanja na čitanje smijemo preći i u trenutku kada je spremnik izlaznog toka ispražnjen, 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*, bez njenog prethodnog 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. Ovo možete probati izvesti 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 klasa u datoteke, 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). Međutim, treba naglasiti da se na ovaj način smiju smještati samo sadržaji slogova ili instanci klasa koje predstavljaju POD podatke, što znači da svi atributi takvih struktura ili klasa također moraju biti POD podaci (što isključuje strukture ili klase koje sadrže atribute tipa "`string`" i slične). Također, ukoliko se radi o instancama klase, one ne smiju sadržavati virtuelne funkcije. Sam princip pohranjivanja je ilustriran 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 <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;
  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 predvidjeti 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 interno zasnovane na pokazivačima (kao što su "`string`" ili "`vector`"), implementacija ovih metoda je trivijalna. Na primjer, pretpostavimo 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(Student));
  if(!izlaz) throw "Nešto nije u redu sa upisom!\n";
}
```

Obnavljanje sadržaja iz datoteke je donekle komplikovanije. Na prvom mjestu, 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. Jedan mogući tretman ovog problema je da *ignoriramo* vrijednost pokazivača "`studenti`" sačuvanog u datoteci, a da umjesto njega koristimo aktuelnu vrijednost pokazivača "`studenti`" koja pokazuje na aktuelno alocirani prostor u koji treba učitati obnovljeni sadržaj (pri tome bi aktuelnu vrijednost pokazivača "`studenti`" trebalo sačuvati u pomoćnoj promjenljivoj, s obzirom da će ona biti "ubrljana" čitanjem sadržaja klase iz datoteke). Drugi problem je što postoji mogućnost da sadržaj nekog primjerka klase "`StudentskaSluzba`" izvjesnog kapaciteta probamo obnoviti iz datoteke u koju je snimljen sadržaj nekog drugog primjerka klase "`StudentskaSluzba`" većeg kapaciteta. Ukoliko dinamički alocirani niz u objektu čiji sadržaj želimo da obnovimo nema dovoljnu veličinu da se u njega može učitati čitav snimljeni niz, doći če do kraha programa. Stoga, ukoliko je kapacitet objekta čiji sadržaj obnavljamo manji u odnosu na kapacitet objekta koji je snimljen u datoteku, potrebno je izvršiti realokaciju memorije (tj. povećati kapacitet razmatranog objekta). Zapravo, kako je obnavljanje sadržaja objekta iz datoteke operacija koja se izvodi prilično rijetko, nema ništa loše u tome da realokaciju obavljamo uvijek, tj. da prethodno uništimo kompletan sadržaj objekta a da nakon toga izvršimo njegovu obnovu na osnovu aktualnih podataka u datoteci. 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";
  delete[] studenti;
  ulaz.read((char*)this, sizeof *this);
  studenti = new Student[MaxBrojStudenata];
  ulaz.read((char*)studenti, broj_studenata * sizeof(Student));
  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*. Pretpostavimo, 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:

```
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(Student));
  if(!izlaz) throw "Nešto nije u redu sa upisom!\n";
}
```

Za pisanje metode "`Obnovi`" možemo iskoristiti sličnu logiku. Radi jednostavnosti, realokaciju memorije vršimo 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*[MaxBrojStudenata];
  for(int i = 0; i < broj_studenata; i++) {
    studenti[i] = new Student;
    ulaz.read((char*)studenti[i], sizeof(Student));
  }
  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 memorijskog prostora (pri čemu se adresa alociranog prostora dodijeljuje 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.

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 je mnogo pametnije 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 izvedba takvog konstruktora za posljednji, najsloženiji primjer klase "`StudentskaSluzba`" u kojoj se koristi dvojni pokazivač (odgovarajući konstruktori za ostale primjere mogu se napisati analogno):

```
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*[MaxBrojStudenata];
  for(int i = 0; i < broj_studenata; i++) {
    studenti[i] = new Student;
    ulaz.read((char*)studenti[i], sizeof(Student));
  }
  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`" proslijediti fiktivne parametre, bez obzira što bi se sam sadržaj objekta tipa "`Student`" kasnije pročitao iz datoteke. Da izbjegnemo ovu neeleganciju, 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";
}
```

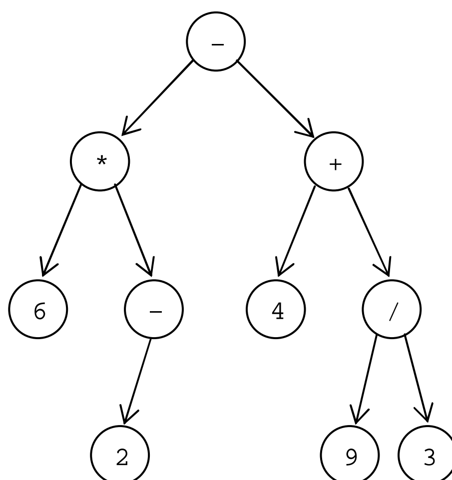
Ovaj konstruktor bi bilo prirodno deklarirati u privatnoj sekciji klase "`Student`", čime bismo spriječili da obični korisnici klase "`Student`" mogu kreirati objekte klase "`Student`" pomoću ovog konstruktora, koji je očigledno čisto pomoćne prirode. Pored toga bi klasu "`StudentskaSluzba`" trebalo 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 se shvati kako bi se moglo obaviti snimanje u datoteku ili obnavljanje iz datoteke čak i u slučaju klasa koje koriste prilično složenu dinamičku alokaciju memorije. Kao veoma korisnu vježbu, možete pokušati proširiti klasu "`Matrica`" koja je razvijena na ranijim predavanjima metodama koje snimaju sadržaj matrice u binarnu datoteku, odnosno obnavljaju sadržaj matrice iz binarne datoteke, s obzirom da ova klasa koristi dosta kompliciran mehanizam dinamičke alokacije memorije (složeniji od svih primjera razmotrenih na ovom predavanju).

Predavanje 14.

Na ranijim predavanjima je istaknuto da objektno orjentirano programiranje predstavlja metodologiju programiranja koja se oslanja na četiri načela: *sakrivanju informacija*, *enkapsulaciji*, *nasljeđivanju* i *polimorfizmu*. Ova metodologija se pokazuje naročito korisnom pri razvoju velikih programa, pogotovo onih koje razvija tim neovisnih programera i koji su podložni čestim izmjenama i dopunama. Prostor nam ne dozvoljava da možemo ilustrirati objektno orjentiranu metodologiju programiranja na nekom zaista velikom programu kod kojeg ove metode dolaze do punog izražaja. Umjesto toga, ovdje će biti prikazan razvoj jednog posve malog demonstracionog programa, ali koji pruža jasan uvid u sve osnovne ideje koje stoje iza objektno orjentiranog programiranja. Bez obzira na svoju kratkoću, prikazani program je posve upotrebljiv kao dio raznih složenijih programa. Ono što je posebno interesantno je što problem neće odmah biti postavljen u cijelosti. Na početku će prvo biti iznijeti minimalistički zahtjevi, koji će se proširivati novim zahtjevima kako postojeći zahtjevi budu realizirani. Bez obzira na takve dopune zahtjeva tokom razvoja, niti jedan novi zahtjev neće zahtijevati izmjenu niti jedne linije do tada napisanog koda! Bez objektno orjentirane metodologije programiranja, takvu fleksibilnost je gotovo nemoguće postići.

Program koji ćemo razvijati manipuliraće sa *stablima aritmetičkih izraza*. Stablo aritmetičkih izraza je struktura podataka koja opisuje aritmetičke izraze na prirodan način, koji omogućava njihovu jednostavnu interpretaciju i efikasno manipuliranje sa njima (npr. algebarske transformacije). Stablo aritmetičkih izraza sadrži različite tipove čvorova. Jedna vrsta čvorova odgovara operandima poput konstanti ili promjenljivih, koji se ne mogu razbiti na prostije dijelove. Takve operande nazivaćemo *atomi*. Druga vrsta čvorova odgovara operatorima i oni imaju svoje potomke. Ovdje opet možemo razlikovati čvorove koji odgovaraju unarnim operatorima i koji imaju jednog potomka (koji odgovara operandu operatora), te čvorove koji odgovaraju binarnim operatorima i koji imaju dva potomka (po jedan za svaki od operanada). Recimo, aritmetičkom izrazu "`6*(-2)-(4+9/3)`" odgovara stablo kao na sljedećoj slici:



Potreba za manipulacijom sa stablima aritmetičkih izraza javlja se u brojnim primjenama, poput kompajlera, kalkulatora, ili programa za simboličku matematiku. Cilj nam je razviti tip podataka, nazovimo ga "`Izraz`", koji će omogućiti rad sa stablima aritmetičkih izraza. Za početak ćemo postaviti posve jednostavne zahtjeve. Na prvom mjestu, želimo obezbijediti način za kreiranje objekata tipa "`Izraz`". Za tu svrhu služe konstruktori, pa ćemo predvidjeti tri konstruktora. Prvi konstruktor, sa samo jednim parametrom, kreiraće atomarne izraze. Da ne bismo previše ulazili u nepotrebne detalje, ograničimo se da atomarni izrazi uvijek predstavljaju realne brojeve (tj. objekte tipa "**`double`**"). Drugi konstruktor, sa dva parametra, kreiraće izraze u čijem se korijenu nalazi neki unarni operator, a njegovi parametri su oznaka operatora i operand (koji može ponovo biti tipa "`Izraz`"). Slično, konstruktor sa tri parametra kreiraće izraze u čijem se korijenu nalazi neki binarni operator, a njegovi parametri će biti oznaka operatora i dva operanda (operator ćemo zadavati između operanada). Dopustimo li da se oznaka

operatora može sastojati od više znakova (kao, recimo, kod operatora "<="), prirodan tip parametara koji opisuju operator je tip "string". Stoga, dio interfejsa klase koji odgovara konstruktorima trebao bi izgledati ovako (s obzirom da je "**operator**" ključna riječ, parametar koji opisuje operator nazvaćemo "operacija"):

```
Izraz(double atom);
Izraz(string operacija, const Izraz &operand);
Izraz(const Izraz &operand_1, string operacija, const Izraz &operand_2);
```

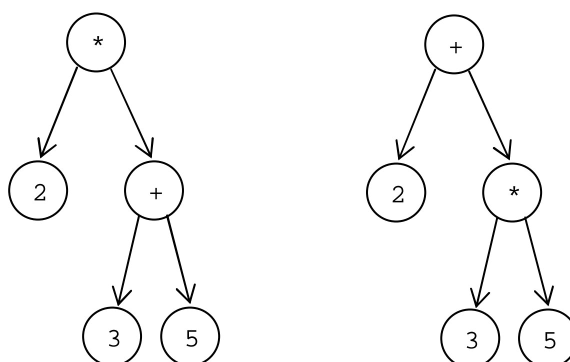
Uz ovako definirane konstruktore, primjer naredbe koja bi kreirala objekat "e" tipa "Izraz" koji odgovara stablu aritmetičkog izraza iz prethodnog primjera, mogao bi izgledati recimo ovako:

```
Izraz e(Izraz(6, "*", Izraz("-", 2)), "-", Izraz(4, "+",
  Izraz(9, "/", 3)));
```

Ovdje je očigledno iskorištena osobina da se konstruktor može pozivati i kao funkcija. Također, primijetimo da se kao drugi i treći parametri konstruktoru klase "Izraz" mogu prosljeđivati i brojevi, zahvaljujući neeksplicitnom konstruktoru sa jednim parametrom koji prima parametar tipa "**double**" i koji se, prema tome, koristi za automatsku konverziju objekata tipa "**double**" u atomarne izraze. Stoga se prethodna naredba interpretira kao sljedeća, znatno rogobatnija i nečitljivija naredba:

```
Izraz e(Izraz(Izraz(6), "*", Izraz("-", Izraz(2))), "-",
  Izraz(Izraz(4), "+", Izraz(Izraz(9), "/", Izraz(3))));
```

Drugi zahtjev koji ćemo postaviti na klasu "Izraz" je da se objekti tipa "Izraz" mogu *ispisivati*, u formi koja je uobičajena za aritmetičke izraze. Nažalost, da bi se tačno odredila pozicija zagrada u aritmetičkim izrazima, potrebno je uvoditi konvencije o prioritetu pojedinih operacija. Na primjer, pogledajmo dva stabla aritmetičkih izraza na sljedećoj slici, koja su veoma slična po izgledu. Stablu sa lijeve strane odgovara izraz "2*(3+5)", a stablu sa desne strane izraz "2+3*5". Međutim, u prvom slučaju zagrade su neophodne, a u drugom nisu, jer množenje ima prioritet u odnosu na sabiranje.



Da ne bismo ulazili u nepotrebne komplikacije uzrokovane dogovorom nametnutim konvencijama o prioritetu pojedinih operatora, dogovorimo se da prosto sve izraze koji nisu atomarni pišemo *unutar zagrada*, čime ćemo svakako osigurati ispravan redoslijed operacija, po cijenu prisustva suvišnih zagrada, što nije osobit problem. Recimo, uz takvu konvenciju, izrazi koji odgovaraju stablima sa prethodne slike, pisali bi se kao "(2*(3+5))" odnosno "(2+(3*5))", što je neosporno korektno, ako zanemarimo prisustvo suvišnih zagrada. Za ispisivanje objekata tipa "Izraz" koristićemo, što je posve prirodno, operator "<<". Recimo, za ranije navedeni primjer objekta "e" tipa "Izraz", naredba poput

```
cout << e << endl;
```

trebala bi ispisati sljedeće:

```
((6*(-2))-(4+(9/3)))
```

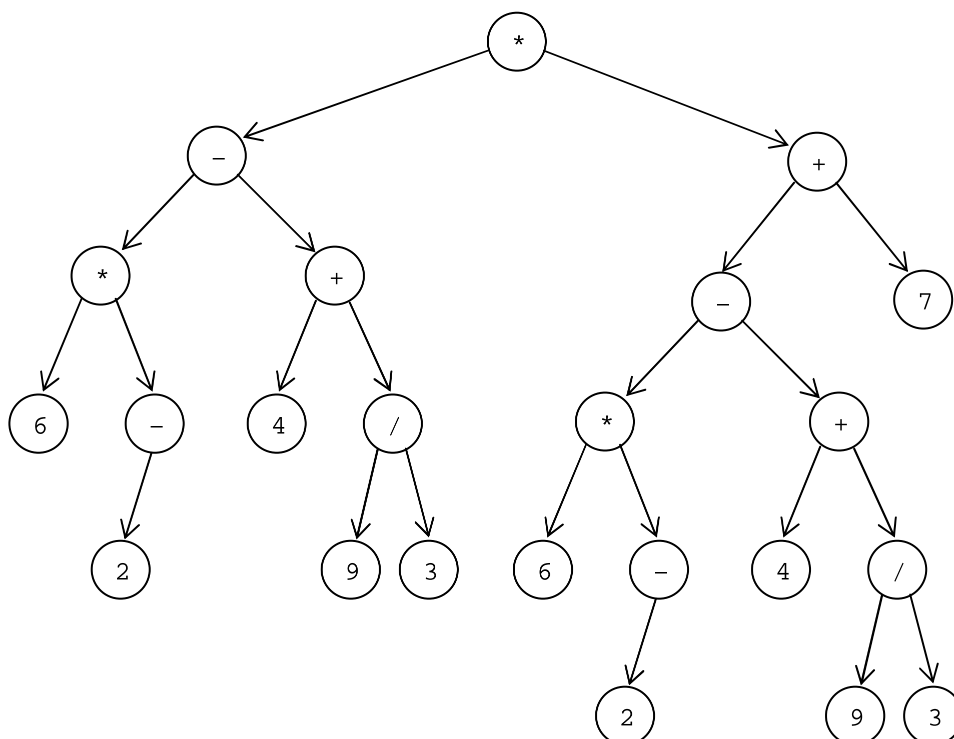
Razumije se da se već konstruisani objekti tipa "`Izraz`" mogu koristiti za formiranje složenijih izraza. Recimo, uz objekat "`e`" definiran kao u ranijem primjeru, naredbe poput

```
e = Izraz(e, "*", Izraz(e, "+", 7));
cout << e << endl;
```

trebale bi ispisati

```
(((6*(-2))-(4+(9/3)))*(((6*(-2))-(4+(9/3)))+7))
```

dok će "`e`" postati objekat koji logički odgovara sljedećem stablu:



Posljednje što ćemo za početak zahtijevati od klase "`Izraz`" je da korisnik ne treba da vodi ikakvu brigu kako su pohranjeni podaci u objektima tipa "`Izraz`" zaista interno organizirani, niti da vodi ikakvu brigu o alokaciji odnosno dealokaciji memorije. Korisnik će (za sada) samo kreirati i ispisivati objekte tipa "`Izraz`" i ništa drugo ne treba da bude njegova briga. Pri tome se podrazumijeva da se objekti tipa "`Izraz`" smiju slobodno kopirati, prenositi kao parametri u funkcije, itd.

Bitno je naglasiti da klasa "`Izraz`", iako djeluje pomalo specifična, obavlja zadatke koji su veoma česti u složenijim programima poput kompajlera, editora, CAD/CAM sistema i sličnih, a koji se u suštini svode na baratanje podacima komplikovane organizacije. Mnogo se truda u takvim programima potroši na manipuliranje raznim stablima, grafovima i drugim sličnim strukturama podataka. Autori takvih programa neprestano se susreću sa problemima upravljanja memorijom, kao i problemima postizanja odgovarajuće fleksibilnosti i efikasnosti. Kao što ćemo vidjeti, objektno orjentirani pristup programiranju omogućava da se rješenja pomenutih problema *lokaliziraju* (odnosno da se njihovo rješenje ne "provlači" i "rasipa" po čitavom programu), tako da kasnije dopune zahtjeva ne dovode do izmjena u onim dijelovima programa koji su već urađeni.

Pređimo sada na razmatranje kako bismo mogli riješiti postavljeni problem. Ako malo bolje pogledamo kako izgledaju stabla aritmetičkih izraza, na njima ćemo primijetiti dvije vrste objekata: *čvorove* (predstavljene kružićima) i *grane* odnosno *veze* (predstavljene strelicama). Da li je potrebno i za čvorove i za grane uvoditi posebne strukture podataka? Na prvi pogled, čini se da je dovoljno modelirati samo strukture podataka koje će reprezentirati čvorove. Pogledajmo dokle će nas takav pristup dovesti.

Možemo primijetiti da svaki čvor posjeduje svoj *sadržaj*, koji može biti ili atomarna vrijednost ili operator. Pored toga, čvor može imati različit broj potomaka (0, 1 ili 2, kako do sada stvari stoje). Očigledno postoje *različite vrste čvorova*. Mogli bismo pokušati napraviti klasu "`Cvor`" čiji bi jedan atribut predstavljao vrstu čvora, ali bi se dalje stvari prilično zakomplicirale, jer čvorovi različite vrste sadrže različite vrste informacija. Nešto bi se po tom pitanju i moglo učiniti pomoću nekih manje ili više prljavih dosjetki, ali je ovo dobar trenutak da se zapitamo da li razmišljamo u pravom smjeru. Generalno, kada god dođemo u situaciju da nam treba neki atribut kojim bismo razlikovali više podvrsta objekata, to je siguran znak da nešto nije u redu sa dizajnom, bar sa aspekta objektno orjentiranog programiranja, s obzirom da se takve situacije mogu mnogo prirodnije riješiti pomoću *nasljeđivanja* i *polimorfizma*. Zaista, tačno je da imamo *nekoliko različitih vrsta čvorova*, ali *svi su oni čvorovi*. Stoga je prirodno rješenje definirati jednu baznu klasu, koja modelira neki čvor bez obzira na njegovu vrstu, a zatim iz nje naslijediti klase koje modeliraju čvor koji sadrži atom, zatim čvor koji sadrži unarni operator, te čvor koji sadrži binarni operator.

Baznu klasu koja će modelirati proizvoljne čvorove nazovimo, recimo, "`CvorApstraktni`". Ona će sadržavati samo ono što je zajedničko za sve čvorove, bez obzira na njihovu vrstu. S druge strane, klase koje modeliraju specifične vrste čvorova, koje možemo nazvati recimo "`CvorAtom`", "`CvorUnarni`" i "`CvorBinarni`", sadržavaće specifičnosti pojedinih vrsta čvorova. Međutim, zahvaljujući mehanizmu nasljeđivanja, čvorovi bilo koje specifične vrste moći će se koristiti u svim kontekstima u kojima je predviđeno da se koriste objekti klase "`CvorApstraktni`", koja modelira čvor neodređene vrste.

Razmislimo sada šta treba da sadrži klasa "`CvorApstraktni`", odnosno šta je to zajedničko svim čvorovima. Na prvom mjestu, moramo imati mogućnost ispisa izraza čiji korijen leži u razmatranom čvoru, bez obzira koji je tip tog čvora. Stoga, klasa "`CvorApstraktni`" treba sadržavati neku metodu, nazovimo je "`Ispisi`", koja obavlja upravo taj zadatak. Parametar metode "`Ispisi`" mogao bi biti referenca na objekat izlaznog toka preko kojeg se ispis vrši (da bi se, recimo, podržao i ispis u datoteku), dok bi rezultat funkcije mogla biti referenca na taj isti tok (takva izvedba omogućava fleksibilniju upotrebu nego kada bismo stavili da metoda ne vraća nikakav rezultat). Međutim, sada nastupa problem: postupak ispisivanja očigledno zavisi od *vrste čvora* koji je u pitanju. Rješenje se sada samo nameće: ova metoda treba u baznoj klasi da bude *virtualna*, dok će svaka od klasa naslijeđenih iz klase "`CvorApstraktni`" sadržavati *svoju vlastitu verziju* metode "`Ispisi`", koja obavlja ispis na način specifičan za posmatranu vrstu čvora. Zahvaljujući činjenici da je metoda "`Ispisi`" virtualna, kad god se nad pokazivačem ili referencom na objekat tipa "`CvorApstraktni`" pozove metoda "`Ispisi`", pozvaće se ona metoda "`Ispisi`" koja pripada onoj vrsti čvora na koju razmatrani pokazivač zaista u tom trenutku pokazuje (ili na koji je razmatrana referenca vezana). Drugim riječima, pokazivači i reference na objekte tipa "`CvorApstraktni`" postaju *polimorfne promjenljive*. Ovo je pravi primjer primjene polimorfizma: čvorovi različitih vrsta različito se ponašaju na poziv metode "`Ispisi`", ali pokazivači na čvorove nespecificirane vrste mogu pokazivati na čvorove posve konkretne vrste, pri čemu će se oni tada ponašati u skladu sa onim na šta pokazuju. Isto vrijedi i za reference: reference nespecificirane vrste mogu se vezati za čvorove posve konkretne vrste i tada se one ponašaju potpuno u skladu sa vrstom čvora za koji su se vezale.

Pored metode "`Ispisi`", bilo bi zgodno da se izrazi čiji korijen leži u razmatranom čvoru mogu ispisivati pomoću operatora "`<<`" prostim slanjem tog čvora na tok, umjesto pozivom metode "`Ispisi`". To lako možemo uraditi prostim pozivom metode "`Ispisi`" iz operatorske funkcije koja definira traženo ponašanje operatora "`<<`". Na taj način se približavamo zahtjevima iz postavke problema. Ako malo razmislimo, vidjećemo da za sada nema ništa više što bi bilo zajedničko za sve čvorove, tako da možemo formirati definiciju klase "`CvorApstraktni`" (obratimo pažnju na *virtualni destruktor*, koji je neophodan kad god koristimo polimorfizam):

```
class CvorApstraktni {
  virtual ostream &Ispisi(ostream &tok) const = 0;
public:
  virtual ~CvorApstraktni() {}
  friend ostream &operator <<(ostream &tok, const CvorApstraktni &c) {
    return c.Ispisi(tok);
  }
};
```

Ovdje je metoda "`Ispisi`" definirana kao čista virtualna metoda, s obzirom da u baznoj klasi zaista nemamo informacija na osnovu koje bismo je mogli implementirati, jer ne znamo kako ispisivati izraze čiji korijeni leže u čvoru čiju vrstu ne znamo. Naravno, ovo nije problem, jer će se u svakom slučaju pozivati metoda "`Ispisi`" definirana u klasi koja modelira čvor konkretne vrste. Deklaracija čiste virtualne metode ujedno čini baznu klasu "`CvorApstraktni`" apstraktnom baznom klasom, što znači da se objekti tipa "`CvorApstraktni`" uopće ne mogu ni kreirati, nego samo objekti tipova naslijeđenih iz njega (tj. čvorovi konkretne vrste). Ako malo razmislimo, vidjećemo da nam upravo ovo i odgovara.

Primijetimo da je glavna svrha uvođenja pomoćne virtualne funkcije "`Ispisi`" njen poziv iz operatorske funkcije za preklapanje operatora "`<<`". Zahvaljujući činjenici da je ona virtualna, pozvaće se metoda "`Ispisi`" iz onog tipa na koji je referenca "`c`" zaista u trenutku poziva vezana. Neko bi se mogao zapitati zar nismo mogli odmah definirati da operatorska funkcija za preklapanje operatora "`<<`" bude virtualna, umjesto da ona poziva virtualnu metodu "`Ispisi`". Odgovor je odrečan. Naime, operatorska funkcija za preklapanje operatora "`<<`" je *obična funkcija*, a ne funkcija članica, a *samo funkcije članice mogu biti virtualne*!

Pređimo sada na razmatranje kako bi trebale izgledati klase koje modeliraju konkretne vrste čvorova. Najjednostavnija je klasa koja modelira čvorove koji sadrže atome. Ona očigledno mora imati atribut koji čuva vrijednost atoma, i to će joj biti jedini atribut. Ona također mora implementirati svoju verziju metode "`Ispisi`", koja je trivijalna: prosto se ispisuje vrijednost atoma. Zgodno je još izvesti i konstruktor koji inicijalizira vrijednost atoma, i to bi bilo sve:

```
class CvorAtom : public CvorApstraktni {
  double vrijednost;
  ostream &Ispisi(ostream &tok) const { return tok << vrijednost; }
public:
  CvorAtom(double vrijednost) : vrijednost(vrijednost) {}
};
```

Što se tiče čvorova koji sadrže operatore, oni očigledno moraju sadržavati informaciju o operatoru i operandima. Stoga ćemo svakako imati atribut tipa "`string`" koji čuva informaciju o operatoru. Što se tiče informacija o operandima, to je malo problematičnije s obzirom da operandi mogu biti različitih vrsta. Stoga, informacije o operandima *moraju biti polimorfne*, odnosno odgovarajući atributi moraju biti ili pokazivači ili reference na objekte tipa "`ApstraktniCvor`". Odlučimo se, recimo, za pokazivače (upotrebom referenci ne bismo ostvarili nikakve suštinske prednosti). Potrebni su još konstruktori za inicijalizaciju atributa i verzije metoda "`Ispisi`" koje ispisuju izraze čiji je korijen u razmatranom čvoru. Sve u svemu, definicije klasa "`CvorUnarni`" i "`CvorBinarni`" mogle bi izgledati ovako:

```
class CvorUnarni : public CvorApstraktni {
  string operacija;
  CvorApstraktni *pok_operand;
  ostream &Ispisi(ostream &tok) const {
    return tok << "(" << operacija << *pok_operand << ")";
  }
public:
  CvorUnarni(const string &operacija, CvorApstraktni *pok_operand) :
    operacija(operacija), pok_operand(pok_operand) {}
};

class CvorBinarni : public CvorApstraktni {
  string operacija;
  CvorApstraktni *pok_operand_1, *pok_operand_2;
  ostream &Ispisi(ostream &tok) const {
    return tok << "(" << *pok_operand_1 << operacija
      << *pok_operand_2 << ")";
  }
public:
  CvorBinarni(CvorApstraktni *pok_operand_1, const string &operacija,
    CvorApstraktni *pok_operand_2) : operacija(operacija),
    pok_operand_1(pok_operand_1), pok_operand_2(pok_operand_2) {}
};
```

Interesantno je razmotriti kako su izvedene metode "`Ispisi`". One se oslanjaju na to da, kakav god da je tip operanada (atom, ili ponovo izraz u čijem je korijenu unarni ili binarni operator), postupak za ispis tog operanda je *definiran* u klasi koji opisuje taj tip operanda. Primijetimo da su ove metode zapravo *rekurzivne*: one će pozvati *same sebe* kad god se pojavi potreba za ispisom nekog podizraza posmatranog izraza čiji je korijen istog tipa kao i tip čitavog izraza koji se ispisuje.

Pogledajmo šta smo do sada postigli. Mada još nismo definirali objekte tipa "`Izraz`", uvedena hijerarhija čvorova nam već omogućava neku vrstu kreiranja stabala aritmetičkih izraza. Doduše, još smo daleko od sintakse koja je tražena u postavci problema. Recimo, za kreiranje i ispis stabla aritmetičkog izraza "`6*(-2)-(4+9/3)`", zamišljena sintaksa je bila

```
Izraz e(Izraz(6, "*", Izraz("-", 2)), "-", Izraz(4, "+",
  Izraz(9, "/", 3)));
cout << e << endl;
```

ili, alternativno,

```
Izraz e = Izraz(Izraz(6, "*", Izraz("-", 2)), "-", Izraz(4, "+",
  Izraz(9, "/", 3)));
cout << e << endl;
```

Ovako nešto je za sada nemoguće: konstruktori koji grade čvorove koji odgovaraju unarnim i binarnim operatorima očekuju *pokazivače*, a ne objekte tipa "`Izraz`" (takvi objekti za sada još uvijek ne postoje). Međutim, svako manipuliranje sa pokazivačima na nekakve objekte zahtijeva da ti objekti ili od ranije postoje, ili da se oni *stvore dinamički*. Razumije se da nam najveću fleksibilnost daje mogućnost dinamičke alokacije (u suprotnom bismo svaki čvor morali posebno definirati kao imenovani objekat), ali tada se neko mora pobrinuti za oslobađanje alocirane memorije. Dakle, za sada jedina mogućnost da postignemo sličan efekat kao u prethodne dvije konstrukcije (koje za sada još ne rade) je pomoću prilično rogobatne konstrukcije, u kojoj se vrši dinamička alokacija svakog neophodnog čvora:

```
CvorBinarni *pok_e(new CvorBinarni(new CvorBinarni(new CvorAtom(6),
  "*", new CvorUnarni("-", new CvorAtom(2))), "-",
  new CvorBinarni(new CvorAtom(4), "+", new CvorBinarni(
  new CvorAtom(9), "/", new CvorAtom(3)))));
cout << *pok_e << endl;
```

Ova sintaksa je daleko od onoga što smo željeli. Međutim, problem sintakse je najmanji problem u odnosu na probleme koji nas čekaju. Problem sintakse nastaje prosto zbog činjenice da mi još uvijek nismo ni dizajnirali klasu "`Izraz`", tako da će problem sintakse biti riješen sam po sebi onog trenutka kada razvijemo ovu klasu. Kako bi ova klasa mogla izgledati? Prirodno bi bilo kao njen jedini atribut staviti pokazivač na korijen stabla, dok bi konstruktori kreirali odgovarajuće čvorove, zavisno od broja parametara. Također bismo u ovoj klasi trebali definirati operator ispisa "`<<`" tako da se može primjenjivati na objekte tipa "`Izraz`" (ona bi se mogla samo preusmjeriti na odgovarajuću operatorsku funkciju definiranu u klasi "`CvorApstraktni`"). Tako bi klasa "`Izraz`" mogla izgledati recimo ovako:

```
class Izraz {
  CvorApstraktni *pok_korijen;
public:
  Izraz(double vrijednost) : pok_korijen(new CvorAtom(vrijednost)) {}
  Izraz(const string &operacija, const Izraz &operand) :
    pok_korijen(new CvorUnarni(operacija, operand.pok_korijen)) {}
  Izraz(const Izraz &operand_1, const string &operacija,
    const Izraz &operand_2) : pok_korijen(new CvorBinarni(
    operand_1.pok_korijen, operacija, operand_2.pok_korijen)) {}
  friend ostream &operator <<(ostream &tok, const Izraz &e) {
    return tok << *e.pok_korijen;
  }
};
```

Ovako definirana klasa "Izraz" riješava problem sintakse, i sintaksno sve radi u skladu sa onim što je traženo u postavci problema. Klasa "Izraz" u sebi očigledno "sakriva" čvorove, tako da korisnik klase "Izraz" ne mora znati da strukture podataka koje modeliraju čvorove uopće postoje (a pogotovo ne mora znati da je za njihovu realizaciju iskorišten mehanizam nasljeđivanja i polimorfizam). Prikazano "rješenje" izvedeno je u programskoj datoteci "izraz1.cpp" priloženoj uz ovo predavanje. Međutim, u ovoj izvedbi potpuno je zanemaren *problem upravljanja memorijom*, odnosno upravljanje memorijom *nismo niti pokušali riješiti*. Naime, primijetimo da konstruktori klase "Izraz" samo *kreiraju nove čvorove*, koje kasnije *niko ne briše*. Jasno je da sve alocirane čvorove neko mora i obrisati. Pitanje je *ko* i *kada*. Ukoliko koristimo ovako napisanu klasu "Izraz", ručno brisanje ne možemo obaviti sve i da hoćemo, jer nemamo direktan pristup niti jednom od pokazivača koji pokazuju na kreirane čvorove. Čak i u ranije navedenom primjeru u kojem su svi neophodni čvorovi ručno kreirani (eksplicitnim pozivom operatora "**new**"), programer nije u mogućnosti da neposredno obriše kreirane čvorove, s obzirom da pokazivači na objekte kreirane unutrašnjim pozivima operatora "**new**" nisu nigdje sačuvani na način koji bi programeru bio dostupan. Razumije se da mi zapravo i ne želimo da neposredno pristupamo svim tim pokazivačima, jer nešto što definitivno nikako ne želimo je da za postizanje željenog efekta uz oslobađanje zauzete memorije moramo pisati nešto poput

```
CvorAtom *pok_1(new CvorAtom(6));
CvorAtom *pok_2(new CvorAtom(2));
CvorUnarni *pok_3(new CvorUnarni("-", pok_2));
CvorBinarni *pok_4(new CvorBinarni(pok_1, "*", pok_3));
CvorAtom *pok_5(new CvorAtom(4));
CvorAtom *pok_6(new CvorAtom(9));
CvorAtom *pok_7(new CvorAtom(3));
CvorBinarni *pok_8(new CvorBinarni(pok_6, "/", pok_7));
CvorBinarni *pok_9(new CvorBinarni(pok_5, "+", pok_8));
CvorBinarni *pok_e(new CvorBinarni(pok_4, "-", pok_9));
cout << *pok_e << endl;
delete pok_1; delete pok_2; delete pok_3; delete pok_4; delete pok_5;
delete pok_6; delete pok_7; delete pok_8; delete pok_9; delete pok_e;
```

Očigledno, trenutno razvijeni dizajn ne omogućava nikakav pogodan način za oslobađanje alocirane memorije. Prirodno se postavlja pitanje može li se oslobađanje memorije nekako automatizirati? Ako ništa drugo, zar destruktori ne služe upravo za tu svrhu? Iako je odgovor u suštini potvrdan, nije sve baš tako jednostavno kao što na prvi pogled izgleda. Da bismo uočili sve potencijalne probleme, zanemarimo za trenutak klasu "Izraz" (tj. pretpostavimo da ona još uvijek ne postoji), nego se fokusirajmo samo na čvorove. Mogli bismo pokušati u sve potomke čvorova koji odgovaraju unarnim i binarnim operatoroma *staviti u njihovo vlasništvo*, odnosno u klase "CvorUnarni" i "CvorBinarni" dodati destruktore koji prosto oslobađaju memoriju na koju pokazuju pokazivači koji pokazuju na njihove potomke, tako da se, kada se neki čvor ukloni, automatski uklone i svi njegovi potomci (što bi dalje, rekurzivno, dovelo do brisanja cijelog stabla sa korijenom u posmatranom čvoru). Recimo, u klasu "CvorBinarni" bismo mogli pokušati dodati sljedeći destruktor:

```
class CvorBinarni : public CvorApstraktni {
  ...
public:
  ...
  ~CvorBinarni() { delete operand_1; delete operand_2; }
};
```

Na prvi pogled, ovakvo rješenje bi riješilo problem oslobađanja memorije. Međutim, kod ovog rješenja se može pojaviti gadan problem što bi u nekim situacijama isti objekat mogao biti obrisan više od jedanput, a znamo da brisanje već obrisanog objekta može dovesti do kraha sistema. Recimo, neka želimo kreirati stablo aritmetičkog izraza "(2+6/3)*(2+6/3)". Pošto se u njemu isti podizraz "2+6/3" javlja dvaput, želja nam je da se ta kreacija može obaviti konstrukcijom poput

```
Izraz e(2, "+", Izraz("/", 6, 3));
e = Izraz(e, "*", e);
```

Pošto smo rekli da ćemo za trenutak zanemariti postojanje klase "`Izraz`", posmatrajmo umjesto ove konstrukcije funkcionalno ekvivalentnu, ali sintaksno rogobatniju konstrukciju, u kojoj se svi čvorovi kreiraju eksplicitno (i u kojoj se mnogo jasnije vidi šta se tačno dešava "ispod haube"):

```
CvorBinarni *pok_e(new CvorBinarni(new CvorAtom(2), "+",
  new CvorBinarni(new CvorAtom(6), "/", new CvorAtom(3))));
pok_e = new CvorBinarni(pok_e, "*", pok_e);
```

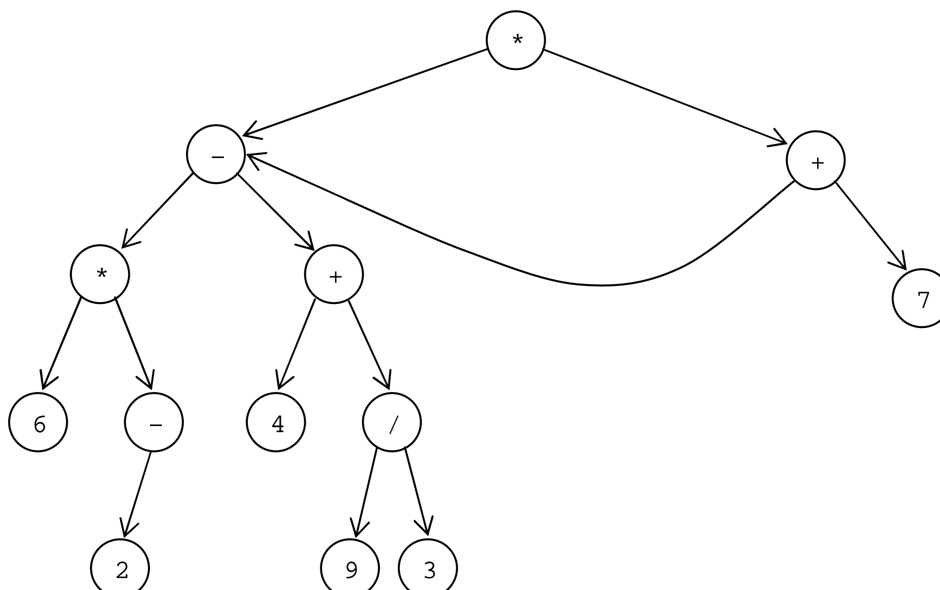
Mada ovo u načelu radi, izuzetno je važno uočiti da će u ovom primjeru posljednji kreirani čvor koji odgovara binarnoj operaciji sadržavati *dva identična pokazivača* koja odgovaraju lijevom i desnom operandu, odnosno oba pokazivača će pokazivati na *jedno te isto podstablo*, umjesto na dva neovisna, ali identična podstabla. Ovo u suštini uopće nije loše (štedi se memorija), ali sve dok korisnik nema problema zbog te činjenice. Nažalost, ukoliko bi klasa "`CvorBinarni`" imala destruktor koji oslobađa memoriju na koju pokazuju njeni pokazivači, imali bismo dvostruko oslobađanje memorije, jer u ovom slučaju oba njena pokazivača pokazuju na isto mjesto u memoriji! Konkretnije, onog trenutka kada bismo izvršili naredbu poput "**`delete`** `pok_e`", došlo bi do dvostrukog oslobađanja memorije. Već smo rekli da se ovo ne smije dozvoliti. Sličan, samo donekle skriveniji problem bismo imali ukoliko bismo htjeli postići onaj efekat koji smo planirali da se postiže naredbom poput

```
e = Izraz(e, "*", Izraz(e, "+", 7));
```

S obzirom da smo rekli da nećemo koristiti klasu "`Izraz`" sve dok detaljno ne istražimo šta se dešava "u pozadini", naredba najsličnija prethodnoj koja ne koristi klasu "`Izraz`" glasi

```
pok_e = new CvorBinarni(pok_e, "*", new CvorBinarni(pok_e, "+",
  new CvorAtom(7)));
```

Međutim, pažljivim posmatranjem možemo vidjeti da će i ova konstrukcija kreirati takvu strukturu u kojoj neka dva pokazivača pokazuju na isti čvor. Recimo, pretpostavimo li da je prije izvršavanja ove naredbe varijabla "`pok_e`" pokazivala na korijen stabla koje odgovara izrazu "`6*(-2)-(4+9/3)`", nakon izvršavanja ove naredbe, varijabla "`pok_e`" će pokazivati na korijen "stabla" prikazanog na sljedećoj slici. Riječ "stablo" stavljena je pod navodnike, jer strogo rečeno ovo više nije stablo, nego općenitija struktura podataka, poznata pod nazivom *usmjereni aciklični graf* ili *dag* (od engl. *directed acyclic graph*). U načelu, to nije nikakav problem sve dok korisnik ne vidi razliku (tj. sve dok korisniku izgleda da zaista ima pravo stablo u kojem se jedno podstablo ponavlja na dva mjesta). Nažalost, korisnik će (uz ovako izvedene destruktore) itekako vidjeti razliku, jer će naredba poput "**`delete`** `pok_e`" neminovno dovesti do kraha. Zaista, rekurzivno brisanje potomaka čvora na koji "`pok_e`" pokazuje u jednom trenutku dovešće do brisanja objekta koji je već obrisan!



Stvari se očito kompliciraju. Prave komplikacije tek dolaze kada se u igru uključi i klasa "`Izraz`". Zanemarimo za trenutak uočene probleme da bi se mogli fokusirati na neke još nezgodnije probleme. Zapitajmo se prvo *ko će uopće inicirati brisanje*? Mi ne želimo da korisnik klase "`Izraz`" ikada eksplicitno mora koristiti operator "**`delete`**". Odgovor djeluje izuzetno prirodan: brisanje treba inicirati destruktor klase "`Izraz`". Zaista, vrlo prirodno djeluje da se u klasu "`Izraz`" doda destruktor koji bi izvršio naredbu "**`delete`**" nad pokazivačem na korijen stabla, a to bi dalje rekurzivno dovelo do brisanja čitavog stabla (naravno, uz pretpostavku da su u čvorovima implementirani destruktori kako je gore opisano). Za klasu "`Izraz`" to bi izgledalo ovako:

```
class Izraz {
  ...
public:
  ...
  ~Izraz() { delete pok_korijen; }
};
```

Nažalost, ovim ne samo da nismo riješili problem, nego smo postigli da će čak i posve bezazlena deklaracija poput

```
Izraz e("-", 2);
```

ostaviti sistem u totalno nestabilnom stanju koje će prije ili kasnije sigurno dovesti do kraha. Šta se ovdje zapravo dešava? Primijetimo prvo da je prethodna konstrukcija, zahvaljujući neeksplicitnom konstruktoru klase "`Izraz`" sa jednim parametrom, praktično ekvivalentna konstrukciji

```
Izraz e("-", Izraz(2));
```

Pratimo sada pažljivo šta se ovdje dešava. Konstrukcija "`Izraz(2)`" kreira pomoćni bezimeni objekat tipa "`Izraz`" i inicijalizira ga pozivom konstruktora sa jednim parametrom. Taj konstruktor dalje dinamički kreira jedan atomarni čvor (koji se inicijalizira parametrom "`2`") i postavlja atribut "`pok_korijen`" u tom pomoćnom bezimenom objektu da pokazuje na novokreirani čvor. Tako kreirani objekat prosljeđuje se kao drugi parametar konstruktoru objekta "`e`". U ovom konstruktoru kreira se još jedan čvor (ovaj put čvor koji odgovara unarnoj operaciji). Taj čvor inicijalizira se operacijom zadanom kroz prvi parametar konstruktora objekta "`e`", te pokazivačem "`pok_korijen`" preuzetim iz drugog parametra (taj pokazivač pokazuje na maločas kreirani atomarni čvor). Naravno, pokazivač "`pok_korijen`" unutar objekta "`e`" pokazivaće upravo na svježe kreirani čvor koji odgovara unarnoj operaciji. Sve je zasada upravo kako treba da bude. Međutim, problemi upravo počinju. Pomoćni bezimeni objekat kreiran konstrukcijom "`Izraz(2)`" prestaje postojati onog trenutka kada više nije potreban (uklanjanje pomoćnih bezimenih objekata obično se dešava odmah po završetku izraza unutar kojih su upotrijebljeni). Neposredno prije trenutka kada taj bezimeni objekat prestane postojati (tačan trenutak kada se to dešava nije uopće ni bitan), biće automatski pozvan njegov destruktor, koji će obrisati atomarni čvor kreniran unutar njegovog konstruktora. Problem je u tome što na taj isti atomarni čvor pokazuje i pokazivač unutar maloprije kreiranog čvora koji odgovara unarnoj operaciji. Dakle, čvor koji odgovara unutarnoj operaciji nakon uništavanja atomarnog čvora sadrži divlji pokazivač! Slijedi da je objekat "`e`" u neispravnom stanju (s obzirom da jedan od njegovih "sastavnih dijelova" sadrži divlji pokazivač) i krah je vrlo vjerovatan čim pokušamo objekat "`e`" upotrijebiti izašta. Štaviše, do kraha će sigurno doći onog trenutka kada i sam objekat "`e`" prestane postojati. Naime, tada će se pozvati njegov destruktor, koji će pokušati obrisati čvor koji odgovara unarnoj operaciji, a na koji pokazuje njegov pokazivač "`pok_korijen`". Brisanje tog čvora će dovesti do pozivanja destruktora čvora, koji će pokušati da obriše objekat na koji pokazuje pokazivač unutar njega, a vidjeli smo da je taj objekat već obrisan! Dakle, ponovo imamo dvostruko oslobađanje memorije. Bitno je uočiti da problem ne bi bio riješen kada bismo umjesto pomoćnog bezimenog objekta imali imenovani objekat, nego bi samo bio odgođen za kasnije (do trenutka kada taj imenovani objekat prestaje postojati). Šta god učinili, imamo dva pokazivača (u dva različita objekta) koji pokazuju na isti čvor, tako da je, prije ili kasnije, dvostruko oslobađanje memorije sa ovako izvedenim destruktorima neizbježno!

Vidimo da naš naivni pokušaj da riješimo upravljanje memorijom nije završio uspješno. To očigledno ukazuje da klase koje smo do sada razvili nisu u potpunosti dobro dizajnirane. Imamo propust u dizajnu i

to je stvar koja se relativno često dešava u razvoju softvera, ukoliko se odmah u startu ne sagledaju dobro sve činjenice. Tehnički gledano, problem se može objasniti na veoma jednostavan način: poletili smo da implementiramo *destruktore*, dok smo potpuno zaboravili na *konstruktore kopije*, a poznato je da te dvije stvari uvijek idu zajedno (doduše, tu još obavezno ide i *preklopljeni operator dodjele*, ali to ćemo sve kasnije srediti u jednom paketu). Ispravno implementirani konstruktori kopije (i preklopljeni operatori dodjele) mogli bi riješiti problem (pri čemu tek trebamo da vidimo kako bi to trebala da izgleda "ispravna" implementacija). Znači li to da bismo u svaku klasu u koju smo uveli destruktor morali uvesti i konstruktor kopije kao i preklopljeni operator dodjele? U načelu, odgovor je potvrdan. Međutim, prije nego što počnemo kukati kako će to biti vrlo naporno i nezgrapno, treba ukazati na činjenicu da nam je loše osmišljen dizajn jednostavno doveo do toga da nam destruktor treba na *previše mjesta* (što bi za posljedicu imalo i da bi nam konstruktor kopije i preklopljeni operator dodjele također trebali na previše mjesta). Uskoro ćemo vidjeti da se boljim dizajnom može postići da će nam destruktor trebati *na samo jednom mjestu* (što povlači i samo jedan konstrukor kopije i samo jedan preklopljeni operator dodjele). U nastavku ćemo malo detaljnije sa *čisto logičke strane* razmotriti kako je došlo do propusta u dizajnu.

Da bismo uvidjeli gdje su napravljeni logički propusti u dizajnu, moramo se vratiti skroz na samu postavku problema. Podsjetimo se da smo zaključili da se stabla aritmetičkih izraza sastoje od *čvorova* i *grana* (*veza*). Kako stvari do sada stoje, mi smo pomoću odgovarajućih klasa modelirali *isključivo čvorove*, dok smo grane odnosno veze modelirali prosto preko *pokazivača*. Upravo u tome je problem: pokazivači su *neinteligentni objekti* koji nisu u stanju da automatski upravljaju zauzimanjem i oslobađanjem memorije, nego se o tome eksplicitno brine programer (pomoću operatora "**`new`**" i "**`delete`**"). Jedan od načina za rješavanje problema bio bi zamjena pokazivača nekom "inteligentnijom" strukturom podataka (recimo, nekom namjenski napisanom klasom) koja bi unutar sebe sadržavala implementiran mehanizam zauzimanja i oslobađanja memorije, a ne samo "sirovi" pokazivač (tj. nekom vrstom "pametnog" pokazivača). Takva klasa bi zapravo modelirala grane stabla na adekvatniji način nego što to čine pokazivači. Mogli bismo razviti posebnu klasu za tu svrhu, ali pokazaćemo da se upravo sama klasa "`Izraz`" može lijepo upotrijebiti za tu svrhu, odnosno mehanizmi za upravljanje memorijom koji se ugrade u tu klasu mogu se iskoristiti i za upravljanje memorije u samim čvorovima.

Razmotrimo sada malo podrobnije samu klasu "`Izraz`". Jasno je da ukoliko je ova klasa odgovorna za upravljanje memorijom, ona mora posjedovati nekakav destruktor koji oslobađa sve resurse koje su objekti tipa "`Izraz`" zauzeli onog trenutka kada se ustanovi da oni više nisu potrebni (šta će tačno raditi ovaj destruktor, uvidjećemo nešto kasnije). Međutim, jasno je da gdje god prisutan destruktor, trebaju biti prisutni i konstruktor kopije i preklopljeni operator dodjele (osim ukoliko želimo zabraniti kopiranje i međusobno dodjeljivanje primjeraka klase, a to ovdje ne dolazi u obzir). Dakle, konstruktor kopije i preklopljeni operator dodjele nam trebaju, te prvo razmotrimo kako bi oni mogli izgledati.

Što se tiče konstruktora kopije, prvo što pada na pamet je da bi konstrukor kopije trebao napraviti cjelokupnu kopiju stabla na koje pokazuje pokazivač na korijen stabla, čime bismo spriječili da ikada imamo dva različita pokazivača koji pokazuju na isti čvor, tako da se problemi sa dvostrukim brisanjem memorije više ne bi javljali. To zaista jeste jedna od mogućnosti, ali koja nije baš trivijalna za izvesti, jer stablo može imati složenu strukturu. Osnovna ideja za izvedbu tog kopiranja je sljedeća. U slučaju da korijen pokazuje na čvor koji ima potomke, trebalo bi rekurzivno napraviti kopije stabala na koje pokazuju njegovi potomci, nakon čega bi trebalo napraviti kopiju samog čvora i u njega smjestiti pokazivače na tako napravljene kopije stabala. Međutim, to je mnogo lakše reći nego izvesti, s obzirom da potomci mogu biti čvorovi različitih vrsta. Vjerovatno najbolje rješenje je da se u baznu klasu "`ApstraktniCvor`" doda čista virtualna metoda "`NapraviKopiju`", pri čemu bi se u svakoj klasi izvedenoj iz ove bazne klase implementirala konkretna verzija metode "`NapraviKopiju`" koja bi kreirala kopiju primjerka klase nad kojim je pozvana (pri tome, različite vrste čvorova kopiraju sebe na različite načine). Tada bi se kopiranje stabala na koji pokazuju potomci prosto vršilo pozivom metode "`NapraviKopiju`", a mehanizam virtualnih funkcija garantirao bi da će za svaku konkretnu vrstu čvora biti pozvana ispravna verzija metode "`NapraviKopiju`" (sličan mehanizam smo imali kada smo pravili kontejnersku klasu koja sadrži kolekciju polimorfnih likova u Predavanju 12).

Sve što je do sada rečeno bilo bi mnogo jasnije kada bismo prikazali konkretnu implementaciju izložene ideje. Međutim, umjesto da žurimo sa implementacijom, pametnije je zapitati se *da li nam to zaista treba*. Odnosno, da li konstruktor kopije mora zaista da napravi cjelokupnu kopiju stabla, ili se

možemo zadovoljiti plitkim kopijama? Znamo da korisnik klase ne može vidjeti nikakvu razliku između plitke i potpune kopije sve dok ne pokuša da promijeni kopiju (u slučaju plitke kopije, promjena će se odraziti i na original). Ako uvidimo da naša klasa "Izraz" ne predviđa nikakvu metodu kojom bi se struktura izraza mogla promijeniti, zaključujemo da će plitke kopije raditi sasvim dobro, a stvari će svakako biti neuporedivo efikasnije nego ukoliko bismo zaista kreirali kompletnu kopiju čitavog stabla. Konstruktor kopije će tada samo trebati da vodi evidenciju o tome koliko pokazivača pokazuje na jedan te isti čvor, da destruktor ne bi obrisao čvor (zajedno sa cijelim podstablom koje eventualno iz njega izlazi) ukoliko postoji još pokazivača koji na njega pokazuju. Ovo je, u suštini, tehnika brojanja pristupa (brojanja referenciranja) o kojoj smo već govorili.

Ukoliko smo se odlučili za plitke kopije, tada je posve jasno da se ponovo može desiti da više pokazivača pokazuje na isti čvor (ukoliko neki izraz sadrži više identičnih podizraza), odnosno da stablo postane dag, slično kao u do sada postojećem rješenju koje eksplicitno manipulira sa pokazivačima (da smo insistirali na dubokim kopijama, ovo se ne bi moglo desiti). Već smo rekli da ovo nije loše sa aspekta štednje memorije, ali tada moramo paziti da destruktor ne obriše dva puta isti objekat. Sve u svemu, zaključujemo da je potrebno u svakom čvoru voditi evidenciju o tome koliko pokazivača na njega pokazuje (u načelu, brojač bi se mogao i dinamički kreirati izvan samog čvora, slično kao što smo radili kada smo se prvi put upoznavali sa tehnikom brojanja pristupa, ali ovdje nema nikakvog razloga da brojač ne bude *u samom čvoru*). Stoga ćemo u klasu "CvorApstraktni" dodati novi atribut "brojac" koji će brojati koliko pokazivača pokazuje na čvor. Konstruktor klase će ovaj brojač inicijalizirati na 1. S obzirom da će metode klase "Izraz" pristupati ovom brojaču, klasu "Izraz" ćemo proglasiti prijateljskom klasom klase "CvorApstraktni". Također, kako nikada nećemo direktno ispisivati čvorove nego samo objekte klase "Izraz", nema nikakve potrebe da definiramo funkciju za preklapanje operatora "<<" za klasu "CvorApstraktni", nego samo za klasu "Izraz". Ipak, ovu funkciju treba proglasiti prijateljem klase "CvorApstraktni", s obzirom da će ona pozivati njenu privatnu metodu "Ispisi". Konačno, konstruktor klase "CvorApstraktni" izveden je u "**protected**" sekciji, čime se garantira da će samo nasljednici ove klase moći koristiti ovaj konstruktor. Stoga, bi nova verzija klase "CvorApstraktni" mogla izgledati recimo ovako:

```
class CvorApstraktni {
  int brojac;
  virtual ostream &Ispisi(ostream &tok) const = 0;
protected:
  CvorApstraktni() : brojac(1) {}
  virtual ~CvorApstraktni() {}
  friend class Izraz;
  friend ostream &operator <<(ostream &tok, const Izraz &e);
};
```

Klasa "Izraz" sadržavaće sve što i do sada (uz minijaturnu izmjenu definicije preklopljenog operatora "<<" koji sad direktno poziva metodu "Ispisi" nad korijenom, s obzirom da čvorovi više ne podržavaju ovaj operator), samo što treba predvidjeti i destruktor, konstrukor kopije i preklopljeni operator dodjele:

```
class Izraz {
  CvorApstraktni *pok_korijen;
public:
  Izraz(double vrijednost);
  Izraz(const string &operacija, const Izraz &operand);
  Izraz(const Izraz &operand_1, const string &operacija,
    const Izraz &operand_2);
  ~Izraz();
  Izraz(const Izraz &e);
  Izraz &operator =(const Izraz &e);
  friend ostream &operator <<(ostream &tok, const Izraz &e) {
    return e.pok_korijen->Ispisi(tok);
  }
};
```

Implementacije ćemo odložiti do trenutka kada vidimo kako će izgledati nove verzije klasa naslijeđenih iz klase "CvorApstraktni". Što se tiče klase "CvorAtom", ona bi načelno mogla ostati ista kao i ranije. Međutim, ipak ćemo učiniti jednu izmjenu i konstruktor ove klase proglasiti privatnim, a klasu "Izraz" proglasiti njenom prijateljskom klasom. Šta smo ovim postigli? Zahvaljujući privatnom konstruktoru, niko neće moći kreirati primjerke klase "CvorAtom" osim metoda klase "Izraz". Ako malo bolje razmislimo, uvidjećemo da tako zapravo i treba da bude:

```
class CvorAtom : public CvorApstraktni {
  double vrijednost;
  ostream &Ispisi(ostream &tok) const { return tok << vrijednost; }
  CvorAtom(double vrijednost) : vrijednost(vrijednost) {}
  friend class Izraz;
};
```

Slične "kozmetičke" izmjene načinićemo i u klasama "CvorUnarni" i "CvorBinarni". Međutim, u njima ćemo izvršiti i jednu dodatnu, na prvi pogled sitnu, ali zapravo vrlo radikalnu izmjenu, koja je ključ za rješenje mnogih problema sa kojima bi se mogli susresti. Naime, primijetimo da bi svaka klasa koja sadrži pokazivače na dinamički alocirane objekte koji su u njenom vlasništvu trebala posjedovati destruktor, konstruktor kopije i preklopljeni operator dodjele. Odavde očigledno slijedi da ćemo imati tim manje problema što manje imamo klasa koje sadrže takve pokazivače. Upravo u skladu sa tom idejom, u klasama "CvorUnarni" i "CvorBinarni" ćemo potpuno izbjeći atribute koji su pokazivači! Konkretno, atribute koji predstavljaju operande nećemo više čuvati kao pokazivače na objekte tipa "CvorApstraktni", nego kao objekte tipa "Izraz", koliko god to na prvi pogled djelovalo neobično. Također, u skladu tim ćemo neznatno modificirati konstruktore ovih klasa, kao i metodu "Ispisi":

```
class CvorUnarni : public CvorApstraktni {
  string operacija;
  Izraz operand;
  ostream &Ispisi(ostream &tok) const {
    return tok << "(" << operacija << operand << ")";
  }
  CvorUnarni(const string &operacija, const Izraz &operand) :
    operacija(operacija), operand(operand) {}
  friend class Izraz;
};
class CvorBinarni : public CvorApstraktni {
  string operacija;
  Izraz operand_1, operand_2;
  ostream &Ispisi(ostream &tok) const {
    return tok << "(" << operand_1 << operacija << operand_2 << ")";
  }
  CvorBinarni(const Izraz &operand_1, const string &operacija,
    const Izraz &operand_2) : operacija(operacija),
    operand_1(operand_1), operand_2(operand_2) {}
  friend class Izraz;
};
```

Pri površnom posmatranju, čini se da ovom modifikacijom nismo dobili ništa posebno, s obzirom da na prvi pogled izgleda da objekti tipa "Izraz" ne sadrže ništa drugo osim pokazivača na objekat tipa "ApstraktniCvor", što smo imali i ranije. Međutim, suština je u tome da objekti tipa "Izraz" sadrže u sebi i *mehanizam za upravljanje memorijom* (iskazan kroz prisustvo konstruktora, destruktora, konstruktora kopije i destruktora). Ova na prvi pogled sitna izmjena će drastično pojednostaviti pisanje destruktora klase "Izraz". Druga neočekivana posljedica ove izmjene je da objekti tipa "CvorUnarni" i "CvorBinarni" neće morati imati niti destruktor niti konstruktor kopije (a ni preklopljeni operator dodjele), jer će se o njihovom ispravnom uništavanju odnosno kopiranju brinuti destruktor i konstruktor kopije klase "Izraz". Zapravo, razlika između upotrebe pokazivača na objekat tipa "ApstraktniCvor" i upotrebe objekta tipa "Izraz" je upravo razlika između modeliranja grane stabla običnim pokazivačem i nekim "inteligentnijim" objektom koji vodi računa o upravljanju memorijom. Stoga, o klasama "CvorUnarni" i "CvorBinarni" više uopće ne moramo misliti kao o klasama koje sadrže pokazivače (mada sadrže objekte tipa "Izraz" koji interno u sebi sadrže pokazivače), slično kao što kada neka klasa

sadži objekte tipa "string" ne moramo misliti o tome da ti objekti u sebi interno sadrže pokazivače, s obzirom da se o upravljanju memorijom brinu mehanizmi ugrađeni u samu klasu "string".

Kada smo vidjeli kako klase "CvorAtom", "CvorUnarni" i "CvorBinarni" tačno izgledaju, implementacije konstruktora klase "Izraz" postaju trivijalne:

```
Izraz::Izraz(double vrijednost) :
  pok_korijen(new CvorAtom(vrijednost)) {}
Izraz::Izraz(const string &operacija, const Izraz &operand) :
  pok_korijen(new CvorUnarni(operacija, operand)) {}
Izraz::Izraz(const Izraz &operand_1, const string &operacija,
  const Izraz &operand_2) : pok_korijen(new CvorBinarni(operand_1,
  operacija, operand_2)) {}
```

Zahvaljujući činjenici da smo se odlučili za plitko kopiranje, konstruktor kopije je također trivijalan. Sve što treba uraditi je kopirati pokazivač na korijen stabla, i uvećati njegov brojač pristupa za jedinicu:

```
Izraz::Izraz(const Izraz &e) : pok_korijen(e.pok_korijen) {
  (pok_korijen->brojac)++;
}
```

Što se tiče destruktora, on na prvi pogled djeluje kompliciran. Izgleda da bi trebalo obići čitavo stablo (odnosno dag) počev od korijena kroz sve čvorove, zatim za svaki od čvorova umanjiti njihov brojač pristupa za jedinicu, uz brisanje onih čvorova čiji je brojač dostigao nulu. Tako bismo zaista morali raditi kada bismo operande čvorova čuvali kao obične pokazivače i kada čvorovi ne bi sadržavali nikakav mehanizam za upravljanje memorijom. Međutim, sada je taj postupak dovoljno obaviti *samo nad pokazivačem na korijen*! Zaista, pretpostavimo da je došao red na brisanje korijena. Destruktor će tada svakako izvršiti operaciju "**delete** pok_korijen". Operacija "**delete**" će, između ostalog, pozvati destruktor objekta na koji pokazivač "pok_korijen" pokazuje, ako takav postoji. Na prvi pogled, objekti na koje pokazivač "pok_korijen" pokazuje nemaju destruktor. No, ne smijemo zaboraviti da neki od njih sadrže atribute tipa "Izraz", tako da će pri brisanju tih objekata biti pozvani destruktori klase "Izraz" za sve atribute koji su tog tipa. Ti destruktori će dalje ponovo obaviti *istu stvar nad pojedinim operandima* kao što je izvršeno i nad korijenom i tako rekurzivno kroz čitavo stablo! Dakle, ono što bismo inače morali implementirati "ručno", sada se dešava automatski. Stoga, implementacija destruktora također postaje krajnje jednostavna:

```
Izraz::~Izraz() {
  if(--(pok_korijen->brojac) == 0) delete pok_korijen;
}
```

Ostaje još operator dodjele koji kombinira tehnike iz destruktora i konstruktora kopije. Naime, on treba da izvrši eventualnu "destrukciju" objekta sa lijeve strane (ukoliko je potrebno), a zatim da izvrši kopiranje pokazivača, uz uvećanje brojača pristupa:

```
Izraz &Izraz::operator =(const Izraz &e) {
  (e.pok_korijen->brojac)++;
  if(--(pok_korijen->brojac) == 0) delete pok_korijen;
  pok_korijen = e.pok_korijen;
  return *this;
}
```

Mukotrpni posao dizajniranja tražene klase "Izraz" je konačno gotov. Dizajnirana klasa ispunjava sve postavljene uvjete postavljene u postavci problema. Korisnik može kreirati stabla aritmetičkih izraza proizvoljne složenosti i štampati ih, bez ikakvog razmišljanja o upravljanju memorijom i njihovoj internoj organizaciji. Pored toga, objekti tipa "Izraz" mogu se koristiti u svim kontekstima u kojima i objekti ma kojih drugih tipova (prenos u funkcije, vraćanje kao rezultata iz funkcija, itd.) pri čemu se sve manipulacije obavljaju krajnje efikasno, bilo sa aspekta vremena izvršavanja, bilo sa aspekta utroška memorije. Kompletna implementacija ovako izvedene klase, zajedno sa testnim primjerom, data je u programskoj datoteci "izraz2.cpp" priloženoj uz ovo predavanje.

Pokažimo sada da se trud uložen u razvoj ove klase isplati. Na klasu "`Izraz`" postavićemo nove zahtjeve, pri čemu ćemo vidjeti da ti zahtjevi neće zahtijevati izmjenu niti jedne linije već napisanog koda (samo dodavanje novih stvari). Na prvom mjestu, klasa "`Izraz`" je veoma ograničena. Objekti ove klase se mogu kreirati (doduše, na dosta kompleksan način) i ispisivati, ali sa njima ne možemo raditi ništa drugo. Šta se to još prirodno može raditi sa izrazima? Prvo što pada na pamet je *izračunavanje njihove vrijednosti*. Dodajmo sada u klasu "`Izraz`" metodu "`Izracunaj`", koja izračunava vrijednost izraza predstavljenog objektom tipa "`Izraz`". Na primjer, želimo da isječak programa poput

```
Izraz e(Izraz("-", 5), "*", Izraz(3, "+", 4));
cout << e << " = " << e.Izracunaj() << endl;
e = Izraz(e, "*", e);
cout << e << " = " << e.Izracunaj() << endl;
```

proizvede sljedeći ispis na ekranu:

```
((-5)*(3+4)) = -35
(((-5)*(3+4))*((-5)*(3+4))) = 1225
```

Malo razmišljanja dovodi nas do zaključka da izračunavanje treba da radi na način sličan načinu kako je izvedeno ispisivanje. Kada korisnik pozove metodu "`Izracunaj`" nad nekim objektom tipa "`Izraz`", ona treba proslijediti zadatak izračunavanja čvorovima od kojih se stablo tog izraza sastoji. Stoga će svi čvorovi također sadržavati metodu "`Izracunaj`", a poziv ove metode nad objektom tipa "`Izraz`" će prosto biti proslijeđena korijenu stabla. Stoga ćemo klasu "`Izraz`" dopuniti na sljedeći način:

```
class Izraz {
  ...
public:
  ...
  double Izracunaj() const { return pok_korijen->Izracunaj(); }
};
```

Kako se izračunavanje razlikuje za različite vrste čvorova, metoda "`Izracunaj`" u baznoj klasi "`CvorApstraktni`" mora biti virtualna (pri tome nema nikakve potrebe da je u baznoj klasi uopće implementiramo, odnosno ona može biti čisto virtualna metoda):

```
class CvorApstraktni {
  ...
  virtual double Izracunaj() const = 0;
};
```

Naravno, klase koje predstavljaju konkretne vrste čvorova trebaju definirati i konkretne verzije metoda "`Izracunaj`". Situacija je najlakša za čvorove tipa "`CvorAtom`". Naime, "izračunavanje" atoma se svodi prosto na vraćanje njegove vrijednosti kao rezultata:

```
class CvorAtom : public CvorApstraktni {
  ...
  double Izracunaj() const { return vrijednost; }
};
```

Jasno je da metoda "`Izracunaj`" mora postojati i u klasama "`CvorUnarni`" i "`CvorBinarni`". Kako će njihove implementacije biti nešto duže, izvešćemo ih izvan klase, a u klase ćemo ubaciti samo prototipove:

```
class CvorUnarni : public CvorApstraktni {
  ...
  double Izracunaj() const;
};
class CvorBinarni : public CvorApstraktni {
  ...
  double Izracunaj() const;
};
```

Implementacija metode "Izracunaj" za klasu "CvorUnarni" također ne bi trebala biti problem. Sve što treba uraditi je prvo izračunati operand (rekurzivnim pozivom metode "Izracunaj" nad njim), a nakon toga primijeniti operaciju pohranjenu u čvoru nad tim rezultatom. Međutim, mi smo do sada imali mogućnost gradnje stabala aritmetičkih izraza bez obzira na to kakve su operacije u pitanju. Ukoliko želimo vršiti izračunavanje, moramo se odlučiti koje ćemo operacije podržati. Odlučimo se, recimo, da podržimo *unarni minus* "–", koji negira svoj operand, i *logičku negaciju* "!" koja operand različit od nule pretvara u nulu, a operand jednak nuli pretvara u jedinicu. Tada će implementacija metode "Izracunaj" za klasu "CvorUnarni" izgledati ovako:

```
double CvorUnarni::Izracunaj() const {
  double vrijednost(operand.Izracunaj());
  if(operacija == "-") return -vrijednost;
  if(operacija == "!") return (vrijednost == 0) ? 1 : 0;
  throw "Nepoznata unarna operacija!\n";
}
```

Ovdje, poziv "operand.Izracunaj()" računa vrijednost operanda. Sjetimo se da taj poziv zapravo poziva virtualnu funkciju "Izracunaj" nad korijenom podstabla koje "operand" predstavlja, a ona će biti preusmjerena na konkretnu verziju funkcije "Izracunaj" u ovisnosti od toga koje je vrste taj korijen, upravo zahvaljujući činjenici da je ona virtualna. Dakle, ponovo imamo polimorfizam na djelu, u svojoj punoj snazi.

Sličnu stvar imamo u slučaju klase "CvorBinarni". Ovdje treba izračunati vrijednost *oba operanda* i zatim primijeniti operaciju pohranjenu u čvoru nad njima. Odlučimo li se da podržimo četiri osnovne aritmetičke operacije, implementacija bi mogla izgledati recimo ovako:

```
double CvorBinarni::Izracunaj() const {
  double vrijednost_1(operand_1.Izracunaj());
  double vrijednost_2(operand_2.Izracunaj());
  if(operacija == "+") return vrijednost_1 + vrijednost_2;
  if(operacija == "-") return vrijednost_1 - vrijednost_2;
  if(operacija == "*") return vrijednost_1 * vrijednost_2;
  if(operacija == "/") {
    if(vrijednost_2 == 0) throw "Dijeljenje nulom!";
    return vrijednost_1 / vrijednost_2;
  }
  throw "Nepoznata binarna operacija!\n";
}
```

Ovim su sve neophodne dopune gotove, i sada se vrijednosti izraza predstavljenih objektima tipa "Izraz" mogu bez problema izračunavati pozivom metode "Izracunaj", kako je i planirano. Kompletna realizacija ovako proširene klase "Izraz" i odgovarajući testni primjer, priloženi su u programskoj datoteci "izraz3.cpp" datoj uz ovo predavanje.

Prije nego što nastavimo dalje sa novim zahtjevima, rezimirajmo šta smo za potrebe novog zahtjeva (mogućnost izračunavanja izraza) izmijenili u prethodnoj realizaciji klase "Izraz" i, što je možda još važnije, šta nismo trebali mijenjati. Najvažnija je stvar da dodavanje nove funkcionalnosti nije dovelo do potrebe da se išta mijenja u kodu koji je do tada bio napisan, odnosno kodu koji je zadužen za prethodno podržane operacije. S obzirom da prethodno napisana klasa veoma realistično modelira sve elemente stabla aritmetičkih izraza (kako čvorove, tako i veze između njih), dodavanje nove funkcionalnosti kao što je mogućnost izračunavanja izraza izvedeno je uz dodavanje posve male količine novog koda, toliko male da je teško i zamisliti da bi se takva izmjena mogla učiniti sa manje dodatnog koda. Kao i pri izvedbi podrške štampanju izraza, polimorfizam i mehanizam virtualnih funkcija su nam i ovdje bili od velike koristi. Naime, dovoljno je bilo opisati kako se vrši izračunavanje izraza za svaki od mogućih tipova čvorova. Nakon toga se, zahvaljujući polimorfizmu, izbor koja će se od više različitih metoda za izračunavanje pozvati nad nekim konkretnim čvorom vrši automatski, bez ikakve potrebe da programer testira o kojoj se vrsti čvora radi i da na osnovu rezultata tog testiranja "ručno" poziva odgovarajuću metodu.

Vidjeli smo kako ispravno modeliranje struktura podataka i polimorfizam čine jednostavnim dodavanje novih operacija. Sada ćemo postaviti jedan novi, još radikalniji zahtjev za proširenjem klase "`Izraz`". Poznato je da neki programski jezici pored unarnih i binarnih posjeduju i *ternarne operatore*. Na primjer, programski jezici C i C++ posjeduju ternarni operator "`? :`" sa *tri operanda x*, *y* i *z* koji se koristi u obliku "*x* `?` *y* `:` *z*". Postavimo novi zahtjev da klasa "`Izraz`" pored mogućnosti kreiranja izraza sa unarnim i binarnim operatorima, posjeduje mogućnost kreiranja i izraza koji sadrže ternarne operatore. Pored toga, zahtijevaćemo još da za konkretan ternarni operator "`? :`" postoji i podrška izračunavanja izraza koji sadrže taj operator.

Ovaj novi zahtjev je radikalan u smislu da on zahtijeva dodavanje *posve nove vrste čvorova*. Naime, da bismo podržali ternarne operatore, potrebna nam je nova vrsta čvorova koja odgovara ternarnim operatorima i koja ima tri potomka (po jedan za svaki od tri operanda ternarnog operatora). Pokažimo sada kako je, zahvaljujući objektno orjentiranom dizajnu, dodavanje novih vrsta čvorova također posve jednostavno izvodljivo, bez ikakve izmjene koda koji koristi već postojeće čvorove.

Oznaka svih ternarnih operatora sastoji se iz dva dijela (dijela koji razdvaja prvi i drugi operand, kao i dijela koji razdvaja drugi i treći operand). Recimo, oznaka ternarnog operatora "`? :`" sastoji se iz prvog dijela "`?`" i drugog dijela "`:`". Stoga ćemo u klasu "`Izraz`" dodati novi konstruktor koji kreira stablo izraza u čijem se korijenu nalazi ternarni operator. Parametri ovog konstruktora biće dva dijela koja čine oznaku operatora (oba će biti tipa "`string`"), kao i tri parametra koji odgovaraju operandima:

```
class Izraz {
  ...
public:
  Izraz(const Izraz &operand_1, const string &operacija_1,
    const Izraz &operand_2, const string &operacija_2,
    const Izraz &operand_3);
  ...
};
```

Sada ćemo dodati novu klasu "`CvorTernarni`", naslijeđenu iz klase "`CvorApstraktni`", koja modelira čvor koji odgovara ternarnim operatorima. Ova klasa je po svojoj strukturi posve analogna klasama "`CvorUnarni`" i "`CvorBinarni`":

```
class CvorTernarni : public CvorApstraktni {
  string operacija_1, operacija_2;
  Izraz operand_1, operand_2, operand_3;
  ostream &Ispisi(ostream &tok) const {
    return tok << "(" << operand_1 << operacija_1 << operand_2
      << operacija_2 << operand_3 << ")";
  }
  CvorTernarni(const Izraz &operand_1, const string &operacija_1,
    const Izraz &operand_2, const string &operacija_2,
    const Izraz &operand_3) : operacija_1(operacija_1),
    operacija_2(operacija_2), operand_1(operand_1),
    operand_2(operand_2), operand_3(operand_3) {}
  double Izracunaj() const;
  friend class Izraz;
};
```

Nakon što smo vidjeli kako tačno izgleda klasa "`CvorTernarni`", implementacija dodatnog konstruktora klase "`Izraz`" postaje trivijalna:

```
Izraz::Izraz(const Izraz &operand_1, const string &operacija_1,
    const Izraz &operand_2, const string &operacija_2,
    const Izraz &operand_3) : pok_korijen(new CvorTernarni(operand_1,
    operacija_1, operand_2, operacija_2, operand_3)) {}
```

Konačno, potrebno je još implementirati metodu "`Izracunaj`" za klasu "`CvorTernarni`", koja je u opisu klase samo deklarirana. S obzirom da smo rekli da ćemo podržati izračunavanja samo ternarnog

operatora "`? :`" i imajući u vidu značenje ovog operatora u jezicima C i C++, implementacija ove metode mogla bi izgledati recimo ovako:

```
double CvorTernarni::Izracunaj() const {
  if(operacija_1 == "?" && operacija_2 == ":")
    if(operand_1.Izracunaj() != 0) return operand_2.Izracunaj();
    else return operand_3.Izracunaj();
  throw "Nepoznata ternarna operacija!\n";
}
```

To bi bilo sve! Radikalno novi zahtjev koji zahtijeva podršku posve novog tipa operatora u odnosu na ono što je prvobitno bilo zamišljeno riješen je uz svega dvadesetak linija koda. Pri tome, dopune su izvedene ponovo bez ikakvih izmjena u postojećem kodu, prostim uvođenjem nove klase. Štaviše, u postojećim klasama nisu izvršene čak ni nikakve dopune, ukoliko ne računamo dodavanje jednog novog konstruktora u klasu "`Izraz`" (izvršene dopune date su u programskoj datoteci "`izraz4.cpp`" priloženoj uz ovo predavanje). Nije teško prepoznati koje su ključne metodologije koje su omogućile ovako jednostavno rješavanje novopostavljenog zahtjeva: nasljeđivanje i polimorfizam, odnosno upravo metodologije objektno orjentiranog programiranja.

Nije na odmet istaći da su u doba prije uvođenja objektno orjentiranih tehnika programiranja zahtjevi ovog tipa dovodili do drastičnih izmjena u postojećem kodu. Zaista, da nismo koristili nasljeđivanje i polimorfizam, morali bismo ručno ispitivati koja je vrsta nekog čvora i u zavisnosti od toga pozivati odgovarajuće metode. Stoga bi dodavanje nove vrste čvora zahtijevalo intervencije na svim mjestima na kojima ručno ispitujemo vrstu čvora, s obzirom da bi nakon dodavanja nove vrste čvora ispitivanje moralo uključiti i tu novu vrstu. Interesantno je napomenuti da su autorima prvih verzija kompajlera za jezik C koji nisu podržavali ternarni operator "`? :`" bile potrebne sedmice pa čak i mjeseci da dodaju podršku za ovaj operator. Doduše, činjenica je da su kompajleri neuporedivo složeniji programi od programa koji smo upravo razvili, ali su i činjenice da bi dodavanje novog operatora (kakav god on bio) bilo relativno jednostavno u kompajleru koji je dizajniran u skladu sa objektno orjentiranim načelima, kao i da bi dodavanje podrške novim tipovima operatora čak i u relativno jednostavan program poput upravo razvijenog bilo znatno složenije da nismo koristili objektno orjentirane tehnike programiranja. Uostalom, razmislite kako biste uopće riješili problem implementacije klase koja barata sa stablima aritmetičkih izraza ukoliko bismo se ograničili samo na *objektno zasnovane* a ne na *objektno orjentirane* metode, odnosno ukoliko bismo odustali od nasljeđivanja i, pogotovo, od polimorfizma. Naravno, to nije nemoguće izvesti, međutim komplikacije koje bi se javile pri implementaciji bile bi višestruko teže od komplikacija sa kojima smo se susretali. Što je još važnije, svaki novi zahtjev tražio bi ponekad i drastičnu izmjenu već napisanog koda.

Na ovom predavanju smo vidjeli kako objektno orjentirane tehnike programiranja mogu pojednostaviti dizajn i razvoj neke aplikacije. Bit svega je da pokušamo što je god preciznije modelirati objekte iz stvarnog svijeta koje želimo predstaviti u aplikaciji koju razvijamo. Tako, recimo, onog trenutka kada smo spoznali da stabla aritmetičkih izraza kao bitne komponente imaju i *čvorove* i *grane* (odnosno da su bitne i grane, a ne samo čvorovi kako pri površnom modeliranju izgleda), stvari su krenule nabolje, nakon čega smo relativno lako dizajnirali odgovarajuće strukture podataka koje tačno modeliraju ponašanje stabala aritmetičkih izraza. Pri tome, nasljeđivanje je iskorišteno za modeliranje činjenice da razne vrste čvorova imaju *zajedničkih osobina* (ako ništa drugo, zajedničko im je da su svi oni ipak čvorovi, mada različitih vrsta), dok je polimorfizam iskorišten za modeliranje njihovih *različitosti*. Posebno, polimorfizam je omogućio da se načelno iste operacije, poput ispisa i izračunavanja, implementiraju na različit način za različite vrste čvorova, pri čemu se izbor prave implementacije u pravom trenutku vrši automatski, bez potrebe da o tome eksplicitno vodi računa programer. Tako su nam tehnike objektno orjentiranog programiranja omogućile da u jednom trenutku razmišljamo samo o tome kako korektno implementirati klasu koju u tom trenutku razvijamo, bez potrebe da razmišljamo kako će ona sarađivati sa drugim klasama.

Interesantno je napomenuti još jednu činjenicu. Ranije smo isticali da je u jeziku C++ polimorfizam moguće ostvariti samo pomoću pokazivača ili referenci. Mada je to neosporno tačno, interesantno je da klasa "`Izraz`" koju smo razvili predstavlja tipičan primjer izrazito polimorfne klase (samim tim što objekti

tipa "Izraz" mogu sadržavati izraze posve raznorodne strukture), bez obzira što korisnik klase "Izraz" nema potrebe da radi niti sa pokazivačima, niti sa referencama. Suština je u tome da je polimorfno ponašanje klase (koje je, naravno, zasnovano na pokazivačima) *sakriveno unutar implementacije klase*, tako da korisnik klase o tome ništa ne zna (niti treba da zna). Slijedi da su u jeziku C++ pokazivači (ili reference) zaista neophodni za *implementaciju* polimorfnih tipova podataka, ali da se, sa aspekta korisnika, polimorfni tipovi podataka mogu implementirati tako da njihov krajnji korisnik nema nikakve potrebe za korištenjem pokazivača.

Razvijena klasa se, naravno, može i dalje proširivati. Recimo, mogla bi se dodati podrška operatorima koji imaju propratne efekte, poput operatora "++" ili operatora dodjele "=". Za podršku ovakvim operatorima, trebalo bi dodati nove vrste čvorova, između ostalog čvorove koji modeliraju *promjenljive* (a ne samo brojčane konstante). Ovakva modifikacija ostavlja se studentima za samostalnu vježbu. U svakom slučaju, poenta je u tome da se svaka aplikacija prije ili kasnije treba nadograđivati i da nam je cilj da se te nadogradnje mogu izvoditi što je god moguće jednostavnije. U našem primjeru, pošli smo od relativno jednostavnih zahtijeva, za koje smo, nakon djelimičnih početnih promašaja, uspjeli dizajnirati rješenje, za koje smo ustanovili da radi kako treba. Kako se naše potrebe mijenjaju, poželjeli smo dodati nove funkcionalnosti. Ono što je bitno je da se pokazalo da su izmjene koje treba pri tome učiniti *lokalizirane*, tako da je razvoj aplikacije olakšan time što znamo da dodavanje novih funkcionalnosti neće ugroziti rad onoga za šta već znamo da radi, tako da pri svakoj dopuni treba testirati samo rad novododanih stvari, a ne čitave aplikacije ispočetka. Stoga, svaki novi zahtjev traži ulaganje dodatnog truda koji je proporcionalan samo složenosti *novog zahtjeva*, a ne složenosti *čitave aplikacije*, što je obično slučaj kada se ne koriste objektno orjentirane tehnike programiranja.

Na kraju, treba još istaći da nam čak ni objektno orjentirano programiranje ne može pomoći ukoliko tokom razvoja aplikacije dođe ne do potrebe za novim zahtjevima, nego do *izmjene već ranije postavljenih zahtjeva*. U tom slučaju, izmjene već napisanog koda su *neizbježne*. Recimo, ukoliko bismo sada poželjeli da se izrazi ispisuju *bez suvišnih zagrada*, vodeći računa o *prioritetima operacija*, to već predstavlja izmjenu ranije postavljenih zahtjeva, a ne novi zahtjev. Stoga bi to tražilo osjetnu izmjenu postojećeg koda. Recimo, sve operacije ispisa morale bi upoređivati prioritet operacije u čvoru sa prioritetom operacija u njegovim potomcima, sa ciljem da se utvrdi treba li ispisati zagrade ili ne.