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:`

```
Pero Perić
1234
Huso Husić
4132
Marko Marković
3142
Meho Mehić
2341
```

`OCJENE.TXT:`

```
1234 7
4132 8
1234 6
2341 9
1234 10
4132 9
1234 8
```

Naredni program za svakog studenta iz datoteke "`STUDENTI.TXT`" pronalazi njegove ocjene u datoteci "`OCJENE.TXT`", računa njegov prosjek, i ispisuje njegovo ime i izračunati prosjek na ekranu (odnosno komentar "NEOCIJENJEN" ukoliko student nema niti jednu ocjenu, kao što je recimo slučaj sa studentom "Marko Marković" u gore navedenom primjeru). U razmatranom programu se za svakog pročitanog studenta iz datoteke "`STUDENTI.TXT`" vrši prolazak kroz čitavu datoteku "`OCJENE.TXT`" sa ciljem da se pronađu sve njegove ocjene (datoteka "`OCJENE.TXT`" se svaki put iznova otvara pri svakom prolasku kroz vanjsku petlju). Slijedi i kompletan prikaz programa:

```
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

int main() {
  ifstream studenti("STUDENTI.TXT");
  for(;;) {
    char ime[100];
    int indeks;
    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 će 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).