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;
};
```

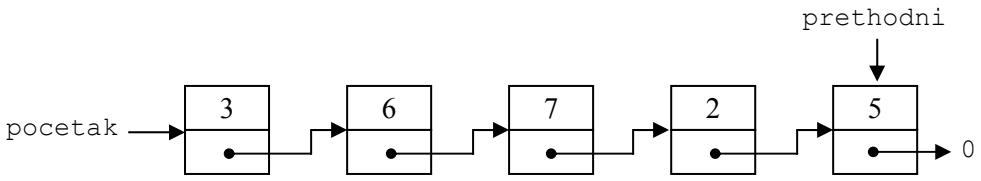
Čvorovi predstavljaju osnovni gradivni element za formiranje tzv. *dinamičkih struktura podataka*, o kojima ćemo govoriti na samom kraju ovog kursa. Na ovom mjestu ćemo ilustrirati samo osnovnu ideju koja ilustrira smisao čvorova. Pretpostavimo da želimo unijeti i zapamtiti slijed brojeva koji se završava 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:

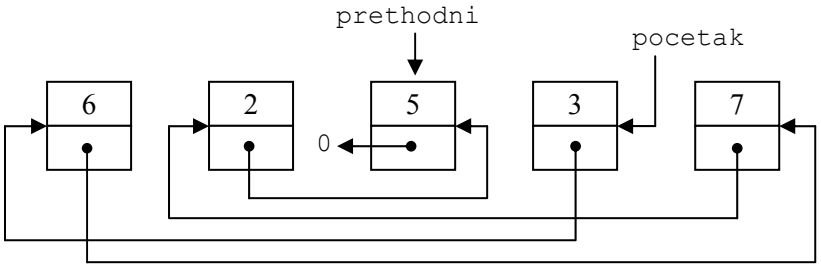
```
cout << pocetak->element;
cout << pocetak->veza->element;
cout << pocetak->veza->veza->element;
```

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).