ZBIRKA ZADATAKA

Mr Vesna Popović

Prof. dr Igor Đurović

1. Razlike C i C++-a neobjektne prirode

Zadatak 1. 1

Implementirati funkciju max koja daje maksimalni element cjelobrojnog niza, maksimum dva cijela broja, maksimalni od dva stringa, "maksimalni" od dva karaktera, za jedan string karakter sa najvećom vrijednošću unutar stringa.

```
#include<iostream.h>
#include<string.h>

int max(int *,int);
inline int max(int,int);
char * max(char *, char *);
inline char max(char,char);
char max(char*);

main(){
int a[]={1,5,4,3,2,6},n=6;
cout<<max(a,n)<<endl;
cout<<max("string","program")<<endl;
cout<<max('a','A')<<endl;
cout<<max(2,5)<<endl;
cout<<max("string");
}

int max(int *a,int n){
int m;
m=a[0];
for(int i=1;i<n;i++)
        if(a[i]>m)
                m=a[i];
return m;
}

inline int max(int a,int b){
return a>=b?a:b;
}

char * max(char *a,char *b){
if(strcmp(a,b)>=0)
        return a;
else
        return b;
}

inline char max(char a,char b){
return a>=b?a:b;
}

char max(char *a){
char m;
int i;
m=a[0];
```

```
i=1;
while(a[i]!='\0') {
      if(a[i]>m)
            m=a[i];
      i++;
      }
return m;
}
```

U ovom zadatku je ilustrovana upotreba *inline* funkcija i preklapanje imena funkcija. Na samom početku je potrebno obratiti pažnju na korišćenje biblioteke *iostream.h* kako bi se omogućila upotreba naredbi za unos i očitavanje podataka *cin*>> i *cout*<<, respektivno. Funkcije su realizovane redoslijedom kojim su navedene u postavci zadatka. U samoj deklaraciji funkcija ne uočavaju se razlike u odnosu na programski jezik C, osim kod funkcija za poređenje dva cijela broja i dva karaktera. Ove funkcije su realizovane kao *inline* funkcije što znači da će prilikom svakog pozivanja neke funkcije ista biti ugrađena u kod. Iz same definicije *inline* funkcije se može zaključiti da je potrebno voditi računa o tome koja će funkcija biti realizovana kao takva. Naime, kao *inline* se realizuju samo one funkcije koje su veoma jednostavne te bi vrijeme koje je potrebno za prenos podataka programa na stek, proslijeđivanje stvarnih argumenata funkciji, a nakon izvršavanja programa brisanje podataka koji pripadaju funkciji i uzimanje podataka sa steka, bilo duže od izvršavanja same funkcije. Ključna riječ *inline* ispred zaglavlja funkcije govori kompajleru da je u pitanju *inline* funkcija. Primjećujemo da sve funkcije imaju isto ime, što u programskom jeziku C nije bilo moguće. U C++ je moguće funkcijama koje obavljaju srodne radnje, nad različitim tipovima podataka, dati ista imena. Za svaki tip i broj podataka je potrebno zasebno realizovati funkciju. Ovdje se mora voditi računa da se sve funkcije razlikuju po broju ili tipu argumenata, kako ne bi nastupila dvosmislenost prilikom njihovog pozivanja što bi izazvalo grešku pri kompajliranju, o čemu će biti riječi u nekom od kasnijih primjera.

Prva funkcija, za određivanje maksimalnog elementa niza, je realizovana na isti način kao što bi se to odradilo u programskom jeziku C. Niz je proslijeđen preko pokazivača, kako se radi o nizu cijelih brojeva proslijeđen je i broj elemenata niza. U funkciji se prvi element niza proglašava za najveći, zatim se, u *for* petlji, vrši pretraživanje niza od drugog do poslednjeg elementa. Svaki element se poredi sa trenutno maksimalnim i ukoliko je veći od njega proglašava se maksimalnim. Kao rezultat se vraća vrijednost promjenjive *m* u kojoj je smještena vrijednost maksimalnog elementa niza. Treba obratiti pažnju na zaglavlje *for* petlje, u njemu postoji jedna razlika u odnosu na programski jezik C. Naime, u C++ je moguće izvršiti definiciju promjenjive u okviru prvog izraza u naredbi *for*. Doseg tako uvedene promjenjive je do kraja tijela te *for* petlje {}. Nakon njenog izvršavanja, tako definisana promjenjiva više ne postoji.

Druga funkcija, koja daje maksimum dva cijela broja, je realizovana kao *inline*. Kompajler to zna zbog navođenja ključne riječi *inline* ispred standardnog zaglavlja ove funkcije. Za njenu realizaciju smo koristili ternarni operator *? :* koji ispituje da li je *a* veće od *b*, ako jeste kao rezultat funkcije se vraća *a* a u suprotnom *b*.

Treća funkcija upoređuje dva stringa i vraća pokazivač na veći od njih. Kao i u programskom jeziku C, i ovdje je string realizovan kao niz karaktera. Poređenje se vrši funkcijom *strcmp()* čija se upotreba ne razlikuje u odnosu na programski jezik C a omogućena je uključivanjem bibloteke *string.h.*

Funkcija za poređenje dva karaktera je, zbog svoje jednostavnosti, realizovana kao *inline*.

Narednom funkcijom se traži karakter sa najvećom vrijednošću u stringu. Ovdje je iskorišćena činjenica da se string realizuje kao niz karaktera i da se kraj tog niza označava kao *'\0'*, tako da funkciji nije potrebno proslijeđivati dužinu niza već se u njoj brojač *i* postavlja na 0 i inkrementira u petlji sve dok ne dođe do kraja niza *'\0'*.

Primjer rada programa

6
string
a
5
t

Protumačiti za koji poziv u glavnom programu će se izvršiti koja funkcija. Vidimo da sve funkcije izvršavaju blisku radnju, traženje maksimuma, i da je samimim tim preklapanje imena funkcija potpuno opravdano.

Zadatak 1.2

Postoje najave funkcija:
int f(int,char='0');
int f(char,char);
int f(double);
Da li će sledeći pozivi izazvati sintaksu grešku i koja će funkcija biti izvršena?

b=f('0');
b=f(2.34);
c=f('c',3);

Obrazložiti.

Uočimo da i ovdje sve funkcije imaju isto ime. Iskoristili smo još jednu novinu programskog jezika C++, a to je da funkcije mogu imati podrazumijevane vrijednosti za neke formalne argumente. Ovo znači da prvu funkciju iz našeg primjera možemo pozvati sa samo jednim stvarnim argumentom, pri čemu će se za vrijednost drugog argumenta uzeti podrazumijevana vrijednost. Ukoliko navedemo dva argumenta u pozivu funkcije, za vrijednost drugog argumenta se uzima navedena, a ne podrazumijevana vrijednost. S obzirom da u našem programu sve funkcije imaju isto ime, na osnovu ranije navedenog, znamo da će se razriješavanje prilikom poziva funkcije izvršiti na osnovu tipova stvarnih argumenata. Posmatramo kojoj funkciji najbolje odgovaraju argumenti za koje je pozvana.

Prvi poziv kao stvarni argument ima tip *char*. Prvi argumet u drugoj funkciji jeste tipa *char* ali se ona ne može izvršiti za ovaj poziv, jer se funkcija poziva sa samo jednim stvarnim argumentom a ona očekuje dva, jer drugi nema podrazumijevanu vrijednost. Za jedan argument se mogu pozvati prva i treća funkcija. S obzirom da postoji standardna konverzija iz tipa *char* u tip *int* izvršiće se prva funkcija. Poziv će, zapravo, biti *f('0','0')*.

Drugim pozivom će se, bez sumnje, izvršiti poslednja funkcija. Naime, jedino ova funkcija ima *float* argument.

Kod trećeg poziva je veći problem jer i prva i druga funkcija mogu da odgovaraju prvom argumentu (bolje odgovara druga funkcija) ali drugi argument ne odgovara ni jednoj funkciji tako da će doći do greške u programu.

Zadatak 1. 3

Dat je niz cjelobrojnih elemenata. Kreirati funkciju koja po defaultu obrće elemente niza "naopačke", a ako je spoecificiran cijeli broj veći od nula formira novi niz koji predstavlja elemente starog niza od prvog sa datim preskokom do kraja, a za negativan broj ide od posljednjeg elementa niza ka početku sa odgovarajućim preskokom.

```
#include<iostream.h>
#include<conio.h>

int uredi(int *, int,int *,int=-1);

main(){
int a[]={2,3,2,1,5},c;
int i, b=5, * r;
r=new int[b];
c=uredi(a,b,r);
for(i=0;i<c;i++)
	cout<<r[i];
cout<<endl;
c=uredi(a,b,r,2);
for(i=0;i<c;i++)
	cout<<r[i];
cout<<endl;
c=uredi(a,b,r,-2);
for(i=0;i<c;i++)
	cout<<r[i];
cout<<endl;
delete []r;
r=0;
getch();
}

int uredi(int *a, int n,int *c, int m){
int j=0;
if(m>0)
	for(int i=0;i<n;j++,i+=m)
		c[j]=a[i];
else
	for(int i=n-1;i>=0;j++,i+=m)
		c[j]=a[i];
return j;
}
```

Treba obratiti pažnju da se i u ovom zadatku, kao i u prethodnom, podrazumijevani argumenti nalaze na kraju. Naime, ukoliko imamo podrazumijevanu vrijednost formalnog argumenta, nakon njega smije biti samo argument sa podrazumijevanom vrijednošću. Ovo je i logično, jer će se prilikom poziva funkcije stvarni argumenti dodjeljivati formalnim onim redoslijedom kojim su ispisani. Prilikom odlučivanja koju vrijednost da uzmemo kao default, iskoristili smo činjenicu da obrtanje elemenata niza "naopačke", u stvari, jeste formiranje novog niza od poslednjeg ka prvom elementu sa preskokom 1, minus govori da se počne od kraja niza.

Funkcija za preuređivanje je realizovana tako što joj se šalje pokazivač na niz koji treba preurediti, broj njegovih elemenata i pokazivač na početak niza u koji ćemo smjestiti preuređene

elemente. Kao rezultat se vraća broj elemenata novonastalog niza. U glavnom programu se vrši dinamičko zauzimanje memorije, na isti način kao ranije. U funkciji se postavlja brojač *j* na nulu, zatim se provjerava da li treba uzimati elemente od početka (*m>0*) ili kraja niza (*m<0*). U oba slučaja se za preskok koristi naredba *i+=*. U prvom slučaju će to povećavati indeks *i* za *m* a u drugom umanjivati jer je *m* negativno. *j* predstavlja broj elemenata novog niza.

U ovom programu se koristi naredba *cout<<endl*, koja predstavlja prelazak u novi red. Može se, i dalje, koristiti *cout<<'\n'*.

Obratite pažnju da je vrijednost podrazumijevanog argumenta navedena samo u zaglavlju funkcije prilikom njegove deklaracije. Naime, podrazumijevana vrijednost se ne može redefinisati, čak ni u istu vrijednost. Ovo znači da, kada imamo odvojeno deklaraciju i definiciju funkcije, podrazumijevanu vrijednost navodimo samo u deklaraciji a ne i u kompletnoj definiciji funkcije.

U našem primjeru se vrši dinamička alokacija memorije. Za to smo koristili operator *new*. *r=new int[b];* govori kompajleru da želimo da zauzmemo memoriju u koju može da se smjesti niz od *b* elemenata cjelobrojnog tipa, kao i da vrijednost pokazivača na početak te memorije smještamo u promjenjivu *r*. Na kraju programa je potrebno osloboditi memoriju koju je niz zauzimao. Za to koristimo operator *delete* (*delete []r;*). Uglaste zagrade prije pokazivača *p* se koriste uvijek kada imamo niz, tako govorimo da je potrebno osloboditi memoriju koju je zauzeo cio niz a ne samo prvi element. Nakon ovog, pokazivač *r* će i dalje sadržati adresu prvog elementa niza čiji smo sadržaj upravi izbrisali. To nikako nije poželjno jer nam pokazivač pokazuje na nešto što više ne postoji. Naredbom *r=0* ostavljamo ga u ispravnom, praznom, stanju. Ne ukazuje ni na jednu memorijsku adresu. Kada bismo u glavnom programu zahtjevali da se prvo stvara novi niz sa preskokom 2 od početka, a nakon toga sa preskokom 3 od kraja, ukoliko isčitamo sve elemente drugog niza (ima ih 3) u drugom slučaju bi kao treći imali element 5, koji ne bi trebalo da bude dio tog niza. Zašto je došlo do toga i kako to da eliminišemo? Zašto smo u funkciji *uredi(...)* definisali promjenjivu *j* van zaglavlja *for* petlje? Da smo je deklarisali kada i promjenjivu *i* da li bi nam nakon izvršavanja *for* petlje bila sačuvana vrijednost koja se nalazi u njoj? Da li bi ta promjenjiva bila definisana nakon izvršavanja petlje.

Primjer rada programa

51232
225
522

Zadatak 1. 4

Napisati program za računanje težine tijela. Ako je zadat jedan argument računati težinu na površini zemlje Q=m*9.81, ako su zadata dva argumenta računati na visini r iznad zemlje: $Q=6.672*10^{-11}*m*4.9156*10^{24}/(6371377+r)^2$; a ako je zadato m, r i M težinu na rastojanju r od planete mase M: $Q=6.672*10^{-11}*m*M/r^2$.

```
#include<iostream.h>

float tezina(int);
float tezina(int,int);
float tezina(int,int,int);

main(){
int n, m, M, r;
cout<<"Koliko argumenata zelite da unesete?"<<endl;
```

```
cin>> n;
if(n==1){
	cin>>m;
	cout<<tezina(m)<<endl;
	}
else if(n==2){
	cin>>m>>r;
	cout<<tezina(m,r)<<endl;
	}
else {
	cin>>m>>r>>M;
	cout<<tezina(m,r,M)<<endl;
	}
}

float tezina(int a){
const float k=9.81;
return a*k;
}

float tezina(int a, int b){
const float k=6.672*pow(10,-11)*4.9156*pow(10,24), k1=6371377;
return k*a/pow((k1+b),2);
}

float tezina(int a, int b, int c){
const float k=pow(10,-11)*6.672;
return k*a*c/pow(b,2);
}
```

Ovdje se, na samom početku, mora zapaziti da je nemoguće uspostaviti vezu između ova tri načina za računanje težine tijela, tako da realizacija sa podrazumijevanim parametrima odmah otpada. Funkcije obavljaju srodnu aktivnost pa će se iskoristiti mogućnost preklapanja imena. Što se tiče same realizacije pojedinih funkcija, ne postoji razlika u odnosu na programski jezik C.

Primjer rada programa

Koliko argumenata zelite da unesete?	*Koliko argumenata zelite da unesete?*
1	*2*
60	*60*
588.6	*2*
	484.74

---

2. Klase

Zadatak 2. 1

Projektovati klasu sa elementarnim operacijama nad kompleksnim brojevima (sabiranje, oduzimanje).

```
# include<iostream.h>

class complex{
        float imag, real;
  public:
        void cUcitaj(float, float);
        complex cSab(complex);
        complex cOduz(complex);
        float cRe()const {return real;};
        float cIm()const {return imag;};
};

void complex::cUcitaj(float a, float b){
      real=a;
      imag=b;
}

complex complex::cSab(complex a){
      complex priv;
      priv.real=real+a.real;
      priv.imag=imag+a.imag;
      return priv;
}

complex complex::cOduz(complex a){
      complex priv;
      priv.real=real-a.real;
      priv.imag=imag-a.imag;
      return priv;
}

main(){
complex c1, c2, c3;
float a, b;
cout<<"Unesi vrijednost za realni i imaginarni dio c1"<<'\n';
cin>>a>>b;
c1.cUcitaj(a,b);
cout<<"Unesi vrijednost za realni i imaginarni dio c2"<<'\n';
cin>>a>>b;
c2.cUcitaj(a,b);
c3=c1.cSab(c2);
cout<<"Zbir dva data kompleksna broja je"<<"("<<c3.cRe()<<","\
<<c3.cIm()<<")"<<endl;
c3=c1.cOduz(c2);
```

```
cout<<"Razlika dva data kompleksna broja je"<<"("<<c3.cRe()<<","\
<<c3.cIm()<<")"<<endl;
}
```

U ovom zadatku je prvo izvršena definicija klase. Na početku ove definicije se nalazi rezervisana riječ *class*, nakon ove riječi pišemo identifikator, odnosno ime klase. Između vitičastih zagrada se nalazi deklaracija promjenjivih i funkcija, a može biti i definicija neke funkcije. Obavezno je nakon vitičastih zagrada staviti *;* jer to govori kompajleru da smo završili definiciju klase i da nakon toga možemo definisati pojedine funkcije članice. Koji će članovi klase i funkcije biti vidljivi van klase određuju rezervisane riječi *public* i *private*. Ukoliko ih nema podrazumijeva se da su svi podaci članovi i funkcije članice privatni. Praksa je da se podaci članovi postavljaju za privatne a funkcije članice za javne. U ovom primjeru je tako učinjeno. Iz primjera se vidi da će ako se ne navede ključna riječ *private*, početni dio klase, prije prve rezervisane riječi *public*, biti privatan. Dobra programerska praksa je da se uvijek eksplicitno navede koji je dio klase privatan. Klasa, zapravo, predstavlja korisnički definisan tip podataka. Isto kao za ugrađene tipove podataka i za ovaj, korisnički definisan, možemo uvoditi instance, primjerke tog tipa. Primjerak klase je objekat i dekleriše se isto kao i primjerak ugrađenog tipa, navođenjem identifikatora tipa, *complex,* a nakon njega identifikatora za konkretan primjerak tog tipa *complex c1,c2,c3;*. Mi ovdje želimo definisati tip podataka koji će imati osobine kompleksnih brojeva. Znači, želimo omogućiti dodjeljivanje vrijednosti kompleksnom broju, očitavanje njegovog realnog i imaginarnog dijela, kao i sabiranje dva kompleksna broja. S obzirom da smo promjenjive koje predstavljaju realni i imaginarni dio kompleksnog broja realizovali kao privatne podatke članove, za izvršavanje ranije navedenih operacija potrebno je realizovati odgovarajuće funkcije članice klase, one imaju pravo da pristupaju podacima članovima iste klase koji su privatni. U našem programu imamo 6 funkcija članica. Prva služi za inicijalizaciju objekta klase *complex* (primjeraka klase) i ona će dodijeliti odgovarajuće vrijednosti podacima članovima objekta za koji je pozvana. Vidimo da funkcije članice pored svojih formalnih argumenata sadrže i podrazumijevani argument a to je objekat za koji su pozvane (odnosno pokazivač na taj objekat). Članovima tog objekta može da se pristupa navođenjem samo identifikatora (imena) podatka člana. Naime, u ovoj funkciji smo dodjeljivanje vrijednosti realnom dijelu objekta, za koji ćemo je pozvati, izvršili naredbom *real=a;* (na isti način smo odradili i za imaginarni dio). Sada pogledajmo što će se, zapravo, desiti kada u glavnom programu pozovemo funkciju *c1.cUcitaj(a,b)*;. Mi smo ovu funkciju pozvali za konkretan objekat *c1* pa će se pristupati njegovom podatku članu *real.* Dakle, funkcija može direktno, bez navođenja imena objekta kojem pripadaju, da pristupa članovima objekta za koji je pozvana. Iz tog razloga kada je potrebno obaviti neku radnju samo sa jednim objektom nema potrebe da se šalje taj objekat kao argument funkcije, funkcija može da pristupa njegovim podacima direktno. Ukoliko želimo da obavimo neku radnju nad više objekata uvijek šaljemo za jedan manje, jer će objekat za koji pozivamo funkciju biti dostupan samim pozivom te funkcije a ostale moramo poslati kao argumente funkcije. Funkcija *cSab()* vrši sabiranje dva kompleksna broja. Rezultat je tipa *complex* jer je zbir dva kompleksna broja takođe kompleksan broj. Sada je kao argument potrebno poslati jedan objekat (jedan kompleksan broj) dok će se članovima drugog, za koji je funkcija pozvana, moći pristupiti direktno. Naime, naredbom *c3=c1.cSab(c2);* (u glavnom programu) vršimo sabiranje dva kompleksna broja *c1* i *c2*, pri čemu smo *c2* poslali kao stvarni argument a podacima objekta *c1* smo mogli direktno pristupiti, pokazivač na njega *this* je implicitni argument ove funkcije. Obratiti pažnju da argumenti kao i rezultati funkcije članice mogu biti tipa *complex.*

Kada se vrši realizacija funkcije izvan definicije klase, klasa kojoj pripada funkcija se mora naglasiti u zaglavlju funkcije navođenjem imena klase nakon kojeg imamo operator dosega *::*. Na ovaj način kompajler zna da je funkcija čije ime navodimo poslije *::* članica date klase. Ukoliko je neka funkcija realizovana u okviru definicije klase biće implicitno *inline*. Kod nas su to funkcije *cRe()* i *cIm().* Vidimo da će ove funkcije vratiti vrijednost realnog i imaginarnog

dijela kompleksnog broja (objekta) za koji su pozvane. (Napomenimo, još jednom, da se ovo mora učiniti na ovaj način jer ukoliko nam bude potreban realan dio objekta *c3* u glavnom programu mu ne možemo pristupiti naredbom *c3.real*; jer je *real* privatan podatak klase *complex* i njemu ne možemo pristupati direktno.). Nakon navođenja liste parametara ovih funkcija imamo rezervisanu riječ *const* što znači da je data funkcija *inspektor*, ne mjenja stanje objekta klase, dok *mutatori* mijenjaju. Realizacija funkcije *cOduz()* je izvršena veoma slično kao i funkcije za sabiranja pri čemu je praćena logika oduzimanja kompleksnih brojeva jer je nama cilj da modeliramo kompleksne brojeve.

U glavnom programu, prilikom očitavanja rezultata sve naredne nisu stale u jedan red pa smo ih prebacili u drugi iz pomoć \. Ovim je kompajleru dato na znanje da sve naredbe koje slijede nakon ovog znaka pripadaju prethodnom redu. Važno je zapamtiti da ukoliko imamo funkciju bez argumenata koje joj eksplicitno šaljemo, kao kod nas *cIm()* i *cRe(),* moramo, uvijek, prilikom njihovog pozivanja navoditi zagrade poslije identifikatora te funkcije, jer ukoliko bismo napisali *cRe* a ne *cRe(),* kompajler bi mislio da nam je to identifikator za promjenjivu a ne za funkciju i javio bi grešku.

Primjer rada programa

```
Unesi vrijednost za realni i imaginarni dio c1
1 2.5
Unesi vrijednost za realni i imaginarni dio c2
-3 8.9
Zbir dva data kompleksna broja je(-2,11.4)
Razlika dva data kompleksna broja je(4,-6.4)
```

Zadatak 2.2

Projektovati klasu sa elementarnim operacijama nad tačkama u ravni (računanje udaljenosti zadate tačke od koordinatnog početka, rastojanja dvije tačke). U glavnom programu unijeti podatke za dvije tačke a dati na izlazu koordinate tačke koja je na većem rastojanju od koordinatnog početka i rastojanje te dvije tačke. Koordinate tačke su privatni članovi klase.

```
#include<iostream.h>
#include<math.h>

class tacka{
      double x, y;
  public:
      void pravi(double a, double b){x=a;y=b;}
      double aps()const{return x;}
      double ord() const{return y;}
      double poteg()const;
      double rastojanje(tacka) const;
      void ispisi() const;
      };

double tacka::poteg()const{
return sqrt(pow(x,2)+pow(y,2));
}
```

```
double tacka::rastojanje(tacka a)const{
      return sqrt(pow(x-a.x,2)+pow(y-a.y,2));
}

void tacka::ispisi()const{cout<<"("<<x<<","<<y<<")"<<'\n';}

void main(){
int n;
tacka t1;
cout<<" Unesite koordinate prve tacke "<<endl;
double x, y;
cin>>x>>y;
t1.pravi(x,y);
tacka t2;
cout<<" Unesite koordinate druge tacke "<<endl;
cin>>x>>y;
t2.pravi(x,y);
cout<<"Koordinate dalje tacke su: ";
t1.poteg()>t2.poteg()?t1.ispisi():t2.ispisi();
cout<<"Rastojanje zadatih tacaka je: "<<t1.rastojanje(t2)<<'\n';
}
```

U ovom primjeru želimo da definišemo korisnički tip *tačka* koji će imati neke od osobina tačaka u Dekartovom koordinatnom sistemu. Zaglavlje za matematičku biblioteku **<math.h>** je uključeno jer se koriste matematičke funkcije. Klasa *tacka* ima dva privatna podatka člana tipa *double*. Rad sa ovim podacima omogućen je funkcijama članicama. Iako u postavci zadatka nije eksplicitno navedeno da je potrebno praviti prve tri funkcije, morali smo ih formirati kako bismo mogli zadovoljiti zahtjeve iz glavnog programa. Ne možemo pristupiti privatnim podacima iz glavnog programa neposredno, već je potrebno za to imati funkciju koja ima pravo pristupa. U ovom slučaju je to funkcija članica *pravi()*, koja zapravo inicijalizuje podatke članove na željenu vrijednost. Ova funkcija je definisana u okviru definicije klase pa će biti realizovana kao *inline*, što je i opravdano s obzirom da nije pretjerano komplikovana, svega dvije mašinske instrukcije. Kao argument joj šaljemo dva podatka tipa *double*, koja će ona dodijeliti podacima *x* i *y*, objekta za koji je budemo pozvali u glavnom programu *t1* i *t2*. Rezultat joj je *void* to jest ne vraća ništa, iako je, zapravo, rezultat objekat za koji je pozvana sa novododijeljenim vrijednostima. Ovo je i logično ako se posjetimo da svaka funkcija članica ima sakriveni argument *this* koji je pokazivač na objekat za koji je pozvana pa, kao što ga nije potrebno slati, nije ga potrebno ni vraćati. Sve će se radnje nad podacima članovima objekta za koji je funkcija pozvana automatski odraziti na njih jer je ovaj objekat, implicitno, poslat po referenci. Funkcije *aps()* i *ord()* nam daju podatak o apscisi i ordinati odgovarajuće tačke, objekta za koji su pozvane. Obratiti pažnju na rezervisanu riječ *const* u zaglavlju nekih funkcija članica. Podsjetiti se iz prethodnog primjera šta ona znači. Funkcija *poteg()* računa rastojanje tačke od koordinatnog početka po poznatoj formuli i vraća ga kao rezultat funkcije tipa *double*. Funkcija *rastojanje()* računa rastojanje dvije tačke, zadate koordinatama *x* i *y*. Jednu tačku šaljemo kao argument funkcije a elementima druge možemo direktno pristupiti, tačka za koju pozivamo funkciju. Ovdje smo realizovali i funkciju *ispisi()* za ispisivanje koordinata tačaka u formatu koji se najčešće koristi u matematici a(x,y). U principu bi i bez ove funkcije mogli ispisati koordinate odgovarajuće tačke ali bi tada morali pozivati dvije funkcije i stalno voditi računa o formatiranju pa je preporučljivo uvijek imati funkciju za ispis podataka klase koju formiramo. Ova funkcija ne vraća rezultat.

U glavnom programu smo uveli (deklarisali) objekat *t1* klase *tacka*, zatim smo, nakon unošenja koordinata tačaka u dvije promjenjive tipa *double*, poslali te dvije promjenjive kao

argumente funcije *pravi()* naredbom *t1.pravi(x,y);*. Rekli smo kompajleru da se u ovoj funkciji direktno pristupa podacima članovima objekta *t1*. Na isti način smo formirali objekat *t2* i dodijelili mu vrijednost. Naredba *t1.rastojanje(t2)* šalje funkciji kao stvarni argument objekat *t2* dok se podacima objekta *t1* direktno pristupa. Koristeći odgovarajuću funkciju članicu ispisali smo koordinate tačke koja je udaljenija od koordinatnog početka.

Primjer rada programa

Unesite koordinate prve tacke
2.3 4
Unesite koordinate druge tacke
3 1.2
Koordinate dalje tacke su: (2.3,4)
Rastojanje zadatih tacaka je: 2.88617

Zadatak 2.3

Prepraviti zadatak 5 tako da sadrži konstruktore.

```
# include<iostream.h>

class complex{
      float imag, real;
  public:
      complex(){};
      complex(float,  float);
      complex cSab(complex);
      complex cOduz(complex);
      float cRe()const {return real;};
      float cIm()const {return imag;};
};

complex::complex(float a,  float b):real(a),imag(b){}

complex complex::cSab(complex a){
      complex priv;
      priv.real=real+a.real;
      priv.imag=imag+a.imag;
      return priv;
}

complex complex::cOduz(complex a){
      complex priv;
      priv.real=real-a.real;
      priv.imag=imag-a.imag;
      return priv;
}

main(){
float a, b;
cout<<"Unesite vrijednost za realni i imaginarni dio c1"<<\n';
```

```
cin>>a>>b;
complex c1(a,b);
cout<<"Unesite vrijednost za realni i imaginarni dio c2"<<'\n';
cin>>a>>b;
complex c2(a,b);
complex c3;
c3=c1.cSab(c2);
cout<<"Zbir dva data kompleksna broja je "<<"("<<c3.cRe()<<","'\
<<c3.cIm()<<")"<<endl;
c3=c1.cOduz(c2);
cout<<"Razlika dva data kompleksna broja je "<<"("<<c3.cRe()<<","'\
<<c3.cIm()<<")"<<endl;
}
```

Kao što se vidi i iz postavke, ova klasa se razlikuje od klase u zadatku 5 samo po tome što sadrži konstruktore. Konstruktor je funkcija članica klase koja se koristi za automatsku inicijalizaciju primjeraka svoje klase u momentima formiranja tih primjeraka. Konstruktor će biti pozvan automatski svaki put kada bude deklarisan objekat određene klase ili kada je kreiran na bilo koji drugi način. Data klasa može imati više konstruktora, bez obzira na to koliko ih ima svi moraju imati isti identifikator (ime) kao i klasa kojoj pripadaju. Primjetimo da se konstruktor razlikuje od ostalih funkcija i po tome što nema označen tip rezultata, to je s toga što on i nema rezultat koji vraća, nije dozvoljena ni upotreba riječi *void* za označavanje tipa rezultata koji se vraća. Ukoliko ne definišemo nijednog konstruktora kompajler će sam definisati podrazumijevanog (default). Međutim, ukoliko definišemo makar jednog, moramo i podrazumijevanog jer bi inače naredbe tipa *complex c3;* koje pozivaju podrazumijevanog konstruktora tj. konstruktora bez argumenata, izazvale grešku u kompajliranju. Podsjetimo se da se bez argumenata može pozvati i funkcija koja ima sve argumente sa podrazumijevanim vrijednostima tako da se default konstruktor može realizovati i na taj način, u tom slučaju ćemo prilikom deklaracije izvršiti i inicijalizaciju datog objekta (dodjeljivanje nekih početnih vrijednosti podacima članovima) dok bi deklaracija podrazumijevanim konstruktorom, koji nije realizovan sa podrazumijevanim vrijednostima argumenata, samo zauzela prostor u memoriji za njegove podatke, bez dodjeljivanja smislenog sadržaja. U zavisnosti od potreba u narednim primjerima ćemo koristiti jedan ili drugi tip realizacije podrazumijevanog konstruktora. I za konstruktore, kao i ostale funkcije članice, važi da će biti realizovani kao *inline* funkcije ukoliko su definisani unutar definicije klase. Ukoliko se definišu van definicije klase potrebno je koristiti ime klase i operator dosega:: kao i kod običnih funkcija članica. U našem slučaju to bi značilo da je potrebno pisati za drugog konstruktora *complex::complex(float a, float b):real(a),imag(b){}*. Prva riječ *complex* ukazuje na to da želimo definisati funkciju članicu te klase a drugo *complex* nakon operatora :: govori da je u pitanju konstruktor (ima isto ime kao klasa kojoj pripada, svojstvo konstruktora). Vidimo da je iskorišćen nov način definicije funkcije, takozvano navođenje inicijalizacione liste. Ova lista se piše odmah nakon liste parametara a kompajleru se najavljuje postavljanjem *:*. Ovdje vidimo da će podatak član *real* biti inicijalizovan vrijednošću *a, real(a)*, na isti način imamo *imag(b).* Iako smo inicijalizaciju izvršili van tijela funkcije, i ova funkcija nema nijednu naredbu u njemu, svaka funkcija mora imati tijelo funkcije. Znači, nakon liste inicijalizatora mora postojati tijelo funkcije iako u njemu nema nijedne naredbe, što znači da moramo postaviti vitičaste zagrade, koje označavaju tijelo funkcije. Konstruktor se može definisati na isti način kao i sve funkcije. U ovom slučaju bi to bilo *complex::complex(float a, float b){real=a; imag=b;},* može se koristiti i kombinacija ova dva načina: na primjer *complex::complex(float a, float b):real(a){imag=b;}*. Moramo se odlučiti samo za jednu od ovih realizacija jer bi ukoliko koristimo više različitih realizacija istog konstruktora u jednoj klasi došlo do greške. Naime, konstruktori su ovdje funkcije sa preklopljenim imenima pa i za njih važi da moraju imati argumente takve da kompajler nedvosmisleno zna za koji poziv koju funkciju da realizuje. Ovdje bi svi konstruktori bili pozvani sa dva argumenta tipa *float* i

kompajler bi javio grešku. Mi ćemo, kad god je to moguće, koristiti novo uvedeni način definisanja konstruktora. On se mora koristiti ukoliko naša klasa posjeduje podatke članove koji su delarisani kao *const*, njima nije moguće dodijeliti neku vrijednost koristeći operator dodjele pa se to mora učiniti na ovaj način, ista stvar važi i kada kao podatak imamo referencu.

Konstruktori se ne pozivaju na isti način kao obične funkcije članice. Pogledajmo naredbu *complex c1(a,b);* u glavnom programu. Ovom naredbom se poziva drugi konstruktor, koji očekuje dva argumenta tipa *float*. Podsjetimo se, neku od funkcija članica bi pozvali narednom, na primjer, *c1.cRe()*. Znači ime objekta, operator . pa ime funkcije. Konstruktor se može pozvati samo prilikom inicijalizacije novog objekta i poziva se tako što pišemo ime novog objekta i u zagradama parametre kojima ćemo ga inicijalizovati, koji se moraju poklapati sa onima koje konstruktor očekuje. Drugi način pozivanja konstruktora bi bio bez navođenja parametara u zagradama *complex c1;* ovom naredbom bi se pozvao podrazumijevani konstruktor koji bi formirao promjenjivu *c1*. Postoji još jedan način pozivanja konstruktora, *complex c1=complex(a,b);* ovakav način pozivanja konstruktora se ne preporučuje jer bi kompajler mogao da stvori privremeni objekat koji bi inicijalizovao pa onda njega kopirao u *c1*, što je nepotrebno produžavanje vremena izvršavanja programa. Postoje neke izuzetne situacije u kojima je ovaj drugi način pozivanja konstruktora bolji, i o tome će biti riječi kasnije. U glavnom programu se za *c1* i *c2* poziva konstruktor sa parametrima tipa *float* dok ze za *c3* poziva podrazumijevani konstruktor. Ostatak programa protumačite sami.

Primjer rada programa

Unesite vrijednost za realni i imaginarni dio c1
2 -3.4
Unesite vrijednost za realni i imaginarni dio c2
1.2 8
Zbir dva data kompleksna broja je (3.2,4.6)
Razlika dva data kompleksna broja je (0.8,-11.4)

Zadatak 2.4

Realizovati klasu Text koja će predstavljati dinamičke znakovne nizove nad kojima je moguće izvršiti sljedeće operacije: izračunavanje dužine znakovnog niza, čitanje i zamjenu zadatog karaktera u nizu. Konstruisati odgovarajuće konstruktore i destruktore za datu klasu.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>

class Text{
  private:
      char *p;
  public:
      Text(){p=0;}//Dodjeljujemo pokazivacu vrijednost 0
      Text(const char *);//da upotrebom destruktora ne bismo brisali
      ~Text();          //pokazivac slucajnog sadrazaja i sam sadrzaj, delete.
      int tLength();
      char tRead(int);
      void tWrite(int, char);
      };
```

```
Text::Text(const char* m){
	p=new char[strlen(m)+1];
	strcpy(p,m);}

Text::~Text(){
	delete []p;
	p=0;
	}

char Text::tRead(int i){
	return p[i];
	}

void Text::tWrite(int i, char c){
	p[i]=c;
	}

int Text::tLength(){
	return strlen(p);}

void main(){
	char *ch={"docar dan"};
	Text T(ch);
	cout<<"Koji karakter zelite da zamijenite?"<<endl;
	int j;
	cin>>j;
	cout<<"Koji karakter zelite da postavite"<<endl;
	char c;
	cin>>c;
	T.tWrite(j,c);
	cout<<"Novi tekst je: "<<endl;
	int l=T.tLength();
	for(int i=0; i<=l;i++) cout<<T.tRead(i);
	cout<<endl;
	getch();
	}
```

Ovaj zadatak je lako realizovati ako se ima na umu sve iz prethodnog. Jedina razlika jeste što nam sad u default konstruktoru postoji naredba za postavljane pokazivača, koji ukazuje na početak niza, na nulu. To radimo jer je moguće da se desi da se nakon deklaracije nekog objekta defalult konstruktorom izvrši brisanje tog objekta. Da nemamo naredbu *p=0;* pokazivaču *p* bi se dodjeljivala neka proizvoljna vrijednost, on bi ukazivao na nešto u memoriji što bi se brisalo, nestankom ovog objekta odnosno pozivanjem destruktora. Sledeći konstruktor ne vrši samo zauzimanje memorije za elemente niza već vrši i njihovu inicijalizaciju, proslijeđen mu je pokazivač na početak znakovnog niza. Ostatak programa protumačite sami.

Primjer rada programa

K*oji karakter zelite da zamijenite?*
2
Koji karakter zelite da postavite

b
Novi tekst je:
dobar dan

Zadatak 2.5

Koristeći klasu tačka (koja će sadržati koordinate tačke i funkciju za računanje rastojanja dvije tačke) konstruisati klasu krug koja će kao članove imati tačku centar i neku tačku sa kruga i funkcije članice za izračunavanje površine i obima kruga.

```
#include<iostream.h>
#include<math.h>

class tacka{
      float x,y;
  public:
      tacka(){}
      tacka(float,float);
      float rastojanje(tacka) const;
      };

tacka::tacka(float a,float b):x(a){y=b;}

float tacka::rastojanje(tacka a)const{
      return sqrt(pow(x-a.x,2)+pow(y-a.y,2));
}

class krug{
   private:
      tacka centar;
      tacka sa_kruga;
   public:
      krug(){}
      krug(float, float, float, float);
      //Moze i na sledeci nacin
      //krug(tacka, tacka);
      float kPovrsina();
      float kObim();
      };

krug::krug(float x1, float x2, float x3, float x4):centar(x1,x2){sa_kruga=tacka(x3,x4);}
      /* za drugi nacin realizacije bi morali imati naredbe
      krug::krug(tacka t1, tacka t2):centar(t1){
      sa_kruga=t2;}; */

float krug::kPovrsina(){
            float r;
            const float pi=3.14159;
            r=centar.rastojanje(sa_kruga);
            return pow(r,2)*pi;
```

```
        }

float krug::kObim(){
        float r;
        const float pi=3.14159;
        r=centar.rastojanje(sa_kruga);
        return 2*pi*r;
        }

void main(){
    cout<<"Unesite koordinate centra kruga i tacke sa kruga"<<'\n';
    float x, y, x1,y1;
    cin>>x>>y>>x1>>y1;
    /*za drugi nacin realizacije bi morali imati naredbe
    tacka t1(x,y),t2(x1,y1);
    krug k1(t1,t2);*/
    krug k1(x,y,x1,y1);
    cout<<"Povrsina kruga je: "<<k1.kPovrsina()<<endl;
    cout<<"Obim kruga je: "<<k1.kObim();
}
```

U ovom primjeru je pokazano da podatak član klase može biti i objekat druge klase. U našem slučaju podatak klase *krug* je objekat klase *tacka.* Da bi to mogli realizovati potrebno je prije definicije klase (*krug*), koja posjeduje objekat neke druge klase (*tacka*), izvršiti, makar, deklaraciju te klase u našem slučaju je potrebno imati naredbu tipa *class tacka;* kako bi kompajler znao da je podatak član nove klase korisnički izvedenog tipa. Mi smo izvršili odmah kompletnu definiciju klase *tacka*. Ova klasa ima default konstruktora i konstruktora koji ima dva argumenta tipa *float* što je i logično jer ćemo njihove vrijednosti dodijeliti podacima članovima ove klase koji su istog tipa. Klasa posjeduje i funkciju za računanje rastojanja dvije tačke. Klasa *krug* posjeduje, kao što je zahtjevano postavkom zadatka, dva podatka tipa *tacka* koji predstavljaju koordinate centra kruga i tačke sa kruga, kao i funkciju za računanje površine i obima kruga. Potrebno je zapaziti da smo za računanje poluprečnika kruga koristili funkciju članicu klase *tacka* koju smo pozivali iz funkcija za računanje površine i obima. Ovo smo morali realizovati na ovaj način jer iako klasa *krug* ima kao podatke objekte tipa *tacka*, centar kruga i tacka sa njegovog oboda, ne možemo direktno pristupati koordinatama ovih tačaka jer su one privatni podaci članovi. Njima se može pristupiti samo uz pomoć funkcija članica klase kojoj pripadaju ili uz pomoć prijateljskih funkcija i klasa o čemu će biti riječi kasnije. Dakle, naredba tipa *r=pow(centar.x-sa_kruga.x,2)+pow(centar.y-sa_kruga.y,2);* u okviru funkcije za računanje površine i obima bi izazvala grešku i kompajler bi nam javljao da na tom mjestu podaci članovi tačaka *centar* i *sa_kruga* nisu dostupni. Klasa *krug*, takođe, posjeduje odgovarajuće konstruktore. Realizacija default konstruktora se ne razlikuje od realizacije za ranije korišćene klase ali moramo voditi računa da klasa čiji su objekti njeni podaci takođe mora imati default konstruktor jer bi kompajler u suprotnom prijavio grešku, s obzirom da ovaj konstruktor nema argumente i poziva konstruktora za klasu čiji objekat sadrži bez argumenata, dakle dafault. Sljedeći konstruktor, koji će ujedno izvršiti inicijalizaciju objekta klase za koji bude pozvan, se može realizovati na više načina. Mi smo koristili sledeći način *krug::krug(float x1, float x2, float x3, float x4) :centar(x1,x2) {sa_kruga=tacka(x3,x4);}*. Ovdje vidimo da smo kao argumente konstruktora poslali četri podatka tipa *float* a onda eksplicitno pozivali konstruktora za klasu tačka sa odgovarajućim argumentima. U glavnom programu smo konstruktoru proslijedili četri podatka tipa *float*, nakon toga će on sam pozivati odgovarajuće konstruktore za klasu *tacka* u skladu sa načinom na koji smo ga realizovali. Vidimo da smo jednom to odradili u

inicijalizacionoj listi a drugi put u tijelu konstruktora. Konstruktor se može realizovati i kao *krug*::*krug(tacka t1, tacka t2):centar(t1){ sa_kruga=t2;};* pri čemu moramo voditi računa da tačka nije ugrađeni tip podataka već izvedeni i da u skladu sa tim prije poziva ovog konstruktora sa argumentima tipa *tacka* prvo moramo stvoriti te argumente tj. inicijalizovati ih pozivom odgovarajućeg konstruktora. Dakle, u glavnom programu bismo imali naredbe *tacka t1(x,y),t2(x1,y1); krug k1(t1,t2);*. Naravno, za prvi način realizacije je potrebno imati zaglavlje tipa *krug(float, float, float, float);* a za drugi *krug(tacka, tacka);*. Parametri u deklaraciji funkcije moraju odgovarati onima prilikom njene realizacije.

Ostatak koda je izvršen u skladu sa ranijim primjerima. Obratite pažnju da se ispisivanje komentara u više redova vrši na isti način kao u programskom jeziku C, dok se komentar može ispisati u jednom redu koristeći *//*.

Primjer rada programa

Unesite koordinate centra kruga i tacke sa kruga
0 0 2.5 2.5
Povrsina kruga je: 39.25
Obim kruga je: 22.2144

Zadatak 2. 6

Realizovati klasu red koja će predstavljati niz cijelih brojeva proizvoljne dužine, neka klasa sadrži i podatak o dužini reda. Omogućiti da indeks prvog člana može biti različit od nule ukoliko korisnik to zada. Omogućiti pristup članovima reda za upis i izmjenu podataka. Napisati glavni program u kojem će se unositi odgovarajući red i na izlazu davati suma članova tog reda kao i red koji je sastavljen od kvadrata elemenata reda koji je korisnik unio.

```
#include<iostream.h>

class Array{
  private:
      int *p;       //Pokazivac na pocetak niza cijelih brojeva
      int num;      //Duzina niza
      const int i1;   //Indeks pocetnog elementa niza (moze biti razlicit od 0)
  public:
      Array(int first_index=1,int number=0);
      Array(const Array & a);
       ~Array();
      int first(){return i1;}
      int length(){return num;}
      int last(){return i1+num-1;}
      int read(int index){return p[index-i1];}
      void change(int, int);
      };

Array::Array(int first_index, int number):i1(first_index), num(number){
      p=new int[num];
      }
```

```
Array::Array(const Array &a): i1(a.i1), num(a.num){
        p=new int[num];
        for(int i=0;i<num;i++) p[i]=a.p[i];
        }

Array::~Array(){delete[] p; p=0;}

void Array::change(int index, int value){
        p[index-i1]=value;
        }

int sum(Array &a){
        int s=0;
        for(int i=a.first();i<=a.last();i++) s+=a.read(i);
        return s;
       }

void square(Array &a){
        for(int i=a.first();i<=a.last();i++) {
                int x;
                x=a.read(i);
                a.change(i,pow(x,2));
                }
               }

void main(){
        int i, n,x;
        cout<<"Koliko clanova zelite da ima vas red?"<<'\n';
        cin>>n;
        cout<<"Od kojeg indeksa zelite da pocinje indeksiranje vaseg reda"<<'\n';
        cin>>i;
        cout<<"Unesite clanove reda"<<endl;
        Array a1(i,n);
        for(int j=i;j<n+i;j++){
                cin>>x;
                a1.change(j,x);
                }
        Array a2(a1);
        square(a2);
        cout<<"Suma clanova zadatog reda je: "<<sum(a1)<<endl;
        cout<<"Red dobijen kvadriranjem clanova zadatog je: "<<endl;
        for(int j=a2.first(); j<=a2.last();j++) cout<<a2.read(j)<<" ";
        }
```

Dati primjer ilustruje korišćenje konstruktora kopije. Veoma često u programima želimo inicijalizovati novi objekat tako da ima iste vrijednosti podataka članova kao i neki već postojeći. To ćemo učiniti konstruktorom kopije. Ukoliko mi ne definišemo konstruktor kopije pozvaće se default. Ovakav konstruktor kopije kopira podatke članove postojećeg objekta u objekat koji stvaramo jedan po jedan. Ukoliko data klasa ima kao podatak objekat neke druge klase (*tacka* u prethodnom zadatku) za kopiranje tog podatka (objekta) poziva se konstruktor kopije za klasu

(*tacka*) čiji je on objekat. Kopiranje se dešava i prilikom korišćenja objekata u inicijalizacionoj listi pri realizaciji konstruktora. U prethodnom primjeru bi to bilo pri realizaciji konstruktora kao *krug*::*krug(tacka t1, tacka t2):centar(t1){ sa_kruga=t2;}*, prilikom deklaracije *krug k1(t1,t2);* konstruktor kopije klase *tacka* će se pozivati dva puta, za kopiranje *t1* u *centar* i *t2* u *sa_kruga*. Konstruktor kopije se ne poziva samo prilikom eksplicitne inicijalizacije novog objekta već i prilikom transfera argumenata funkcije koji su objekti neke klase i kada kao rezultat funkcije imamo objekat neke klase, jer se i u tim slučajevima stvaraju novi, privremeni, objekti.

Default konstruktor kopije, u većini slučajeva, zadovoljava naše potrebe i nema svrhe definisati novi. Međutim, ukoliko radimo sa klasama koje dinamički alociraju memoriju moramo definisati sopstvenog konstruktora kopije. Naime, u ovim slučajevima kao podatak član imamo pokazivač, u našem primjeru *p* koji je pokazivač na niz cijelih brojeva. Ukoliko bismo u slučajevima kada nam je to potrebno koristili ugrađenog konstruktora kopije on bi kopirao vrijednost ovog pokazivača tako da bismo imali dva objekta koja pokazuju na isti niz, ukoliko bismo sada izmijenili elemente jednog niza ta izmjena bi se odrazila i na drugi. Može se zaključiti da kada god u nekoj klasi vršimo dinamičko alociranje memorije potrebno ja da definišemo sopstvenog konstruktora kopije.

U ovom primjeru želimo uvesti korisnički tip podataka, niz čiji indeks može počinjati od bilo kojeg broja. Da bi imali sve potrebne informacije za manipulaciju datim nizom naša klasa ima tri podatka člana: pokazivač na početak niza *p*, broj elemenata niza *num* i vrijednost prvog ineksa *i1*. Kada smo koristili nizove onako kako ih kompajler sam formira dovoljno je bilo imati podatke o dužini niza i broju elemenata jer je početni elemant uvijek imao indeks 0. Mi ovdje želimo da nam početni indeks koji ćemo koristiti za pristup elementima ovog niza ima bilo koju vrijednost pa taj podatak moramo imati. U prethodnom primjeru smo rekli da ukoliko definišemo bilo koji konstruktor moramo definisati i deafult. Na prvi pogled, može djelovati, da smo ovdje prekršili to pravilo. Ako se bolje pogleda konstruktor vidi se da on može biti pozivan bez ijednog argumenta, ima podrazumijevane parametre koji će se u tom slučaju koristiti pa ćemo imati niz sa prvim indeksom 1 a bez elemenata, može se pozvati i sa samo jednim argumentom, drugi će dobiti podrazumijevanu vrijednost, ili sa oba. Tek kada pošaljemo oba argumenta vrši se dinamička alokacija memorije. Ovdje smo formirali objekat koji predstavlja istancu niza sa indekasom prvog člana *i1*, *num* elemenata, zauzeli memoriju za njih i vratili pokazivač na početak te memorije *p*. U okviru glavnog programa ćemo elementima niza dodijeliti neke vrijednosti. I konstruktor kopije, kao i običan konstruktor, ima isto ime kao i klasa kojoj pripada, razlika je, pored same radnje koju obavlja, u tome što konstruktor kopije ima tačno određen tip podatka koji mu je argument, to je konstantna referenca na objekat klase za koju se pravi dati konstruktor kopije *Array (const Array & a)*, može se izostaviti samo *const*. Konstruktor kopije preslikava vrijednosti podataka objekta koji se šalje kao argument u vrijednosti podataka novog objekta *num* i *i1*, vrši alociranje novog dijela memorije, odgovarajućeg kapaciteta, u koji presipa elemente niza. Sada dobijamo dva objekta koji imaju isti sadržaj ali ukazuju na različite djelove memorije. Obratite pažnju na način pristupanja elementima objekta koji se šalje kao argument i objekta za koji se poziva konstruktor kopije. Vidimo da smo prilikom presipanja elemenata pristupali im kao da počinju od indeksa 0 pa do kraja niza iako smo rekli da možemo imati proizvoljan početni indeks. Ovo je i logično jer će se elementi niza čuvati od indeksa 0 pa do kraja niza a mi treba da realizujemo klasu tako da korisnik ima osjećaj da počinju od nekog proizvoljnog indeksa *i1*. To ćemo učiniti u funkcijama za pistup gdje će on za zadati početni indeks *i1* pristupati prvom elementu pozivajući ga kao *i1* a zapravo će očitavati element sa indeksom 0. Znači, u funkciji *read()* kada pošaljemo vrijednost indeksa *index* naredbom *return p[index-i1];* mi vraćamo odgovarajući element. Ukoliko zadamo da je početni indeks niza 3 a želimo očitati element sa indeksom 5, kao vrijednost stvarnog argumenta za *index* u pozivu funkcije šaljemo 5 a očitavamo 5-3, dakle element sa indeksom 2. Ovo je i logično jer ćemo u memoriji imati niz sa indeksima 0,1,2...*num*-1 a korisnik ima osjećaj da su indeksi *i1*, *i1*+1,...,

i1+num-1. Podsjetite se koje je ovo svojstvo objektno orjentisanog pristupa. Sličnu logiku smo koristili i prilikom realizacije funkcije *change()*, ovdje smo kao argumente slali i indeks čija se vrijednost mijenja i novu vrijednost koja mu se dodjeljuje. Funkcije *first()*, *lenght()* i *last()* vraćaju indeks (zadati) prvog elementa niza, dužinu niza i indeks poslednjeg elementa niza. One su veoma jednostavne i zato su implementirane kao *inline*.

Obratite pažnju da funkcije za računanje sume i kvadrata elemenata nisu funkcije članice pa iako rade samo sa jednim objektom mora im se taj objekat *a* proslijediti. Mi smo proslijedili referencu na objekat jer kada bismo proslijedili objekat vršilo bi se kopiranje prilikom prenosa parametara funkciji i stvaranje privremenog objekta u toku izvršavanja funkcije a na ovaj način smo to izbjegli. Princip realizacije ovih funkcija se ne razlikuje od onog koji bismo koristili u programskom jeziku C. Trebate obratiti pažnju da, s obzirom da ovo nije funkcija članica klase *Array*, ni njena prijateljska funkcija, nema pravo pristupa njenim privatnim podacima pa kada joj bude potreban početni indeks niza neće mu moći pristupiti direktno naredbom *a.i1* već mora pozivati funkciju za očitavnje vrijednosti tog podatka, koja je javna i kao takvu je može koristiti *a.first()* (*for(int i=a.first();i<=a.last();i++) s+=a.read(i);*). U funkciji *square()* šaljemo kao argument referencu na neki objekat (niz) pa sve što budemo radili sa njegovim elementima odražava se na sam objekat (prenos po referenci). Znači, iako je tip rezultata *void* mi imamo rezultat (niz koji smo poslali kao stvarni argument i čiji su elementi sada kvadrirani) ali ga nije potrebno vraćati jer smo poslali referencu na objekat.

U glavnom programu korisnik zadaje početnu vrijednost niza i broj elemenata, nakon ovog se vrši inicijalizacija objekta a*1*. Ovo je moguće izvršiti prije unošenja vrijednosti pojedinih elemenata niza jer konstruktor samo zauzima memoriju za traženi broj elemenata i postavlja vrijednosti za početni element i dužinu niza. Nakon toga korisnik unosi elemente niza kojima mi pristupamo na način kako to korisnik očekuje, od indeksa *i1*, i mijenjamo im vrijednost uz pomoć funkcije *chage()* jer nam oni nisu direktno dostupni ovdje, privatni podaci klase. Sada smo u potpunosti inicijalizovali ovaj objekat i možemo ga koristiti u pozivu konstruktora kopije za inicijalizaciju novog objekta *Array a2(a1);* (može i Array a2=a1;) dok se naredbom *Array a2; a2=a1;* ne bi pozivao konstruktor kopije već operator dodijele koji, ukoliko mi nismo definisali drugačije, vrši dodjelu na isti način kao default konstruktor kopije. Dakle i *a1* i *a2* bi ukazivali na isti dio u memoriji pa bi se sve što uradimo sa *a1* odrazilo na *a2* a mi to ne želimo već želimo da u *a1* sačuvamo nepromijenjenu vrijednost elemenata niza a u *a2* da smjestimo kopiju članova koju ćemo proslijediti funkcijama za računaje sume i kvadrata elemenata niza. Ove funkcije nisu funkcije članice već globalne funkcije pa ih u skladu sa tim i pozivamo na odgovarajući način, bez korišćena operatora *dot(.)*.

Još jedna funkcija članica koja se automatski poziva i ugrađena je jeste destruktor. Ova funkcija ima isto ime kao i klasa kojoj pripada i znak tilda ~ ispred njega *~Array();*. Deklariše se kao i ostale funkcije članice. Bez obzira što jezik posjeduje sopstvenog default destruktora, dobra programerska praksa je definisati ga uvijek, čak i kada ima prazno tijelo funkcije. Mora se definisati kada imamo dinamičko dodjeljivanje memorije, kao u ovom slučaju. Klasa može imati samo jednog destruktora i on ne može imati argumente kao ni povratni tip podataka, nije dozvoljena ni upotreba riječi *void*. Ukoliko klasa sadrži kao podatak član objekat druge klase prvo se poziva destruktor za taj objekat, pa tek onda za klasu čiji je objekat član.

U našem primjeru je bilo neophodno definisati destruktor jer se u konstruktoru vrši dinamička alokacija memorije. Nakon uništavanja primjeraka klase potrebno je osloboditi memoriju koju je taj primjerak zauzimao. Mi smo to i uradili u tijelu destruktora *Array::~Array(){delete[] p; p=0;}*. Uglaste zagrade prije pokazivača *p* se koriste uvijek kada imamo niz, tako govorimo da je potrebno osloboditi memoriju koju je zauzeo cio niz a ne samo prvi element. Nakon ovog, pokazivač *p* će i dalje sadržati adresu prvog elementa niza, čiji smo

sadržaj upravi izbrisali. To nikako nije poželjno jer nam pokazivač pokazuje na nešto što više ne postoji. Naredbom *p=0;* ostavljamo ga u ispravnom, praznom, stanju. Ne ukazuje ni na jednu memorijsku adresu.

Primjer rada programa

Koliko clanova zelite da ima vas red?
5
Od kojeg indeksa zelite da pocinje indeksiranje vaseg reda
3
Unesite clanove reda
2 5 13 26 4
Suma clanova zadatog reda je: 50
Red dobijen kvadriranjem clanova zadatog je:
4 25 169 676 16

Zadatak 2. 7

Implementirati klasu koja će predstavljati realizaciju steka uz mogućnost brojanja podataka koji se nalaze na steku. Napisati glavni program u kojem će se na stek postaviti pet podataka, izbrisati dva i dopisati tri a zatim na izlaz dati izgled tako dobijenog steka.

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>

class stek{
      private:
            static int ukupno;
            static stek *vrh;
            int broj;
            stek *prethodni;
      public:
            static void stavi_na_stek(int);
            static int uzmi_sa_steka();
            static int prazan(){return vrh==0;}
            static int imaih(){return ukupno;}
            };

int stek::ukupno=0;
stek* stek::vrh=0;

void stek::stavi_na_stek(int i){
      stek *pom=new stek;
      pom->prethodni=vrh;
      pom->broj=i;
      vrh=pom;
      ukupno=ukupno+1;
      }
```

```
int stek::uzmi_sa_steka(){
	if(!vrh) exit(1);
		stek *pom;
		pom=vrh;
		int i=vrh->broj;
		vrh=pom->prethodni;
		delete pom;
		pom=0;
		ukupno=ukupno-1;
		return i;
			}

int main(){
	int n;
	cout<<"Koliko brojeva zelite da ima stek?\n";
	cin>>n;
	int m;
	cout<<"Unesite podatke koje zelite da postavite na stek"<<endl;
	for(int i=0; i<n;i++){cin>>m; stek::stavi_na_stek(m);}
	cout<<"Koliko brojeva zelite da uzmete sa steka?"<<'\n';
	cin>>n;
	cout<<"Uzeli ste brojeve: ";
	for(int i=0;i<n;i++) cout<<stek::uzmi_sa_steka()<<" ";
	cout<<endl;
	cout<<"Koliko brojeva zelite da postavite na stek?"<<'\n';
	cin>>n;
	cout<<"Unesite podatke koje zelite da postavite na stek"<<endl;
	for(int i=0;i<n;i++) {cin>>m; stek::stavi_na_stek(m);}
	cout<<"Ukupno je"<<stek::imaih()<<endl;
	cout<<"Na steku su brojevi od vrha ka pocetku";
	while(!stek::prazan()) cout<<stek::uzmi_sa_steka()<<" ";
	getch();
	}
```

Stek je se ponaša kao memorijski mehanizam tipa "poslednji-u, prvi iz" (engl. last in, first out). Stek ima svojstvo podataka. Mi smo već ranije naveli da klasama definišemo korisnički tip podataka. Stek će nam, zapravo, biti linearna lista sa specijalnim svojstvom da ne možemo čitati element sa bilo kojeg mjesta u njoj već samo poslednji element i da ne možemo postavljati podatke na bilo koje mjesto već samo na vrh steka. Mi želimo upisivati cijele brojeve na stek i oni će biti sačuvani u podatku članu *broj* i biti dio pojedinog objekta klase. Svaki objekat naše klase će biti vezan za jedan element steka, svi objekti čine stek. Statički podatak se koristi za smještanje informacija koje su zajedničke za sve objekte klase. Mi smo rekli da će nam cijela klasa biti stek. Svaki stek, kao specifičan niz, ima jedan vrh i on je zajednički za sve članove klase, svi oni su dio steka koji ima jedan kraj-vrh. Znači, podatak o kraju steka ćemo realizovati kao *static*. Elementi steka su realizovani kao objekti naše klase pa će pokazivač na vrh steka biti tipa pokazivača na objekat klase *stek* i mi smo mu dali ime *vrh*. Ono što je zajedničko za sve objekte koji čine klasu jeste i broj tih objekata, elemenata steka, pa i taj podatak realizujemo kao statički, *ukupno*. Karakteristika pojedinačnih objekata je adresa prethodnog elementa, (elementa koji je prije toga postavljen na stek) nju moramo imati kako bi nam lista bila povezana. Naime, kada izbrišemo neki podatak sa steka moramo znati adresu prethodnog jer je to sada adresa vrha.

Potrebne su nam funkcije za upisivanje elemenata na stek, očitavanje elementa, provjeru da li je stek prazan, kao i informacija o broju elemenata. Nju imamo kao statičku promjenjivu u svakom objektu ali je privatna pa zato pravimo funkciju koja će očitavati njenu vrijednost. Sve funkcije smo definisali kao *static* iz razloga što statičke funkcije rade sa klasom kao cjelinom, tj. sa stekom u našem slučaju. Objekat nad kojim želimo izvršiti neku radnju može da bude argument statičke funkcije, lokalna promjenjiva u njoj ili globalna promjenjiva. Nas, u ovom slučaju, ne interesuju pojedinačni objekti iz klase i ako pogledate glavni program primjetićete da nigdje nismo eksplicitno deklarisali novi objekat. Interesuje nas skup objekata date klase i stek koji oni čine i radimo sve sa klasom kao cjelinom pa je logično da zato koristimo funkcije koje je tako posmatraju a to su *static* funkcije. Da bi pozvali funkciju članicu potrebno je da posjedujemo jedan konkretan objekat. Funkcija *static* se poziva za klasu i ne posjeduje implicitni argument pokazivač *this*, zbog toga pojedinačnim podacima članovima klase ne možemo pristupati samo navođenjem njihovog imena već moramo navesti i kojoj klasi pripadaju dok statičkim podacima pristupamo neposredno. Statički podaci članovi se razlikuju od pojedinačnih po tome što se njihova definicija i inicijalizacija ne vrši u konstruktoru, što je i logično jer se konstruktor poziva za svaki objekat pojedinačno a statički član se inicijalizuje samo jedan put. Statički podataci se definišu van funkcija i van definicije klase. Definišu se posebnim naredbama za definiciju i inicijalizuju se samo konstantom ili konstantnim izrazom. Ovo se najčešće vrši poslije definicije klase a prije glavnog programa. Pošto se na tom mjestu nalaze van dosega klase mora se koristiti operator *::* da bi se označilo kojoj klasi pripadaju. *int stek::ukupno=0; stek* stek::vrh=0;*. Vidimo da se rezervisana riječ *static* koristi samo u deklaraciji kako statičkih promjenjivih tako i statičkih funkcija, dok se prilikom definicije ne smije navoditi.

U programu imamo četri funkcije. Prva vrši operaciju postavljanja podatka na stek. Kao podatak joj šaljemo samo cjelobrojnu vrijednost koju želimo postaviti na stek. Da bi postavila neku vrijednost na stek potrebna joj je adresa vrha steka koja se nalazi u promjenjivoj koja je *static* a rekli smo da ovoj promjenjivoj funkcija može da pristupi samo navodjenjem njenog imena. Dodavanje novog podatka na stek znači uvođenje novog objekta i njegovo povezivanje sa ranijim objektima. U ovoj funkciji prvo vršimo dinamičku alokaciju memorije za novi element. Vrijednost pokazivača na ovaj objekat će nam biti vrijednost pokazivača na novi vrh steka. Mi tu vrijednost ne smještamo odmah u statičku promjenjivu *vrh*, već prvo koristimo prethodnu vrijednost te promjenjive (*vrh*), jer ono što je bilo na vrhu steka prije uvođenja novog elementa steka sada će biti na jednom mjestu iza vrha, dakle, *prethodni* za objekat koji je na vrhu steka. Nakon toga u njegovu promjenjivu *broj* upisujemo vrijednost koja je proslijeđana funkcijom. I tek onda, kada smo povezali taj objekat sa prethodnim elementima, postavljamo *vrh* tako da ukazuje na njega i povećavamo broj elemenata steka za jedan. Primijetimo način pristupanja promjenjivim *broj* i *prethodni*. Nije bilo dovoljno navesti samo njihovo ime jer statička funkcija nema pokazivač *this.* Funkcija *uzmi_sa_steka* prvo provjerava da li je *stek* prazan. To se može desiti u dva slučaja, da ništa nismo stavili na stek, tada je vrijednost *vrh=0* jer smo je tako inicijalizovali ili kada uzmemo sve sa steka, kada je opet vrijednost na koju ukazuje *stek* 0 a što ćemo obezbijedi realizacijom ove funkcije. Dakle, na samom početku provjeravamo da li je stek prazan, ako jeste prekidamo rad programa funkcijom *exit*(), da bi je mogli koristiti uključujemo biblioteku *stdlib.h*. Ukoliko ima nešto na steku vršimo očitavanje. Nema potrebe da šaljemo koji ćemo podatak očitati jer je priroda steka takva da možemo uzeti samo vrijednost sa vrha steka. Ukoliko ima nešto na steku u nekoj lokalnoj promjenjivoj pamtimo pokazivač na taj element *vrh*. Očitavamo vrijednost koja je bila upisana na vrhu steka, što ćemo kasnije i vratiti kao rezultat funkcije. Sada isključujemo element sa steka tako što postavljamo da vrh ne ukazuje na njega već na element koji mu je prethodio u steku, a koji će nakon uzimanja ovog podatka biti na vrhu steka i oslobađamo memoriju. Primjetimo da smo ovom realizacijom omogućili da je vrijednost promjenjive *vrh* kada smo sve očitali sa steka jednaka nuli. Kada očitamo element koji je poslednji element na steku, to je na osnovu pravila o pristupu elementima steka element koji smo prvi upisali, pokazivač *vrh* nakon očitavanja ukazuje na njegov prethodni element a on je dobio

vrijednost koju je imala promjenjiva *vrh* prije nego što je bilo što upisano na stek, dakle nula. Broj elemenata steka umanjujemo za jedan. Funkcija *prazan* provjerava da li je *vrh==0* dakle, da li je stek prazan. Funkcija ukupno nam vraća koliko ukupno elemenata imamo na steku.

Na početku su i *vrh* i *ukupno* inicijalizovane na nulu. Korisnik unosi koliko elemenata želi da stavi na stek, unosi vrijednost koje želi da stavi na njega i u *for* petlji unosi te vrijednosti u odgovarajuće elemente. Nakon toga pitamo korisnika da li želi da očita, i koliko elemenata, a kada izvrši očitavanje javljamo mu koliko je elemenata ostalo na steku.

Primjer rada programa

Koliko brojeva zelite da ima stek?
5
Unesite podatke koje zelite da postavite na stek
1 4 6 9 23
Koliko brojeva zelite da uzmete sa steka?
2
Uzeli ste brojeve: 23 9
Koliko brojeva zelite da postavite na stek?
3
Unesite podatke koje zelite da postavite na stek
2 4 1
Ukupno je6
Na steku su brojevi od vrha ka pocetku1 4 2 6 4 1

Zadatak 2. 8

Realizovati klasu radnici koja će imati tri podatka člana i to: cjelobrojnu promjenjivu za kofecijent za platu, pokazivač na cjelobrojnu promjenjivu za identifikacioni broj radnika i javnu statičku promjenjivu koja će služiti za brojanje ukupnog broja radnika (objekata date klase). Klasa posjeduje konstruktor, destruktor i konstruktor kopije, kao i funkcije članice za pristup podacima radi očitavanja i izmjene i funkciju koja računa koji od dva radnika ima veću platu i vraća identifikacioni broj istog.

```
#include<iostream.h>
#include<conio.h>

class radnici{
      private:
            int koeficijent;
            int *idbroj;
      public:
            radnici(){idbroj=0; broj++;}
            radnici(int,int);
            radnici(const radnici&);
            ~radnici();
            int daj_koef()const{return koeficijent;}
            int daj_idbroj()const{return *idbroj;}
            void promijeni_koef(int);
            void promijeni_idbroj(int);
```

```
        int placeniji(radnici);
        static int broj;
        };

radnici::radnici(int koef, int idb):koeficijent(koef){
        idbroj=new int;
       *idbroj=idb;
        broj=broj+1;
       }

radnici::radnici(const radnici& stari):koeficijent(stari.koeficijent){
       idbroj=new int;
       *idbroj=*stari.idbroj;
       broj=broj+1;
       }

radnici::~radnici(){
       delete idbroj;
       idbroj=0;
       broj=broj-1;
       }

void radnici::promijeni_koef(int a){
       koeficijent=a;
       }

void radnici::promijeni_idbroj(int a){
       *idbroj=a;
       }

int radnici::placeniji(radnici a){
       if(koeficijent>=a.koeficijent)
              return *idbroj;
       else
              return *a.idbroj;
              }

int radnici::broj=0;

main(){
       int a,b;
       cout<<"Unesete podatke za dva radnika"<<endl;
       cin>>a>>b;
       radnici a1(a,b);
       cin>>a>>b;
       radnici a2(a,b);
       cout<<"Vise je placen radnik: "<<a1.placeniji(a2)<<endl;
       cout<<"Ukupno imamo "<<radnici::broj<<" radnika"<<endl;
       getch();
       }
```

U ovom zadatku koristimo ranije uvedene pojmove: konstruktor, destruktor, konstruktor kopije i statičku promjenjivu. Tip promjenjivih nam je zadat postavkom zadatka, za statičku promjenjivu je rečeno da treba da bude javna. Podaci članovi su tipa *int* i *int* *, i konstruktor pozivamo sa dva cijela broja pri čemu u okviru konstruktora dodjeljujemo vrijednost promjenjivoj *int* na koju ukazuje pokazivač *idbroj*. Obratite pažnju da to vršimo naredbom **idbroj=idb*; što je i logično jer mi želimo upisati vrijednost koju smo proslijedili konstruktoru u promjenjivu na koju ukazuje pokazivač *idbroj* a njoj pristupamo upotrebom operatora * ispred imena pokazivača. Konstruktor kopije smo realizovali kao u ranijim primjerima, samo što sad nemamo cijeli niz već samo jedan element na koji nam ukazuje pokazivač i vrijednost promjenjive na koju ukazuje pokazivač iz proslijeđenog objekta (*stari*) prepisujemo u objekat koji kopiramo naredbom **idbroj=*stari.idbroj;*. U zadatku se od nas zahtjeva da vodimo evidenciju o tome koliko imamo radnika. Imamo ih onoliko koliko imamo objekata. Svaki put kada stvorimo novi objekat poziva se konstruktor, bilo da smo taj objekat eksplicitno deklarisali ili da se formira privremeni objekat u koji će da se kopira vrijednost objekta koji je stvarni argument funkcije itd.. Zbog ovoga ćemo povećati vrijednost za statičku promjenjivu *broj* svaki put kada pozovemo konstruktora. Pošto se objekti ne stvaraju samo kada ih mi eksplicitno deklarišemo već i kada želimo da pozovemo neku funkciju, kao što je funkcija za računanje koji radnik ima veću platu, da bi imali pravo stanje o broju radnika potrebno je da u destruktoru imamo naredbu koja će smanjivati broj radnika za 1 svaki put kada se destruktor pozove. Ukoliko smo formirali privremeni objekat u nekoj funkciji nakon završetka izvršavanja te funkcije taj objekat se uništava i poziva se destruktor. Funkcije za promjenu koeficijenta i identifikacionog broja su veoma jednostavne, samo treba voditi računa o tome da je *idbroj* pokazivač na cjelobrojnu promjenjivu i u skladu sa tim na odgovarajući način pristupiti vrijednosti promjenjive na koju on ukazuje i promijeniti je. Isto važi i za funkciju za računanje koji radnik ima veću platu. Potrebni su nam podaci o oba radnika, mi šaljemo samo jedan objekat, podsjetite se zašto to činimo na ovaj način.

U glavnom programu smo unijeli podatke o dva radnika, a kao rezultat ispisujemo koji više zarađuje i koliko ih ima.

Primjer rada programa

Unesete podatke za dva radnika
224
3452
312
5431
Vise je placen radnik: 5431
Ukupno imamo 2 radnika

Zadatak 2. 9

Realizovati klasu trougao koja će kao podatke članove imati tri tačke koje predstavljaju objekte klase tačka. Klasa tačka treba da ima odgovarajuće konstruktore i destruktore kao i funkcije za računanje rastojanja dvije tačke i ugla između prave koju čine dvije tačke i x ose. Klasa trougao treba da posjeduje odgovarajuće konstruktore, destruktore kao i funkciju članicu koja će za tri date tačke provjeravati da li čine trougao ili su to tačke sa jedne prave i druga funkcija članica provjerava da li je trougao, na ovaj način zadat, jednakokraki. Napisati i glavni program u kojem će se unositi koordinate za željene tačke, inicijalizovati potrebni objekti i provjeravati da li date tačke čine trougao i da li je isti jednakokraki.

```
#include<iostream.h>
#include<math.h>
#include<stdlib.h>

class tacka{
      float x;
      float y;
  public:
      tacka(){};
      tacka(float,float);
      ~tacka(){}
      float Aps(){return x;}
      float Ord(){return y;}
      float rastojanje(tacka);
      double radius(tacka);
      };

tacka::tacka(float a,float b):x(a),y(b){}

float tacka::rastojanje(tacka t){
      float d;
      d=sqrt(pow(x-t.x,2)+pow(y-t.y,2));
      return d;
      }

double tacka::radius(tacka a){
     return ((x-a.x)==0||(y-a.y)==0)?0:atan2((y-a.y),(x-a.x));
    }

class trougao{
  private:
      tacka t1;
      tacka t2;
      tacka t3;
  public:
      trougao(){};
      trougao(tacka, tacka,tacka);
      ~trougao(){}
      bool jednakokraki();
      bool je_li_trougao();
      };

trougao::trougao(tacka a,tacka b,tacka c):t1(a),t2(b),t3(c){}

bool trougao::jednakokraki(){
      if(t1.rastojanje(t2)==t2.rastojanje(t3)||t1.rastojanje(t2)==t1.rastojanje(t3)||\
      t2.rastojanje(t3)==t1.rastojanje(t3))
            return true;
      else
            return false;
      }
```

```
bool trougao::je_li_trougao(){
    if((t1.rastojanje(t2)+t1.rastojanje(t3))<t2.rastojanje(t3)||\
    (t1.rastojanje(t2)+t2.rastojanje(t3))<t1.rastojanje(t3)||\
    (t1.rastojanje(t3)+t2.rastojanje(t3))<t1.rastojanje(t2)) return false;
    else return true;
}

void main(){
    float x1,y1;
    cout<<"Unesite koordinate za tjeme A trougla"<<endl;
    cin>>x1>>y1;
    tacka A(x1,y1);
    cout<<"Unesite koordinate za tjeme B trougla"<<endl;
    cin>>x1>>y1;
    tacka B(x1,y1);
    cout<<"Unesite koordinate za tjeme C trougla"<<endl;
    cin>>x1>>y1;
    tacka C(x1,y1);
    trougao tr(A,B,C);
    if(tr.je_li_trougao())
        cout<<"Zadate tacke predstavljaju trougao"<<endl;
    else   {
        cout<<"Zadate tacke ne predstavljaju trougao"<<endl;
        getch();
        exit(EXIT_FAILURE);
        }
    if(tr.jednakokraki())
        cout<<"Trougao je jednakokraki"<<endl;
    else
        cout<<"Trougao nije jednakokraki";
    }
```

Sama realizacija programa predstavlja kombinaciju ranije rađenih zadataka. Kao uslov za određivanje da li su neke tačke tjemena trougla koristili smo činjenicu da zbir dvije stranice trougla mora biti veći od treće.

Primjer rada programa

```
Unesite koordinate za tjeme A trougla
0 0
Unesite koordinate za tjeme B trougla
1 0
Unesite koordinate za tjeme C trougla
0.1
0
Zadate tacke predstavljaju trougao
Trougao nije jednakokraki
```

```
Unesite koordinate za tjeme A trougla
0 0
Unesite koordinate za tjeme B trougla
0 1
Unesite koordinate za tjeme C trougla
1 0
Zadate tacke predstavljaju trougao
Trougao je jednakokraki
```

---

3. Prijateljske klase i funkcije

Zadatak 3.1

Realizovati klasu niz koja će imati kao podatak član pokazivač na niz kompleksnih brojeva, kao i dužinu niza. Implementirati funkciju koja daje proizvod dva kompleksna broja, pri čemu je realizovati kao funkciju članicu i kao prijateljsku funkciju i napisati glavni program u kojem će se ukazati na način pozivanja obije funkcije. Unijeti niz kompleksnih brojeva i implementirati u okviru klase niz funkciju koja će isčitavati dati niz, uzeti da je klasa niz prijateljska klasa klasi kompleksnih brojeva. Klasa sadrži konstruktor, destruktor i konstruktor kopije.

```
#include<iostream.h>
#include<conio.h>

class kompleks{
        float real;
        float imag;
    public:
        kompleks(){}
        kompleks(float a,float b):real(a),imag(b){};
        ~kompleks(){};
        kompleks prod(kompleks);
        float kRe(){return real;}
        float kIm(){return imag;}
        friend kompleks kprod(kompleks,kompleks);
        friend class niz;//sada koristim i class jer klasa niz jos uvijek nije deklarisana.
        };

kompleks kompleks::prod(kompleks a){
    kompleks temp;
    temp.real=real*a.real-imag*a.imag;
    temp.imag=imag*a.real+real*a.imag;
    return temp;
    }

kompleks kprod(kompleks a,kompleks b){
    kompleks temp;
    temp.real=b.real*a.real-b.imag*a.imag;
    temp.imag=b.imag*a.real+b.real*a.imag;
    return temp;
    }

class niz{
    private:
        kompleks *nizk;
        int ukupno;
    public:
        niz(){nizk=0;}
        niz(int,kompleks *);
        niz(niz &);
```

```
        ~niz();
        void read(){
            for(int i=0;i<ukupno;i++){ cout<<nizk[i].real;
            if(nizk[i].imag<0)
                cout<<"-j";
            else
                cout<<"+ j";
            cout<<abs(nizk[i].imag)<<endl;}
            }
        };

niz::niz(int d, kompleks *k):ukupno(d){
    nizk=new kompleks[ukupno];
        for(int i=0;i<ukupno;i++) nizk[i]=k[i];
    }

niz::niz(niz &a):ukupno(a.ukupno){
    nizk=new kompleks[ukupno];
    for(int i=0;i<ukupno;i++)
        nizk[i]=a.nizk[i];
        }

niz::~niz(){
    delete []nizk;
    nizk=0;
    }

int main(){
    float a,b;
    cout<<"Unesite prvi kompleksan broj"<<endl;
    cin>>a>>b;
    kompleks k1(a,b);
    cout<<"Unesite drugi kompleksan broj"<<endl;
    cin>>a>>b;
    kompleks k2(a,b);
    kompleks k3;
    k3=k1.prod(k2);
    cout<<"("<<k3.kRe()<<","<<k3.kIm()<<")"<<endl;
    k3=kprod(k1,k2);
    cout<<"("<<k3.kRe()<<","<<k3.kIm()<<")"<<endl;
    int n;
    cout<<"Koliko clanova imaju vasi nizovi?";
    cin>>n;
    cout<<"Unesite elemente niza < 10"<<endl;
    float x,y;
    kompleks k[10];
    for(int i=0;i<n;i++){
        cin>>x>>y;
        k[i]=kompleks(x,y);}
    niz n1(n,k);
    n1.read();
    getch();
    }
```

Funkcije prijatelji neke klase su funkcije koje nisu članice te klase ali imaju pravo pristupa privatnim elementima klase. Prijateljstvo se ostvaruje tako što ga klasa poklanja datoj funkciji i to tako što se u definiciji date klase navede prototip funkcije i ključna riječ *friend* na njegovom početku. Klase takođe mogu biti prijatelji drugih klasa. U ovom primjeru je data ilustracija klase i funkcije prijatelja. Postavkom zadatka nam je rečeno da naša klasa kao podatak član ima pokazivač na niz kompleksnih brojeva, da bismo deklarisali takav podatak član potrebno je da imamo definiciju (ili makar deklaraciju klase kompleksnih brojeva). Mi smo odmah u potpunosti definisali klasu kompleksnih brojeva i njene prijatelje. U odnosu na ranije korišćene definicije ove klase novina je postojanje prijatelja. Prijateljstvo smo poklonili sledećim naredbama u okviri definicije date klase: *friend kompleks kprod(kompleks,kompleks); friend class niz;*. Ukoliko je prije realizacije klase koja želi pokloniti prijateljstvo nekoj drugoj klasi izvršena deklaracija ili definicija klase prijatelja dovoljno je poslije *friend* napisati samo identifikatora (ime) te klase, kod nas to nije slučaj pa smo prije identifikatora naveli ključnu riječ *class* kako bi kompajler znao da je u pitanju klasa. Kada klasu učinimo prijateljem druge klase mi smo zapravo sve njene funkcije članice učinuli prijateljem te klase.

Poređenje definicije i načina pristupa podacima funkcije članice i prijateljske funkcije učinjeno je realizacijom funkcije za računanje proizvoda dva kompleksna broja jedan put kao funkcije članice a drugi kao prijatelja. Što se tiče funkcije članice *prod()* njena realizacija je jasna. Jedan objekat šaljemo kao argument a podacima drugog funkcija direktno pristupa zahvaljujući implicitnom argumentu *this* koji je pokazivač na objekat za koji se ta funkcija poziva. Prijateljske funkcije nemaju ovaj argument, pa ukoliko želimo pomnožiti dva kompleksna broja oba moramo poslati kao argumente. Novina u odnosu na običnu globalnu funkciju je to što ima pravo pristupa privatnim članovima objekata koji su joj proslijeđeni. Primjetite da im pristupamo direktno a ne preko poziva funkcije članice što smo ranije morali raditi kada koristimo globalnu funkciju. Zapamtite da se kod globalne funkcije mora poslati kao stvarni argument svaki objekat čijem podatku želimo da pristupimo, što je i logično jer ona nije član klase kojoj je prijatelj pa nema pokazivač *this*, samo joj je poklonjena privilegija pristupanja privatnim članovima klase kojoj je prijatelj.

U klasi *Niz* smo pokazali da neka klasa kao podatak član može imati niz objekata druge klase. Pojedinom objektu se pristupa navođenjem identifikatora pointera na početak niza i indeksa tog elementa, kao što bi pristupili *i*-tom elementu niza cijelih brojeva. Pogledajmo realizaciju funkcije za očitavanje elemenata niza, objektu koji je *i*-ti element niza *nizk* se pristupa naredbom *nizk[i]*. Svaki od ovih članova niza je objekat koji ima svoje podatke članove kao što je definisano klasom čiji su oni objekti. Podatku *real i*-tog elementa u nizu *nizk* pristupamo naredbom *nizk[i].real.* Normalno, da klasa kompleksnih brojeva nije poklonila prijateljstvo ovoj klasi ne bismo imali pravo da na ovaj način pristupamo elementima te klase jer su oni privatni, pa bi za očitavanje i izmjenu svakog od njih morali pisati posebnu funkciju. S obzirom da je naša klasa *Niz* prijatelj klasi kompleksnih brojeva ima pravo da pristupi privatnim podacima članovima te klase. Konstruktor ove klase *niz* se razlikuje od ranijih po tome što u njemu ne vršimo samo zauzimanje memorije za elemente niza kompleksnih brojeva već ih i inicijalizujemo, poslali smo kao argument pokazivač na početak niza kompleksnih brojeva kojim će se izvršiti inicijalizacija. Rekli smo da su sve funkcije članice klase koja se učini prijateljskom prijateljske pa će biti i konstruktor, on je specijalna vrsta funkcije članice, dakle, direktno pristupa podacima članovima elemenata niza kompleksnih brojeva. Ovdje nismo prepisivali podatak po podatak iz objekata (kompleksnih brojeva) u nove koji se inicijalizuju već smo iskoristili ugrađeni operator dodjele koji će sam izvršiti dodjelu tako da prepiše podatke članove objekta sa desne strane jednakosti u podatke članove objekta sa lijeve, jedan po jedan. U nekim slučajevima ne treba koristiti ugrađeni operator dodjele već formirati novi, o čemu će biti riječi kod obrade preklapanja operatora.

U glavnom programu smo unijeli podatke za dva komleksna broja *k1* i *k2* a proizvod računali prvi put uz pomoć funkcije članice klase kompleksnih brojeva *k3=k1.prod(k2);* a nakon toga pozivanjem prijateljske funkcije date klase koja vrši istu operaciju *k3=kprod(k1,k2);*. Primjetite razliku u načinu pozivanja. Nakon očitavanja rezultata dobijenih jednom i drugom funkcijom inicijalizovali smo niz komleksnih brojeva koji smo kasnije iskoristili za inicijalizaciju objekta klase *niz*, njegov konstruktor očekuje niz komleksnih brojeva. Na kraju je izvšeno očitavanje tog objekta.

Primjer rada programa

Unesite prvi kompleksan broj
2 -3
Unesite drugi kompleksan broj
1.1 2.3
(9.1,1.3)
(9.1,1.3)
Koliko clanova ima vas niz?4
Unesite elemente niza < 10
1 2
3.4 -2
2 4.3
1 2.3
Niz ima elemente:
1+ j2
3.4- j2
2+ j4.3
1+ j2.3

---

4. Preklapanje operatora

Zadatak 4. 1

Realizovati apstraktni tip podataka koji predstavlja racionalne brojeve, izvršiti preklapanje operatora +, += kao i operatora za prefiksno i postfiksno inkrementiranje. Prilikom realizacije operatora + uzeti u obzir i mogućnosti sabiranja racionalnih brojeva sa cijelim brojem, pri čemu se cijeli broj može očekivati i kao lijevi i kao desni operand.

```
#include<iostream.h>
#include<conio.h>

class Razlomci{
  private:
      int brojilac;
      int imenilac;
  public:
      Razlomci(int=0,int=1);
      ~Razlomci(){}
      friend Razlomci operator+(Razlomci,Razlomci);///Ne saljem referencu jer
      Razlomci& operator+=(const Razlomci &);//mi je potreban poziv konstruktora
      Razlomci& operator++();// posto uzimam u obzir da mogi sabirati sa cijelim brojevima
      Razlomci operator++(int);
      void ispisi(){cout<<brojilac<<"/"<<imenilac<<endl;}
      };

Razlomci::Razlomci(int a,int b):brojilac(a),imenilac(b){}

Razlomci operator+(Razlomci r1,Razlomci r2){
      Razlomci temp;
      temp.brojilac=r1.brojilac*r2.imenilac+r2.brojilac*r1.imenilac;
      temp.imenilac=r1.imenilac*r2.imenilac;
      return temp;
      }

Razlomci &Razlomci::operator+=(const Razlomci &r){
      brojilac=brojilac*r.imenilac+imenilac*r.brojilac;
      imenilac=imenilac*r.imenilac;
      return *this;
      }

Razlomci &Razlomci::operator++(){
      return (*this)+=1;
      }

Razlomci Razlomci::operator++(int i){
      Razlomci temp(*this);//Kreira kopiju objekta koji inkrementiramo.
      (*this)+=1;//Inkrementira vrijednost objekta.
      return temp;//Vraca vrijednost prije inkrementiranja.
      }
```

```
main(){
    int b,i;
    cout<<"Unesite vrijednosti za brojioce i imenioce razlomaka"<<endl;
    cin>>b>>i;
    Razlomci r1(b,i);
    cin>>b>>i;
    Razlomci r2(b,i);
    Razlomci priv;
    priv=r1+r2; priv.ispisi();
    r1++; r1.ispisi();
    priv=r1++; priv.ispisi();
    ++r1; r1.ispisi();
    r1+=r2; r1.ispisi();
    r1+=(r2++);
    r1.ispisi();
    priv=r1+r2+5; priv.ispisi();
    (5+r2).ispisi();
    getch();
    }
```

Rekli smo da klase predstavljaju korisnički definisani tip podataka. Znamo da kod ugrađenih tipova podataka postoje i ugrađeni operatori kojima vršimo operacije nad njima. Logično je očekivati da kad pravimo novi tip podataka želimo da imamo mogućnost manipulacije nad podacima tog tipa. U ovom slučaju želimo da imamo mogućnost sabiranja, inkrementiranja i dekrementiranja racionalnih brojeva. Operatori se preklapaju tako da im se ne izgubi smisao. Dakle, ako želimo preklopiti operator + za dva racionalna broja on treba da obavlja funkciju sabiranja na očekivani način. Operatori se mogu preklopiti bilo kao funkcije članice ili kao prijateljske funkcije. Uputno je operatore koji mijenjaju vrijednost lijevog operanda realizovati kao funkciju članicu, a ako se njime obavlja neka simetrična operacija kao prijateljsku funkciju. Bitno je zapamtiti da nije dozvoljeno uvođenje novih simbola za operatore ni promjena prioriteta. Ako za jednu klasu preklopimo operator + i *, uvijek će * imati veći prioritet nego +, kao i kod standardnih tipova podataka. Objasnimo na početku šta se podrazumijeva pod zahtjevom da operator sabiranja treba da bude preklopljen tako da se racionalni brojevi mogu sabirati sa cijelim, pri čemu se cijeli broj može naći i kao lijevi i kao desni operand. Ovaj uslov nas ograničava da operator sabiranja moramo realizovati kao prijateljsku funkciju. Kada bi je realizovali kao funkciju članicu i ukoliko bismo u programu imali naredbu tipa *5+r2;* pri čemu je *r2* objekat naše klase pozvali bismo funkciju *5.operator+(r2);* i došlo bi do greške u programu. Ukoliko bi imali naredbu tipa *r2+5;* izvršila bi se odgovarajuća konverzija cijelog broja u racionalni i program bi radio kako treba, pogledajmo i zašto. Kada operator preklapamo kao funkciju članicu šalje se samo desni operand kao argument. Ukoliko smo rekli da šaljemo objekat, a ne referencu na taj objekat, i pošaljemo cijeli broj 5 vršiće se kopiranje vrijednosti koju smo poslali u privremeni objekat. Mi nismo poslali tip koji odgovara pa će se tražiti funkcija koja može da izvrši konverziju tipa, ukoliko takva funkcija ne postoji doći će do greške u programu. Mi zapravo pozivamo konstruktora koji će inicijalizovati privremenu promjenjivu tim brojem 5. Da ne bi došlo do greške mi moramo realizovati konstruktora tako da vrši konverziju cijelog broja u racionalni, to smo i učinili. Kada se naš konstruktor pozove sa jednim argumentom (5), drugi uzima podrazumijvanu vrijednost i cijeli broj 5 pretvara u racionalni 5/1. Sada se postavlja pitanje zašto nastaje greška ako je ciijeli broj sa lijeve strane operatora a operator je preklopljen kao funkcija članica. U ovom slučaju bi se pozvala funkcija *5.operator+(r2);* i imali bi grešku jer se konverzija tipa vrši samo za prave argumente a ne i za skriveni argument (pokazivac *this* kojim se prenosi objekat za koji je pozvana funkcija, u ovom slučaju 5), pa bi pozivali funkciju članicu klase racionalnih brojeva za podatak koji nije član te klase. Dakle, važno je zapamtiti da ukoliko realizujemo operator koji je

simetričan poželjno ga je realizovati kao prijateljsku funkciju, a ukoliko se očekuje potreba za konverzijom tipa i sa lijeve i sa desne strane operatora mora se tako realizovati. Pored toga što u tom slučaju mora biti realizovan kao prijateljska funkcija, argumenti funkcije ne smiju biti tipa reference na objekat jer u tom slučaju ne bi došlo do kopiranja i konverzije tipa formalnog argumenta u stvarni, ne bi se pozivao konstruktor. Kada god nam nije potrebna konverzija argumente šaljemo kao referencu na objekat da se ne bi vršilo bespotrebno usporavanje programa kopiranjem stvarnih u formalne argumente. Operator + smo preklopili kao prijateljsku funkciju i, s obzirom da je to globalna funkcija a ne funkcija članica, kao argumente moramo poslati oba operanda. Prijateljska je, dakle ima prava pristupa privatnim podacima članovima klase. Sama logika realizacije u potpunosti prati način sabiranja dva razlomka. Kao rezultat dobijamo takođe razlomak. Postavlja se pitanje zašto je potrebno ovdje vratititi objekat a ne referencu na njega. Vidimo da se vraća vrijednost privremene promjenjive *temp*, znamo da ova promjenjiva postoji samo u toku izvršavanja funkcije odnosno, čim se završi izvršavanje ove funkcija ta promjenjiva više ne postoji. Dakle, kada bismo vratili referencu na nju vratili bi referencu na nešto što ne postoji. Kada rezultat vratimo kao objekat mi zapravo vršimo kopiranje vrijednosti tog objekta u neku promjenjivu koja prihvata rezultat, što je dozvoljeno. Operator += će na već postojeću vrijednost lijevog operanda dodavati vrijednost desnog i vraćati kao rezultat njegovu novu vrijednost. Na osnovu ranije rečenog zaključujemo da ga je poželjno realizovati kao funkciju članicu. Ovo je binarni operator pa obzirom da je funkcija članica klase šaljemo joj samo jedan objekat, desni. Za lijevi će se pozivati i biće joj proslijeđen preko implicitnog argumenta *this*. Realizacija prati logiku same operacije. Primjećujemo da se ovdje vraća kao rezultat referenca na objekat, to je ovdje dozvoljeno jer je to **this* odnosno referenca na objekat za koji je pozvana funkcija a koji postoji i nakon izvršavanja programa. Mogli smo vratiti i vrijednost tog objekta ali bi na taj način vršili nepotrebno kopiranje i usporavanje programa. Postoji jedna specifičnost u realizaciji prefiksne i postfiksne operacije za sabiranje. Obije su unarne operacije i mijenjaju vrijednost operanda pa će biti realizovane kao funkcije članice. Unarne su a funkcije članice dakle nije im potrebno proslijeđivati ništa. Iz tog razloga bi obije imale ime *operator++()* i bilo bi ih nemoguće razlikovati. Razlikovanje je omogućeno tako što se postfiksnoj funkciji uvodi formalni argument tipa *int*, koji služi samo da bi se omogućilo razlikovanje ove dvije funkcije, nije mu potrebno, čak, ni davati ime prilikom definicije te funkcije. Prefiksno inkrementiranje mijenja vrijednost objekta za koji je pozvana funkcija i vraća tako izmjenjeni objekat. Primijetite da smo prilikom realizacije ove funkcije iskoristili već preklopljeni operator +=. Iz istog razloga kao za prethodnu funkciju tip rezultata je referenca. Postfiksno inkrementiranje vrši inkrementiranje objekta koji smo mu poslali ali vraća njegovu prethodnu vrijednost. Zašto ovdje nismo stavili da je tip rezultata referenca već objekat. Obratite pažnju koje se funkcije pozivaju za koju operaciju u glavnom programu. Kako bi glasilo njihovo eksplicitno pozivanje?

Primjer rada programa

Unesite vrijednosti za brojioce i imenioce razlomaka
2 3
3 4
17/12
5/3
5/3
11/3
53/12
248/48
2288/192
27/4

Zadatak 4.2

Realizovati klasu koja predstavlja realne brojeve u zapisu sa decimalnom tačkom (npr. 23.45). Klasa će imati dva podatka člana: jedan za realan broj (23.45) a drugi za cijeli dio tog broja (23). Izvršiti preklapanje operatora -, -= kao i operatora za prefiksno i postfiksno dekrementiranje. Prilikom realizacije operatora - uzeti u obzir i mogućnosti oduzimanja brojeva u decimalnom zapisu i cijelih brojeva, pri čemu se cijeli broj može očekivati i kao lijevi i kao desni operand.

```
#include<iostream.h>
#include<math.h>
#include<conio.h>

class Decimalni{
  private:
      float broj;
      int cijeli_dio;
  public:
      Decimalni(float=0);//,int=0);
      ~Decimalni(){}
      friend Decimalni operator-(Decimalni,Decimalni);
      Decimalni &operator-=(const Decimalni &);
      Decimalni &operator--();
      Decimalni operator--(int);
      void citaj(){cout<<"Broj je:"<<broj<<". Cijeli dio je:"<<cijeli_dio<<endl;}
      };

Decimalni::Decimalni(float a):broj(a),cijeli_dio((int)broj){}

Decimalni operator-(Decimalni a,Decimalni b){
      Decimalni temp;
      temp.broj=a.broj-b.broj;
      temp.cijeli_dio=int(temp.broj);
      return temp;
      }

Decimalni &Decimalni::operator-=(const Decimalni &a){
      *this=(*this)-a;
      return *this;
      }

Decimalni &Decimalni::operator--(){
      (*this)-=1;
      return *this;
      }

Decimalni Decimalni::operator--(int){
      Decimalni temp=*this;
      (*this)-=1;
      return temp;
      }

main(){
```

```
Decimalni a(23.45),b(12.11),c;
(a-b).citaj();
c=a-5;
c.citaj();
c=5-b;c.citaj();
c=b--;
c.citaj();
b.citaj();
c=--b;
c.citaj();
b.citaj();
b-=a;
b.citaj();
b-=5;
b.citaj();
getch();
}
```

Ovaj primjer se realizuje prateći istu logiku kao u prethodnom. Voditi računa o načinu realizacije operatore - - i - =. Uzeti u obzir da se cijeli dio ne može dobiti prostim oduzimanjem podataka članova objekata u kojima je sadržana cjelobrojna vrijednost. Naime, ukoliko imamo dva objekta 2.13 i 3.21 i izvršimo oduzimanje 3.13-2.21=0.92, cio dio rezultata je 0 dok bi prostim oduzimanjem cijelih djelova dobili 1, dakle, pogriješili bi.

Primjer rada programa

```
Broj je:11.34. Cijeli dio je:11
Broj je:18.45. Cijeli dio je:18
Broj je:-7.11. Cijeli dio je:-7
Broj je:12.11. Cijeli dio je:12
Broj je:11.11. Cijeli dio je:11
Broj je:10.11. Cijeli dio je:10
Broj je:10.11. Cijeli dio je:10
Broj je:-13.34. Cijeli dio je:-13
Broj je:-18.34. Cijeli dio je:-18
```

Zadatak 4. 3

Realizovati apstraktni tip podataka koji predstavlja kompleksne brojeve, izvršiti preklapanje operatora ==, != i <. Klasa sadrži konstruktore i destruktore.

```
#include<iostream.h>
#include<conio.h>

class Kompleks{
  private:
     float real;
     float imag;
  public:
     Kompleks(float a=0, float b=0):real(a),imag(b){}
     ~Kompleks(){}
```

```
    friend bool operator==(const Kompleks,const Kompleks);
    friend bool operator<(const Kompleks, const Kompleks);
    friend bool operator!=(const Kompleks, const Kompleks);
    void citaj(){cout<<real<<"+ j"<<imag;}
    };

bool operator==(const Kompleks a,const Kompleks b){
    return (a.real==b.real)&&(a.imag==b.imag);
    }

bool operator!=(const Kompleks a,const Kompleks b){
    return !(a==b);
    }

bool operator<(const Kompleks a,const Kompleks b){
    return (pow(a.real,2)+pow(a.imag,2))<(pow(b.real,2)+pow(b.imag,2));
    }

main(){
    Kompleks k1(1,2), k2(1,3), k3(1,1), k4(1,2);
    if(k1==k2) {
        k1.citaj(); cout<<"=="; k2.citaj();cout<<endl;}
    else
        {k1.citaj(); cout<<"!="; k2.citaj();cout<<endl;}
    if(k1==k4) {
        k1.citaj(); cout<<"=="; k4.citaj();cout<<endl;}
    else
        {k1.citaj(); cout<<"!="; k4.citaj();cout<<endl;}
    if(k1!=k2) {
        k1.citaj(); cout<<"!="; k2.citaj();cout<<endl;}
    else
        {k1.citaj(); cout<<"=="; k2.citaj();cout<<endl;}
    if(k1!=k4) {
        k1.citaj(); cout<<"!="; k4.citaj();cout<<endl;}
    else
        {k1.citaj(); cout<<"=="; k4.citaj();cout<<endl;}
    if(k1<k2) {
        k1.citaj(); cout<<"<"; k2.citaj();cout<<endl;}
    else
        {k1.citaj(); cout<<">="; k2.citaj();cout<<endl;}
    if(k1<k3) {
        k1.citaj(); cout<<"<"; k3.citaj();cout<<endl;}
    else
        {k1.citaj(); cout<<">="; k3.citaj();cout<<endl;}
    if(k1<k4) {
        k1.citaj(); cout<<"<"; k4.citaj();cout<<endl;}
    else
        {k1.citaj(); cout<<">="; k4.citaj();cout<<endl;}
    if(k1<3) {
        k1.citaj(); cout<<"< 3"<<endl;}
    else
        {k1.citaj(); cout<<">= 3";cout<<endl;}
    if(k1!=1) {
        k1.citaj(); cout<<"!= 1";cout<<endl;}
```

```
    else
        {k1.citaj(); cout<<"== 1"; cout<<endl;}
    if(k1==1) {
        k1.citaj(); cout<<"== 1";cout<<endl;}
    else
        {k1.citaj(); cout<<"!= 1"; cout<<endl;}
    getch();
    }
```

Rješenje ovog problema je veoma jednostavno, sve smo operatore realizovali kao prijateljske fukcije jer su svi simetrični. Obratite pažnju na to da smo kao fiktivne argumente funkcija u svim slučajevima imali objekte a ne reference. Zašto?

Primjer rada programa

```
1+ j2!=1+ j3
1+ j2==1+ j2
1+ j2!=1+ j3
1+ j2==1+ j2
1+ j2<1+ j3
1+ j2>=1+ j1
1+ j2>=1+ j2
1+ j2< 3
1+ j2!= 1
1+ j2!= 1
```

Zadatak 4. 4

Kreirati klasu Knjiga koja će imati kao podatak član pokazivač na cijeli broj koji predstavlja broj stranica. Za datu klasu izvršiti preklapanje operatora =.

```
#include<iostream.h>
#include<conio.h>

class Knjiga{
  private:
      int *br_strana;
 public:
      Knjiga(){br_strana=0;}
      Knjiga(int);
      Knjiga(const Knjiga&);
      ~Knjiga();
      Knjiga & operator=(const Knjiga&);
      void dodaj_strane(int a){*br_strana+=a;}
      void ocitavanje(){cout<<"Knjiga ima "<<*br_strana<<" stranica"<<endl;}
      };

Knjiga::Knjiga(int br){
      br_strana=new int;
      *br_strana=br;
      }
```

```
Knjiga::Knjiga(const Knjiga & stara){
      br_strana=new int;
      *br_strana=*stara.br_strana;
      }

Knjiga::~Knjiga(){
      delete br_strana;
      br_strana=0;
      }

Knjiga &Knjiga::operator=(const Knjiga & a){
       if(this!=&a) {//provjera da li je dodjela a=a;
            delete br_strana;
            br_strana=new int;
            *br_strana=*a.br_strana;
            }
       return *this;
       }

main(){
      Knjiga k1;
      cout<<"Unesi broj stranica"<<endl;
      int n;
      cin>>n;
      Knjiga k2(n);
      Knjiga k3=k2; //Poziva se konstruktor kopije;
      k1=k2;//Vrsi se operacija dodjeljivanja-poziva se operator dodjeljivanja.
      k1.ocitavanje();
      k3.dodaj_strane(5);
      k2.ocitavanje();
      k3.ocitavanje();
      getch();
      }
```

Ranije smo pomenuli da kompajler automatski definiše operator dodjele. Način na koji operator dodjele vrši dodjeljivanje je analogan načinu na koji default konstruktor kopije vrši incijalizaciju. Iz istih razloga kao kod konstruktora kopije i ovdje, u nekim slučajevima, moramo definisati sopstveni operator dodjele. Uvijek kada je za neku klasu neophodno definisati konstruktor kopije neophodno je definisati i operator dodjele ukoliko ćemo ga koristiti u radu sa tom klasom. Podsjetimo se da je konstruktor kopije potrebno definisati kad neka klasa posjeduje promjenjivu pokazivačkog tipa. To je slučaj u našem primjeru. Realizacija konstruktora, destruktora i konstruktora kopije je izvršena na uobičajen način. Prvi put se srijećemo samo sa preklapanjem operatora dodjele. On se uvijek mora realizovati kao funkcija članica. Kao argument mu se može poslati objekat, referenca na objekat ili konstantna referenca na objekat iste klase. Mi ćemo uvijek slati konstantnu referencu na objekat. Referencu da ne bismo vršili bespotrebno kopiranje, a konstantnu da bi izbjegli slučajno mijenjanje vrijednosti objekta. Logika operacije dodjeljivanja jeste da se dodijeli vrijednost desnog operanda lijevom i da se njegova vrijednost pri tome ne mijenja i mi ćemo se toga pridržavati. Važno je primjetiti da se operator dodjele razlikuje od konstruktora kopije po tome što on mijenja vrijednost nekom objektu koji već postoji, dok konstruktor kopije stvara novi objekat. Čim taj objekat postoji znači da je imao neku staru vrijednost. Rekli smo da se potreba za realizovanjem sopstvenog operatora dodjele javlja kad kao podatak član imamo pokazivačku promjenjivu. U našem zadatku to nije

slučaj ali ovakva promjenjiva najčešće ukazuje na početak nekog niza koji ima proizvoljan broj elemenata. Pokazivačka promjenjiva na početak niza kojem se dodjeljuje nova vrijednost može ukazivati na niz sa manje ili više objekata od niza na koji ukazuje promjenjiva objekta čija se vrijednost dodjeljuje. U oba slučaja ne možemo u isti memorijski prostor prosto prepisati vrijednosti niza novog objekta. U prvom slučaju nemamo dovoljno memorije a u drugom bi dobili niz koji na početku ima elemente novog niza a na kraju starog. Da se ovakve stvari ne bi dešavale dobra praksa je uvijek prvo izbrisati stari sadržaj promjenjive (niza) na koju je ukazivao pokazivač, pa onda zauzeti novi memorijski prostor koji će biti dovoljan da prihvati novu promjenjivu (niz) i onda izvršiti kopiranje element po element. U našem slučaju smo imali pokazivač na cjelobrojnu promjenjivu i izgleda suvišno oslobađati memoriju koju je zauzimala cjelobrojna promjenjiva pa zauzimati novu, s obzirom da u oba slučaja imamo potrebu za istom količinom memorije, ali smo to učinili u duhu ranijeg razmatranja. Bitno je uzeti u obzir mogućnost dodjele tipa *k1=k1* pri čemu je *k1* objekat klase za koju se preklapa operator dodjele. S obzirom da prvo oslobađamo memoriju koju je zauzimala stara vrijednost promjenjive kojoj se nešto dodjeljuje i brišemo tu vrijednost, u slučaju ovakve naredbe bi značilo da smo izbrisali vrijednost promjenjive koju želimo da dodijelimo. Da bi se to izbjeglo mi provjeravamo da li je adresa objekta čiju vrijednost dodjeljujemo ista kao onog kojem se ta vrijednost dodjeljuje, ukoliko jeste, ne činimo ništa sa objektom, on već sadrži željenu vrijednost, i vraćamo referencu na objekat za koji smo pozvali funkciju. Ukoliko nije, vršimo prethodne operacije, mijenjamo vrijednost objekta za koji je pozvana funkcija i vraćamo referencu na taj objekat, koji sada ima novu, željenu, vrijednost. Primjetimo da ovdje, nakon brisanja sadržaja pokazivačke promjenjive nemamo naredbu kojom joj dodjeljujemo vrijednost nula, to je iz razloga što bi to bila suvišna naredba s obzirom da već u sledećem koraku ta pokazivačka promjenjiva dobija novu vrijednost. Funkcija *dodaj_strane()* je veoma jednostavna i upotrijebili smo je da pokažemo kako će ovakvom realizacijom operatora dodjeljivanja objekti *k3* i *k2* biti potpuno nezavisni nakon naredbe *k2=k3;* što ne bi bio slučaj da smo koristili default operator dodjele, već bi se sve promjene jednog objekta odrazile na drugi.

Primjer rada programa

Unesi broj stranica
123
Knjiga ima 123 stranica
Knjiga ima 123 stranica
Knjiga ima 128 stranica

Zadatak 4. 5

Realizovati klasu student koja će imati podatke o imenu i prezimenu studenta, godini studija i godini upisa, kao i statičku promjenjivu koja računa ukupan broj studenata. Klasa ima odgovarajuće konstruktore i destruktora, preklopljene operatore dodjele i postfiksnog i prefiksnog inkrementiranja (inkrementiranje povećava godinu studija za jedan), kao i prijateljsku funkciju koja za dati skup studenata računa, na svakoj godini, koji student je najduže na studijama i ispisuje njegovo ime, datum upisa kao i godinu na kojoj je.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class Student{
  private:
```

```
        char *ime;
        char *prezime;
        int gd_upisa;
        int gd_studija;
    public:
        static int ukupno;
        Student(){ime=0; prezime=0; ukupno++;}
        Student(int,int,char *,char *);
        Student(const Student &);
        ~Student(){delete []ime; ime=0; delete []prezime; prezime=0;ukupno--;}
        Student &operator=(const Student &);
        Student &operator++();
        Student operator++(int);
        friend void Pretrazi(Student *, int);
        void citaj(){cout<<ime<<" "<<prezime<<" "<<gd_upisa<<" "<<gd_studija<<endl;}
};

int Student::ukupno=0;

Student::Student(int a,int b,char *ime1, char *prezime1):gd_upisa(a),gd_studija(b){
        ime=new char[strlen(ime1)+1];
        strcpy(ime,ime1);
        prezime=new char[strlen(prezime1)+1];
        strcpy(prezime,prezime1);
        ukupno=ukupno++;
        }

Student::Student(const Student & stari):gd_studija(stari.gd_studija),gd_upisa(stari.gd_upisa){
        ime=new char[strlen(stari.ime)+1];
        strcpy(ime,stari.ime);
        prezime=new char[strlen(stari.prezime)+1];
        strcpy(prezime,stari.prezime);
        ukupno=ukupno++;
        }

Student &Student::operator=(const Student &stari){
        if(&stari!=this){
                delete []ime;
                delete []prezime;
                gd_studija=stari.gd_studija;
                gd_upisa=stari.gd_upisa;
                ime=new char[strlen(stari.ime)+1];
                prezime=new char[strlen(stari.prezime)+1];
                strcpy(prezime,stari.prezime);
                }
        return *this;
        }

Student &Student::operator++(){
        gd_studija+=1;
        return *this;
       }

Student Student::operator++(int){
```

```
    Student temp(*this);
       gd_studija+=1;
        return temp;
    }

void Pretrazi(Student *grupa,int broj){
    Student temp[5];
    for(int i=0;i<5;i++) temp[i]=Student(2003,i+1,"Nema","studenata");
    for(int j=0;j<broj;j++){
        if(grupa[j].gd_studija==1){
            if(grupa[j].gd_upisa<temp[0].gd_upisa) temp[0]=grupa[j];}
        else if(grupa[j].gd_studija==2){
            if(grupa[j].gd_upisa<temp[1].gd_upisa) temp[1]=grupa[j];}
        else if(grupa[j].gd_studija==3){
            if(grupa[j].gd_upisa<temp[2].gd_upisa) temp[2]=grupa[j];}
        else if(grupa[j].gd_studija==4){
            if(grupa[j].gd_upisa<temp[3].gd_upisa) temp[3]=grupa[j];}
        else{
            if(grupa[j].gd_upisa<temp[4].gd_upisa) temp[4]=grupa[j];}
        }
    for(int i=0;i<5;i++){
        cout<<"Student koji najduze studira na "<<temp[i].gd_studija\
        <<" -oj godini se zove";
        cout<<temp[i].ime<<" "<<temp[i].prezime;
        if(temp[i].gd_upisa!=2003) cout<<" i upisao je "<<temp[i].gd_upisa<<endl;
        }
    }

main(){
    Student ps[5];
    int upis;
    int gd;
    char pom1[10];
    char pom2[10];
    ps[0]=Student(2001,1,"Slavica","Petranovic");
    cout<<"Unesite podatke za 5 studenata: godinu upisa, godinu na kojoj je";
    cout<<" ime i prezime"<<endl;
    for(int i=1;i<5;i++){
        cin>>upis>>gd>>pom1>>pom2;;
        ps[i]=Student(upis,gd,pom1,pom2);}
    Student a(122,12,"Janko","Marjanovic");
    Student b;
    b=a;
    b.citaj();
    Pretrazi(ps,5);
    for(int i=0;i<5;i++){ps[i].citaj();}
    getch();
    }
```

Ovaj primjer objedinjuje ranije rađene zadatke. Primjetite da smo prijateljsku funkciju realizovali tako što smo na početku pretpostavili da nema studenta na datoj godini pa onda koristili ranije pomenuti metod za traženje najvećeg, najmanjeg člana u nizu. Ukoliko je objekat

niza odgovarao željenoj godini studija provjeravali smo godinu upisa, ukoliko je manja od godine upisa tekućeg, najstarijeg studenta, najstariji je postajao taj objekat.

Primjer rada programa

Unesite podatke za 5 studenata: godinu upisa, godinu na kojoj je ime i prezime
1999 2 Ilija Ilickovic
2001 3 Maja Marjanovic
2000 2 Janko Jovicevic
1978 3 Sanja Mugosa
Janko Marjanovic 122 12
Student koji najduze studira na 1 -oj godini se zove: Slavica Petranovic i upisao je 2001
Student koji najduze studira na 2 -oj godini se zove: Ilija Ilickovic i upisao je 1999
Student koji najduze studira na 3 -oj godini se zove: Sanja Mugosa i upisao je 1978
Student koji najduze studira na 4 -oj godini se zove: Nema studenata
Slavica Petranovic 2001 1
Ilija Ilickovic 1999 2
Maja Marjanovic 2001 3
Janko jovicevic 2000 2
Sanja Mugosa 1978 3

---

5. Nasljeđivanje

Zadatak 5 1

Projektovati klase krug i kvadrat koje su izvedene iz klase figure (sadrži težiste kao zajedničku karakteristiku za sve figure, funkciju koja omogućava pomjeraj težista za zadatu vrijednost i virtuelne funkcije obim, površina i čitaj). Klase treba da imaju specifične funkcije za računanje obima i površine kao i očitavanje odgovarajućih podataka članova.

```
#include<iostream.h>
#include<conio.h>

class Tacka{
  private:
      float x;
      float y;
  public:
     Tacka(float a=0,float b=0):x(a),y(b){}
     ~Tacka(){}
     float aps(){return x;}
     float ord(){return y;}
     void Tcitaj(){cout<<"("<<x<<","<<y<<")";}
     };

const Tacka KP;

class Figura{
  private:
      Tacka teziste;
  public:
      Figura(Tacka t=KP):teziste(t){}
      virtual ~Figura(){}
      void pomjeri(float a, float b){teziste=Tacka(teziste.aps()+a,teziste.ord()+b);}
      virtual float Obim()const=0;
      virtual float Povrsina()const=0;
      virtual void citaj(){cout<<"T="; teziste.Tcitaj();}
      };

class Krug : public Figura{
  private:
      float poluprecnik;
  public:
      Krug(float rr=1, Tacka k=KP):Figura(k),poluprecnik(rr){};
      ~Krug(){}
      float Obim()const{const float pi=3.14; return 2*poluprecnik*pi;}
      float Povrsina()const{const float pi=3.14; return pow(poluprecnik,2)*pi;}
      void citaj();
      };

void Krug::citaj(){
```

```
        cout<<"U pitanju je krug: [ r="<<poluprecnik<<",";
        Figura::citaj();
        cout<<", O="<<Obim()<<", P="<<Povrsina()<<" ]"<<endl;
        }

class Kvadrat:public Figura{
  private:
        float osnovica;
  public:
        Kvadrat(float a=1, Tacka t=KP):Figura(t),osnovica(a){}
        ~Kvadrat(){}
        float Obim()const{return 4*osnovica;}
        float Povrsina()const{return pow(osnovica,2);}
        void citaj();
        };

void Kvadrat::citaj(){
        cout<<"U pitanju je kvadrat: [ a="<<osnovica<<",";
        Figura::citaj();
        cout<<", O="<<Obim()<<", P="<<Povrsina()<<" ]"<<endl;
        }

main(){
        Figura *pf[4];
        pf[0]=new Krug;
        pf[1]=new Kvadrat;
        pf[2]=new Krug(2,Tacka(3,3));
        pf[3]=new Kvadrat(2.5,Tacka(1.3,2));
        for(int j=0;j<4;j++) pf[j]->citaj();
        pf[0]->pomjeri(1,0.5);
        pf[1]->pomjeri(0.5,1);
        for(int j=0;j<2;j++) pf[j]->citaj();
       for(int j=0;j<4;j++) {delete pf[j]; pf[j]=0;}
        getch();
        }
```

Uvijek kada imamo nalјeđivanje klasa prvo definišemo osnovnu klasu da bi imali pregled onoga što radimo. U postavci zadatka je rečeno da osnovna klasa ima podatak o težištu figure. Težište se predstavlja koordinatama u Dekartovom koordinatnom sistemu. Mi smo to riješili tako što smo prvo definisali klasu *Tacka* pa smo kao podatak član klase *Figura* uzeli objekat klase *Tacka*. Moglo se to odraditi i bez definicije posebne klase *Tacka* ali s obzirom na prirodu problema, rad sa geometrijskim figurama, može se očekivati potreba za više objekata tipa *Tacka* kada bi morali imati klasu *Tacka* zasebno realizovanu. Osnovna klasa može imati funkcije članice koje se realizujuju kao i u ranije korišćenim klasama ali se kod nasleđivanja pojavljuju i virtuelne funkcije. Potreba za njima se javlja zbog same prirode nasleđivanja. Nasleđivanje koristimo kada imamo neke nove tipove koji imaju iste osobine kao neki postojeći tipovi (osnovne klase) ali i neke specifične osobine. U našem primjeru smo izveli klase *Krug* i *Kvadrat* iz klase *Figura*. Obije geometrijske figure imaju težište ali krug ima i poluprečnik dok smo kvadrat karakterisali dužinom osnovice. Svaka geometrijska figura se karakteriše mogućnošću izračunavanja obima i površine ali se za svaku od njih izračunava na drugi način. Takođe će i način ispisivanja komentara o određenoj figuri zavisiti od toga koja je: krug ili kvadrat. Shodno tome dolazi do potrebe za korišćenjem virtuelnih funkcija. Da bi kompajler znao da je u pitanju virtuelna funkcija potrebno je navesti ključnu riječ *virtual* u deklaraciji funkcije u osnovnoj klasi,

u definicijama izvedenih klasa nije potrebno to ponavljati. Virtuelne funkcije se karakterišu time da će se izvršavati na način koji je deklarisan u klasi za koju se pozivaju bez obzira što se pozivaju preko pokazivača na osnovnu klasu. Naime, kao što se vidi u glavnom programu, moguće je izvršiti implicitnu konverziju pokazivača na objekat izvedene klase u pokazivač na objekat osnovne klase. Obratno se ne preporučuje! U glavnom programu smo imali skup krugova i kvadrata, da ne bi svaki put provjeravali da li je u pitanju krug ili kvadrat pa onda pozivali odgovarajuću funkciju mi smo iskoristili virtuelne funkcije. Deklarisali smo niz pokazivača na objekte osnovne klase, zatim svakom dodijelili vrijednost pokazivača na neku od izvedenih klasa i u jednoj petlji pozivali virtuelne funkcije za računanje obima i površine. S obzirom da su virtuelne, koriste dinamičko povezivanje odnosno pozivaju onu funkciju koja odgovara objektu na koji ukazuje taj pokazivač a ne objektu koji odgovara tipu pokazivača. Dakle, bez obzira što je pokazivač deklarisan kao pokazivač na osnovnu klasu, *Figure,* izvršavaće se funkcija koja odgovara klasi čija je instanca objekat na koji ukazuje taj pokazivač, *Krug* ili *Kvadrat*. Primijetimo da kada funkciji pristupamo preko pokazivača koristimo operator—>. Da bi se funkcije realizovale kao virtuelne nije dovoljno samo navesti ključnu riječ *virtual* u osnovnoj klasi. Potrebno je da kada budemo definisali tu funkciju u izvedenim klasama ona bude identična po pitanju imena, broja i tipa argumenata kao i tipa rezultata onoj u osnovnoj klasi, čak je potrebno u svim klasama navesti ključnu riječ *const* ukoliko je ona navedena u osnovnoj klasi kao što je slučaj kod nas. Ukoliko ovo nije izvršeno sakrili smo funkciju iz osnovne klase. Dakle neće biti u pitanju funkcija sa preklopljenim imenom, i možemo joj pristupiti samo uz pomoc operatora *::*. Primjer virutelnih funkcija su *obim()*, *povrsina()* i *citaj()*. Vidimo da je samo poslednja i definisana u okviru osnovne klase, to je iz razloga što ne znamo na koji ćemo način izračunati obim i površinu nekog objekta dok ne vidimo koji je to objekat (krug ili kvadrat) a moramo joj navesti deklaraciju u osnovnoj klasi da bi realizovali mehanizam virtuelne funkcije. Da se ne bi pisalo prazno tijelo funkcije uvodi se čisto virtuelna funkcija. Kompajler je razlikuje po tome što nema tijelo funkcije ali ima *=0* na kraju standardnog zaglavlja. Klasa koja posjeduje makar jednu čisto virtuelnu funkciju je apstraktna klasa i ne mogu se deklarisati objekti te klase ali mogu pokazivači i reference na tu klasu. U našem slučaju nismo imali potrebu za pristupanjem podacima članovima osnovne klase *Figura* (koji su ujedno i podaci članovi izvedenih klasa, naslijedili su ih od osnovne) u nekoj od izvedenih klasa pa smo ih realizovali kao *private*. Ukoliko bi imali potrebu za pristupanjem realizovali bi ih kao *protected*. Na ovaj način dajemo dozvolu klasi koja se izvodi iz osnovne da može da direktno pristupa tim podacima, dok se van te izvedene klase i dalje nema pravo pristupa istim.

Primijetimo da smo konstruktor klase *Tacka* realizovali tako da kada je pozvan bez argumenata postavlja sve koordinate na nulu. To smo iskoristili da uvedemo jednu globalnu promjenjivu *KP* koju ćemo koristiti kao podrazumijevanu vrijednost u svim sledećim konstruktorima. U osnovnoj klasi je ujedno realizovan i default konstruktor i standardni. Ukoliko se pozove kao default, težište je u koordinatnom početku. Funkcija za pomjeranje koordinata težišta (članica klase *Figura*) ne može direktno da pristupa koordinatama objekta (instanca klase *Tacka*) koji je podatak te klase jer su one njegovi privatni podaci i samo funkcije članice klase čija je on instanca imaju pravo pristupa, pa smo te funkcije i iskoristili da očitamo koordinate. Za promjenu njihovih vrijednosti smo koristili konstruktor. Naime, mi ovdje stvaramo novi objekat *težište* koji će sadržati informaciju o pomjerenim koordinatama. Funkcija *citaj()* očitava koordinate. Klasa *Krug* je izvedena javno iz klase *Figura*. To znači da će svi članovi (podaci i funkcije) koje je ova klasa naslijedila od klase *Figura* imati istu vidljivost kao u njoj, dakle javni ostaju javni a privatni su i dalje privatni. Izvedena klasa nasleđuje iz osnovne klase sve podatke članove i funkcije članice i može im pristupati navođenjem njihovog imena ukoliko ima pravo pristupa. *Krug* ima kao specifičan podatak u odnosu na klasu *Figura*, *poluprečnik* ali takođe i *teziste* koje je naslijedio iz nje. S obzirom na ovo konstruktoru se moraju proslijediti vrijednosti za inicijalizaciju oba člana. Karakteristika je kod izvedene klase samo to što se ne mora voditi računa o tome kako da inicijalizujemo članove koje smo naslijedili iz osnovne klase već se briga o tome prepušta konstruktoru osnovne klase koji se prvi poziva u konstrukoru izvedene klase. Mi smo ovdje istovremeno realizovali i default konstruktor i konstruktor kojem proslijeđujemo

podatke za inicijalizaciju podataka članova pa smo zato u dijelu za inicijalizaciju eksplicitno naveli poziv konstruktora osnovne klase kao i podatak koji smo mu poslali. Da smo zasebno realizovali default konstruktor ne bi morali da pozivamo u njemu default konstruktor osnovne klase, već bi on bio automatski pozvan na početku izvršavanja konstruktora za izvedenu klasu. Napomenimo da redosled kojim se navode podaci u inicijalizacionoj listi ne utiče na redosled inicijalizovanja u smislu da iako poziv konstruktora osnovne klase nije naveden kao prvi ipak će se on prvi pozivati. U ovoj klasi je moguće definisati funkcije *obim()* i *povrsina()* tako da računaju odgovarajuće podatke na način koji odgovara njenim objektima. Primjećujemo da smo u okviru realizacije funkcije *citaj()* pozivali odgovarajuću funkciju klase *Figura* kojoj smo pristupili pomoći operatora *::*. U principu, ovo smo mogli izbjeći da smo podatke klase *Figura* realizovali kao *protected*. Na taj način bi sada imali pravo pristupa tim članovima a da ne moramo pozivati nijednu funkciju članicu te klase. Ovdje smo koristili ovakav način pristupa jer smo željeli da pokažemo razliku između virtuelne i čisto virtulene funkcije, odnosno, da definišemo funkciju *citaj()* u osnovnoj klasi *Figura*.

Klasa *Kvadrat* je realizovana koristeći istu logiku kao za *Krug*.

Obratimo pažnju šta se dešava u glavnom programu. Na početku smo rekli da je *pf* niz od četiri pokazivača na objekte klase *Figura*. Nakon toga smo izvršili dinamičko zauzimanje memorije za objekte klasa *Krug* i *Kvadrat* i postavili pojedine članove niza pokazivača da ukazuju na tako dobijene objekte. Ovo je moguće jer postoji implicitna konverzija pokazivača izvedene klase na pokazivače osnovne klase. Zanimljivo je da smo sada dobili niz pokazivača koju ukazuju na objekte koji pripadaju različitim klasama. Nakon toga smo u *for* petlji pomoću ovih pokazivača pozvali funkciju *citaj()*. Zahvaljujući činjenici da nam je ovo virtuelna funkcija uvijek će se izvršavati funkcija *citaj()* koja pripada klasi na čiji objekat stvarno ukazuje pokazivac *pf[i]*, bez obzira što je on tipa pokazivača na objekte klase *Figure*. Nakon *for* petlje smo pozvali funkciju *pomjeri()* pomoću pokazivača koji ukazuju na krug i na kvadrat. U oba slučaja je izvršena funkcija na način definisan u okviru osnovne klase što je i logično jer data funkcija nije virtuelna pa će se izvršiti za tip pokazivača a ne za ono na što on stvarno ukazuje. Pošto smo na početku zauzeli dinamički memoriju moramo je i osloboditi koristeći operator *delete*. Sada je jasno zašto smo destruktor realizovali kao virtuelnu funkciju, prilikom izvršavanja naredbe *delete pf[i];* poziva se destruktor. Da nije bio realizovan kao virtuelna funkcija pozivao bi se na osnovu tipa pokazivača, dakle, za osnovnu klasu pa ne bi bila oslobođena memorija koju zauzimaju članovi specifični za izvedenu klasu već samo samo memorija zauzeta članovima naslijeđenim od osnovne. Ne bi se oslobodila sva, ranije, zauzeta memorija. Destruktor realizujemo kao virtuelnu funkciju tako da će se pozvati destruktor za klasu na čije objekte svarno ukazuje pokazivač i osloboditi sva zauzeta memorija.

Primjer rada programa

U pitanju je krug: [r=1,T=(0,0), O=6.28, P=3.14]
U pitanju je kvadrat: [a=1,T=(0,0), O=4, P=1]
U pitanju je krug: [r=2,T=(3,3), O=12.56, P=12.56]
U pitanju je kvadrat: [a=2.5,T=(1.3,2), O=10, P=6.25]
U pitanju je krug: [r=1,T=(1,0.5), O=6.28, P=3.14]
U pitanju je kvadrat: [a=1,T=(0.5,1), O=4, P=1]

Zadatak 5. 2

Pojektovati klase pravougaonik i trougao koje su izvedene iz klase figure (sadrži težište kao zajedničku karakteristiku za sve figure, funkciju koja omogućava pomjeraj težišta za zadatu vrijednost i virtuelne funkcije dijagonala, poluprečnik opisanog i čitaj). Klase treba da imaju specifične funkcije za računanje dijagonale i poluprečnika opisanog kruga kao i

očitavanje odgovarajućih podataka članova. Za klasu trougao treba realizovati konstruktor tako da može stvoriti jednakostranični trougao kada se unosi podatak za dužinu jedne stranice kao i opšti slučaj. Funkcija za računanje dijagonale u slučaju trougla treba da vraća -1. Poluprečnik opisanog kruga, za trougao (stranica a, b i c) se može računati kao:

$$R = \frac{abc}{\sqrt{(a^2+b^2+c^2)^2 - 2(a^4+b^4+c^4)}}$$

```
#include<iostream.h>
#include<math.h>
#include<conio.h>

class Tacka{
  private:
      float x;
      float y;
  public:
      Tacka(float a=0,float b=0):x(a),y(b){}
      ~Tacka(){}
      void tcitaj()const{cout<<"("<<x<<","<<y<<")";}
      float aps(){return x;}
      float ord(){return y;}
      };

const Tacka KP;

class Figure{
  protected:
      Tacka teziste;
  public:
      Figure(Tacka t=KP):teziste(t){}
      virtual ~Figure(){}
      void Pomjeri(float dx,float dy){teziste=Tacka(dx+teziste.aps(),dy+teziste.ord());}
      virtual float Dijagonala()const=0;
      virtual float Poluprecnik()const=0;
      virtual void citaj()const=0;
      };

class Pravougaonik:public Figure{
      private:
            float a;
            float b;
      public:
            Pravougaonik(float a1=1,float b1=0.5,Tacka t=KP):a(a1),b(b1),Figure(t){}
            ~Pravougaonik(){}
            float Dijagonala()const{return sqrt(a*a+b*b);}
            float Poluprecnik()const{return Dijagonala()/2;}
            void citaj()const;
            };

void Pravougaonik::citaj()const{
      cout<<"U pitanju je pravougaonik:[a="<<a<<", b="<<b;
```

```
    cout<<". Teziste je :";teziste.tcitaj();
    cout<<", d="<<Dijagonala()<<", R="<<Poluprecnik()<<"]"<<endl;
    }

class Trougao:public Figure{
    private:
        float a1,b1,c1;
    public:
        Trougao(float a2=1,Tacka t=KP):Figure(t){a1=b1=c1=a2;}
        Trougao(float a2,float b2,float c2,Tacka t=KP):a1(a2),b1(b2),c1(c2),Figure(t){}
        ~Trougao(){}
        float Dijagonala()const{return -1;}
        floatPoluprecnik()const{return a1*b1*c1/sqrt(pow(a1*a1+b1*b1+c1*c1,2)+\
        2*(pow(a1,4)+pow(b1,4) +pow(c1,4)));}
        void citaj()const;
        };

void Trougao::citaj()const{
    cout<<"U pitanju je trougao:[a="<<a1<<", b= "<<b1<<", c= "<<c1;
    cout<<". Teziste je :";teziste.tcitaj();
    cout<<", d="<<Dijagonala()<<", R="<<Poluprecnik()<<"]"<<endl;
    }

main(){
    Figure *pf[5];
    pf[0]=new Pravougaonik;
    pf[1]=new Trougao;
    pf[2]=new Trougao(2.5,3);
    pf[3]=new Pravougaonik(2,3,Tacka(1,2));
    pf[4]=new Trougao(2,4,5,Tacka(1,2));
    for(int i=0;i<5;i++) pf[i]->citaj();
    pf[0]->Pomjeri(2.3,4);
    pf[1]->Pomjeri(3.1,2);
    for(int i=0;i<2;i++) pf[i]->citaj();
   for(int j=0;j<4;j++) {delete pf[j]; pf[j]=0;}
    getch();
    }
```

Ovaj se zadatak razlikuje u odnosu na prethodni samo po tome što smo podatke članove klase *Figure*, tj. težište realizovali kao *protected* tako da smo mogli i iz izvedene klase pristupati objektu *teziste* i njegovim funkcijama, konkretno funkciji za očitavanje koordinata. Takođe smo funkciju *citaj()* realizovali kao čisto virtuelnu funkciju. Specifičan je i zahtjev za realizacijom konstruktora klase *Trougao*, tako da se može formirati jednakostranični trougao kada se unosi podatak za dužinu jedne stranice. Pogledajmo default konstruktor. On se može pozvati bez ijednog argumenta, samo sa jednim (koji će biti vrijednost za *a2* jer je on prvi u listi argumenata, dok će biti *t=KP*, i u tom slučaju se ta vrijednost *a2* dodjeljuje dužinama svih stranica, dakle imamo jednakostraničan trougao) i sa dva argumenta. Razmislite kako biste realizovali konstruktor da se zahtjevalo da za poziv sa dva argumenta formira jednakokraki trougao.

Primjer rada programa

```
U pitanju je pravougaonik:[a=1, b=0.5. Teziste je :(0,0), d=1.11803, R=0.559017]
U pitanju je trougao:[a=1, b= 1, c= 1. Teziste je :(0,0), d=-1, R=0.258199]
```

U pitanju je trougao:[a=2.5, b= 2.5, c= 2.5. Teziste je :(3,0), d=-1, R=0.645497]
U pitanju je pravougaonik:[a=2, b=3. Teziste je :(1,2), d=3.60555, R=1.80278]
U pitanju je trougao:[a=2, b= 4, c= 5. Teziste je :(1,2), d=-1, R=0.64727]
U pitanju je pravougaonik:[a=1, b=0.5. Teziste je :(2.3,4), d=1.11803, R=0.559017]
U pitanju je trougao:[a=1, b= 1, c= 1. Teziste je :(3.1,2), d=-1, R=0.258199]

Zadatak 5. 3

Realizovati klasu časovnik koja će predstavljati vrijeme u satima, minutima i sekundama. Klasa ima funkciju za povećavanje broja sekundi za po jednu. Iz date klase izvesti klasu let koja sadrži naziv i broj leta, neka podaci naslijeđeni iz osnovne klase časovnik predstavljaju vrijeme polaska i neka postoji funkcija koja će omogućiti promjenu ovog vremena za neko zadato kašnjenje. Napisati glavni program u kojem se zadaje vrijeme polaska nekog leta i kašnjenje a čita novo vrijeme polaska, sa uračunatim kašnjenjem.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class Casovnik{
  protected:
      int sec,min,sati;
  public:
      Casovnik(){}
      Casovnik(int h,int m,int s):sati(h),min(m),sec(s){}
      ~Casovnik(){}
      int Daj_s()const{return sec;}
      int Daj_m()const{return min;}
      int Daj_sat()const{return sati;}
      void Otkucaj();
      void citaj(){cout<<sati<<" "<<min<<" "<<sec<<endl;}
      };

void Casovnik::Otkucaj(){
      if(sec==59){
            sec=0;min=min+1;
            if(min==60){
                  min=0;
                  sati=sati+1;
                  if(sati==24)sati=0;}}
      else
            sec=sec+1;
            }

class Let:public Casovnik{
  private:
      char *naziv;
  public:
      Let(){naziv=0;}
      Let(char *,int,int,int);
      Let(const Let&);
      ~Let(){delete []naziv;naziv=0;}
```

```
        void Kasnjenje(Casovnik);
        void Citaj(){cout<<"Naziv leta je: "<<naziv<<". Vrijeme odlaska je:";
                Casovnik::citaj();
                }
        };

Let::Let(char *ime,int h,int m,int s):Casovnik(h,m,s){
        naziv=new char[strlen(ime)+1];
        strcpy(naziv,ime);
        }

Let::Let(const Let & stara):Casovnik(stara.sati,stara.min,stara.sec){
        naziv=new char[strlen(stara.naziv)+1];
        strcpy(naziv,stara.naziv);
        }

void Let::Kasnjenje(Casovnik vrijeme){
        int s=0;
        s=vrijeme.Daj_sat()*60*60+vrijeme.Daj_m()*60+vrijeme.Daj_s();
        for(int i=0;i<s;i++) Otkucaj();
        }

main(){
        Let prvi("Beograd-Podgorica",20,30,0);
        Let drugi(prvi);
        prvi.citaj();
        cout<<"Unesite kasnjenje";
        int h,m,s;
        cin>>h>>m>>s;
        prvi.Kasnjenje(Casovnik(h,m,s));
        prvi.Citaj();
        drugi.Citaj();
        getch();
        }
```

Realizacija ovog programa je izvršena u skladu sa ranije urađenim primjerima. Treba obratiti pažnju na to da smo podatke članove osnovne klase realizovali kao *protected* tako da smo im mogli direktno pristupati u konstruktorima. Ipak, treba praviti razliku kada imamo u funkciji *Kasnjenje()* kao argument objekat tipa *Casovnik*. To je nezavisan objekat koji se šalje kao argument funkcije, nije ga klasa naslijedila, i njegovim podacima članovima moramo pristupati uz pomoć javnih funkcija članica klase čija je taj objekat instanca. Vidimo da u okviru realizacije iste funkcije pristupamo funkciji *Otkucaj()* samo navođenjem njenog identifikatora, naša klasa ju je naslijedila od klase *Casovnik* i može joj tako pristupati u svojoj funkciji članici. Inače, jedna funkcija članica može pozivati drugu funkciju članicu u svojoj realizaciji samo navođenjem njenog identifikatora i potrebnih argumenata.

Primjer rada programa

```
20 30 0
Unesite kasnjenje3 30 5
Naziv leta je: Beograd-Podgorica. Vrijeme odlaska je:0 0 5
Naziv leta je: Beograd-Podgorica. Vrijeme odlaska je:20 30 0
```

Zadatak 5. 4

Implementirati klasu motocikl koja je izvedena iz klasa motorno vozilo i vozilo na dva točka a koje su izvedene iz klase vozilo. Klasa vozilo posjeduje podatak o registarskim tablicama, motorno vozilo sadrži podatak o jačini motora a klasa vozilo na dva točka o broju sjedišta. Klasa motocikl ima kao sopstveni podatak član marku motocikla.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class Vozilo{
  protected:
      char *tablice;
  public:
      Vozilo(){tablice=0;}
      Vozilo(char *);
      virtual ~Vozilo(){delete []tablice;tablice=0;}
      virtual void citaj();
      };

Vozilo::Vozilo(char * tab){
      tablice=new char[strlen(tab)+1];
      strcpy(tablice,tab);
      }

void Vozilo::citaj(){
      cout<<"U pitanju je vozilo reg. tab. "<<tablice<<endl;
            }

class MotornoVozilo:virtual public Vozilo{
  protected:
      int snaga_motora;
  public:
      MotornoVozilo(){}
      MotornoVozilo(char * br_tab, int sm):snaga_motora(sm), Vozilo(br_tab){}
      ~MotornoVozilo(){}
      void citaj();
      };

void MotornoVozilo::citaj(){
      cout<<"Snaga motora je "<<snaga_motora<<endl;
}

class Vozilo_na_dva_tocka:virtual public Vozilo{
  protected:
      int br_sjedista;
  public:
      Vozilo_na_dva_tocka(int a=0):br_sjedista(a){}//za Vozilo_na_dva_tocka v;
      // dodjeljuje 0 broju sjedista i sam poziva default konstruktor za Vozilo.
      Vozilo_na_dva_tocka(char *ch,int a):br_sjedista(a),Vozilo(ch){}
      ~Vozilo_na_dva_tocka(){}
      void citaj();
```

```
        };

void Vozilo_na_dva_tocka::citaj(){
     cout<<"Broj sjedista je:"<<br_sjedista<<endl;
     }

class Motocikl:public Vozilo_na_dva_tocka,public MotornoVozilo, virtual Vozilo{
  private:
        char * proizvodjac;
  public:
        Motocikl(){proizvodjac=0;}
        Motocikl(char *,char*,int,int);
        ~Motocikl(){delete []proizvodjac; proizvodjac=0;}
        void citaj();
        };

Motocikl::Motocikl(char * marka,char *tab, int bs,int sm):MotornoVozilo(tab,sm),\
        Vozilo_na_dva_tocka(tab,bs), Vozilo(tab){
        proizvodjac=new char[strlen(marka)+1];
        strcpy(proizvodjac,marka);}

void Motocikl::citaj(){
        Vozilo::citaj();
        MotornoVozilo::citaj();
        Vozilo_na_dva_tocka::citaj();
        cout<<"Marka motora je "<<proizvodjac<<endl;
        }

void main(){
        Vozilo *pv=new Motocikl("Suzuki","PG1234",2,800);
        pv->citaj();
        delete pv;//da destruktor osnovne klase nije virtuelni ovom naredbom bi
        //oslobodili samo dio memorije u kojem se nalazi niz tablice jer bi se pozvao
        //destruktor samo za klasu koja odgovara tipu pokazivaca-Vozilo bez obzira
        //sto pokazivac ukazuje na Motocikl.
        getch();
        }
```

U ovom slučaju klasa *Motocikl* je direktno izvedena iz dvije klase *MotornoVozilo* i *Vozilo_na_dva_tocka.* Zaključujemo da je u pitanju višestruko nasleđivanje. Klase *MotornoVozilo* i *Vozilo_na_dva_tocka* su izvedene iz osnovne klase *Vozilo*. S obzirom da znamo da svaka izvedena klasa naslijeđuje iz osnovne klase sve podatke ovo bi značilo da klasa *motocikl* ima dva podatka *tablice*, jedan naslijeđen od klase *MotornoVozilo* a drugi od *Vozilo_na_dva_tocka.* Logično je da jedan motocikl ne može imati dvoje različitih tablica pa da bi bili dosledni prirodi problema i naš motocikl mora imati samo jedne tablice. U programskom jeziku C++ je to omogućeno uvođenjem virtuelnih osnovnih klasa. Primjerci podataka članova virtuelne osnovne klase će biti naslijeđeni samo jedan put bez obzira na to iz koliko je klasa, koje su izvedene iz ove osnovne, izvedena naša klasa. Neka klasa se čini virtuelnom osnovnom klasom tako što se u zaglavlju klasa koje se izvode iz nje piše rezervisana riječ *virtual*. Dakle, već pri izvođenju klasa *MotornoVozilo* i *Vozilo_na_dva_tocka* morali smo imati na umu da će se iz obije klase izvesti jedna nova i da se podaci njihove osnovne klase ne trebaju naslijeđivati bez jedan put pa je tu potrebno postaviti identifikator *virtual*, kasnije se to ne može učiniti. Ranije smo rekli da se prilikom realizacije konstruktora neke izvedene klase poziva konstruktor njene

osnovne klase, bilo automatski ako nije naveden eksplicitan poziv ili ga mi pozivamo eksplicitnim pozivom kada želimo proslijediti neku vrijednost za inicijalizaciju. Mi želimo inicijalizovati sve podatke klase *Motocikl*, kako one naslijeđene od osnovnih klasa tako i za nju specifičan podatak, *proizvodjac*. Podatke naslijeđene od osnovnih klasa inicijalizujemo proslijeđujući odgovarajuće argumente konstruktoru tih klasa. Konstruktor klase *MotornoVozilo* zahtjeva podatke za broj tablica i snagu motora respektivno. Konstruktor klase *Vozilo_na_dva_tocka*, takođe, zahtjeva dva podatka, za broj tablica i broj sjedišta. Iz realizacije konstruktora za klasu *Motocikl* vidimo da će se oba konstruktora pozvati. Zaključilo bi se da se onda dva puta inicijalizuje podatak *tablice* naslijeđen iz osnovne klase *Vozilo*, a mi smo je učinili virtuelnom da bi imali samo jedan podatak *tablice*. Mehanizam virtuelne osnovne klase se realizuje tako da se u konstruktoru klase izvedene iz više klasa sa zajedničkom virtuelnom osnovnom klasom konstruktor virtuelne osnove klase poziva samo jedan put i to mora učiniti klasa za koju se konstruktor trenutno realizuje. Obzirom na ovo, dobra programerska praksa je da se uvijek u listi klasa iz koje se trenutna izvodi navede kao poslednja virtuelna osnovna klasa (da se ne bi zaboravila inicijalizacija njenih članova u klasi koja se trenutno realizuje). Ovim nismo ništa promijenili jer će se njeni podaci naslijediti za sve klase samo jedan put, realizovana je kao virtuelna, ali smo se osigurali da će njen konstruktor biti sigurno pozvan. Vidimo da smo, bez obzira što sve klase naslijeđujemo kao javne, za svaku klasu navodili ključnu riječ *public*, da to nismo učinili za neku klasu ona bi bila naslijeđena kao privatna.

Primjer rada programa

U pitanju je vozilo reg. tab. PG1234
Snaga motora je 800
Broj sjedista je:2
Marka motora je Suzuki

---

6. Šabloni

Zadatak 6. 1

Sastaviti šablonsku funkciju kojom se od dva uređena niza formira treći, na isti način uređen, niz. Sastaviti glavni program koji primjenjuje ovu funkciju nad nizovima cijelih brojeva i tačaka, pri čemu su koordinate tačaka cijeli brojevi kao i nad dvije tačke koje imaju realne koordinate. Klasu tačka realizovati kao šablonsku klasu kako bi koordinate mogle biti traženog tipa.

```
#include<iostream.h>
#include<conio.h>
#include<math.h>

template<class T>
void uredi(T *prvi, int n1,T* drugi, int n2,T *treci){
      for(int ia=0,ib=0,ic=0;ia<n1||ib<n2;ic++)
            treci[ic]=ia==n1?drugi[ib++]:ib==n2?prvi[ia++]:prvi[ia]<\
             drugi[ib]?prvi[ia++]:drugi[ib++];
              }

template<class S>
class sablon{
      S x;
      S y;
public:
      sablon(S=0,S=0);
      ~sablon(){};
      //bool operator<(const sablon<S>&);
      bool operator<(const sablon&);//Moze i bez <S>
      void citaj(){cout<<"("<<x<<","<<y<<")"<<endl;}
      };

template<class S>
sablon<S>::sablon(S a,S b):x(a),y(b){}

template<class S>
bool sablon<S>::operator<(const sablon& b){
      return (pow(x,2)+pow(y,2))<(pow(b.x,2)+pow(b.y,2));
      }

void main(){
      int d1[]={1,2,5,7,9};
      int d2[]={3,4,7,8};
      int *d3=new int[9];
      int n3;
      uredi(d1,5,d2,4,d3);
      cout<<"Uredjeni niz je:"<<endl;
      for(int i=0;i<9;i++)cout<<" "<<d3[i];
      cout<<endl;
      delete []d3;
```

```
sablon<int> *p;
cout<<"Koliko ima tacaka prvi niz?"<<endl;
int i1;
cin>>i1;
cout<<"Unesite niz tacaka sa cjelobrojnim koordinatama"<<endl;
p=new sablon<int>[i1];
int a,b;
for(int i=0;i<i1;i++) {cin>>a>>b; p[i]=sablon<int>(a,b);}
cout<<"Koliko ima tacaka drugi niz?";
int i2;
cin>>i2;
cout<<"Unesite niz tacaka sa cjelobrojnim koordinatama"<<endl;
sablon<int> *q;
q=new sablon<int>[i2];
for(int i=0;i<i2;i++) {cin>>a>>b; q[i]=sablon<int>(a,b);}
int i3;
i3=i2+i1;
sablon<int> *r;
r=new sablon<int>[i3];
uredi(p,i1,q,i2,r);
for(int i=0;i<i3;i++) {cout<<" "; r[i].citaj();}
cout<<endl;
delete []p;
delete []q;
delete []r;
float a1,b1;
sablon<float> t1(2.3,4.6),t4(2.5,6.8);
if(t1<t4) {
cout<<"Tacka ima koordinate: ";
t1.citaj();}
else {
cout<<"Tacka ima koordinate: ";
t4.citaj();}
getch();
}
```

U zadatku se od nas zahtjeva da realizujemo funkciju koja će za dva uređena niza formirati treći, takođe uređen, sastavljen od elemenata dva niza koja šaljemo kao argumente funkcije. Zapravo šaljemo pokazivače na početak tih nizova. Sama realizacija funkcije nije novina. Ovo smo, kada su u pitanju nizovi cijelih i realnih brojeva, mogli odraditi i u programskom jeziku C. Tamo smo morali davati različita imena funkcijama iako obavljaju istu radnju jer imaju različite tipove argumenata. Na početku ovog kursa smo rekli da se u C++ može formirati više funkcija sa istim imenom i da će kompajler na osnovu broja i tipova argumenata za koje se funkcija poziva vršiti razriješavanje poziva. I ako bismo koristili ovakav pristup opet bismo morali da realizujemo tri funkcije: za cijele, realne brojeve i niz tačaka. U principu bi sve te funkcije realizovali na isti način, samo bi na određenim mjestima imali različite oznake za tip podataka sa kojima radimo. U jeziku C++ postoje šablonske funkcije koje nam omogućavaju da ovo realizujemo jednom funkcijom tako što ćemo napraviti šablon za rješavanje datog problema (formiranje trećeg niza na osnovu postojeća dva) ne navodeći konkretan tip podataka. Na osnovu ovog šablona kasnije će se generisati odgovarajuća funkcija za konkretan tip podataka. Naime, kada kompajler naiđe na poziv ove funkcije sa stvarnim argumentima tipa *int* generisaće funkciju za formiranje sortiranog niza cijelih brojeva, kada naiđe na poziv za nizove tačaka generiše funkciju za podatke tipa *tacka.* Isto važi i za klase, i one mogu biti šablonske. U našem slučaju

potrebna nam je klasa koja predstavlja tačke zadate koodinatama cjelobrojnog tipa i klasa za realne koordinate. Da ne bi vršili definiciju dvije klase koje se razlikuju samo u tipu podataka članova mi ćemo ovu klasu realizovati kao šablonsku.

Pogledajmo najprije realizaciju tražene funkcije. Primjećujemo da se prije zaglavlja klase nalazi nova linija koda *template<class T>*. Rezervisana riječ *template* govori kompajleru da ono što slijedi nije obična funkcija već šablonska i da je tretira kao takvu. Unutar zagrada <> se navode parametri šablona, može ih biti više. Riječ *class* govori da je *T* tip podatka, bilo koji tip, bilo neki od ugrađenih, bilo korisnički definisan. Prilikom generisanja funkcije za dati tip podataka može se zamisliti da kompajler zapravo prolazi kroz funkciju i da na svako mjesto na koje nađe identifikator parametra šablona (u našem slučaju *T*) postavlja odgovarajući tip i realizuje takvu funkciju. Na primjer, za poziv iz našeg glavnog programa *uredi(d1,5,d2,4,d3); T* će se zamijeniti sa *int* i generisati funkcija koja će raditi sa nizovima cijelih brojeva. Što se tiče same funkcije *uredi* realizujemo je po logici koju smo koristili i u programskom jeziku C. Šaljemo pokazivač na početak dva niza i broj elemenata niza, kao i pokazivač na početak novoformiranog niza (treba nam kao rezultat). Umjesto određenog tipa naveli smo parametar šablona *T*. Na početku su ideksi svih brojača postavljeni na nulu. U svakom ciklusu *for* petlje će se brojač koji označava tekući indeks novog niza povećavati za jedan. Naime, u svakom ciklusu upisujemo nešto u taj niz pa prelazimo na sledeći element koji će nam postati tekući za sledeći ciklus. Ako smo upisali sve elemente taj brojač je za jedan veći od indeksa poslednjeg elementa novoformiranog niza, dakle jednak broju elemenata tog niza. Spajanje ova dva niza je jednostavno. Vrši se sve dok nismo došli do kraja oba niza. Ukoliko smo upisali elemente jednog niza, prepisujemo preostale elemente drugog. Oni su već uređeni pa nema potrebe da ih uređujemo. Ukoliko nismo došli do kraja nijednog od nizova, pitamo koji niz ima veći element, upisujemo element tog niza u novi, povećavamo brojač za taj niz za jedan dok ne diramo brojač koji ukazuje na tekući element drugog niza jer iz njega ništa nismo upisali, pa mu je to i dalje tekući element.

Što se tiče šablona za klasu, kompajleru najavljujemo na isti način kao i za funkciju da je u pitanju šablon i da koristimo parametar *S*, dakle, navođenjem *template<class S>* prije zaglavlja klase. Kada učinimo neku klasu šablonskom automatski smo učinili i sve njene funkcije šablonima pa ukoliko ih definišemo van definicije klase potrebno je da to najavimo navođenjem *template<class S>* prije definicije funkcije. Ovim smo najavili da ćemo definisati šablonsku funkciju i da koristimo parametar *S*. Ranije smo naglasili da kada definišemo funkcije članice klase van definicije klase kojoj te funkcije pripadaju treba da navedemo identifikator te klase zbog proširenja dosega klase, da bi kompajler znao kojoj klasi funkcija pripada. Obratite pažnju da ovdje samo identifikator klase ne određuje jednoznačno konkretnu klasu. Potrebno je iza imena klase navesti i identifikatore argumenata u zagradama <>, u našem slučaju *sablon<S>::*. Ovo je potrebno jer će se za svaku realizaciju klase generisati njene funkcije članice, za *sablon<int>* jedne, *sablon<real>* druge itd. Ukoliko želimo da navedemo objekat klase unutar same klase dovoljno je samo njeno ime. Ovo se podrazumijeva i ako je objekat date klase argument neke njene funkcije članice, kod nas *operator<*. Obratite pažnju da se ime konstruktora ne mijenja! Proširivanjem dosega, navođenjem *sablon<S>::*, mi smo jednoznačno odredili kojoj klasi on pripada pa ne moramo da pišemo *sablon<S>:: sablon<S>()*, to bi izazvalo grešku u kompajliranju. Ovdje smo izvršili definisanje operatora < za naše klase. Niz objekata ovih klasa se može očekivati kao argument šablonske funkcije u kojoj se koristi taj operator nad argumentima pa moramo to da učinimo. Dakle, pri realizaciji šablonske klase moramo voditi računa za što će se koristiti generisane klase i obezbijediti odgovarajuće funkcije i izvršiti preklapanje potrebnih operatora.

U glavnom programu smo inicijalizovali dva niza cijelih brojeva, alocirali memoriju za rezultujući niz i pozvali funkciju naredbom *uredi(d1,5,d2,4,d3);*. Sada će kompajler generisati funkciju za cijelobrojne argumente. Deklaracijom *sablon<int> *p;* generisali smo klasu sa cjelobrojnim podacima članovima, a *p* je pokazivač na tu klasu. *q* je takođe pokazivač na klasu

sa cjelobrojnim tipom podataka članova. Zatim smo alocirali dovoljno memorije u koju se može smjestiti novonastali niz tačaka (*r* je pokazivač na njen početak) i pozvali funkciju *uredi(p,i1,q,i2,r);*. Kompajler generiše funkciju koja će raditi sa nizom tačaka cjelobrojnih koordinata.

Za vježbu realizujte šablonsku funkciju za uređenje niza u rastući. Za nasumice unesene nizove izvršite njihovo uređivanje i nakon toga primijenite ovdje odrađene funkcije.

Primjer rada programa

```
Uredjeni niz je:
 1 2 3 4 5 7 7 8 9
Koliko ima tacaka prvi niz?
3
Unesite niz tacaka sa cjelobrojnim koordinatama
1 2
2 4
3 5
Koliko ima tacaka drugi niz?2
Unesite niz tacaka sa cjelobrojnim koordinatama
1 1
2 3
 (1,1)
 (1,2)
 (2,3)
 (2,4)
 (3,5)

Tacka ima koordinate: (2.3,4.6)
```

Zadatak 6. 2

Sastaviti šablonsku klasu matrice, pri čemu se može mijenjati tip podataka koje ta klasa sadrži. Napisati glavni program u kojem treba inicijalizovati matricu cijelih brojeva i kompleksnih brojeva i isčitati njihovu vrijednost. Klasa matrica ima konstruktor kopije kao i operatore dodjeljivanja i indeksiranja, pri čemu indeksiranje treba realizovati tako da poziv elementa m(1,3), m tipa matrica, daje element prve vrste i treće kolone. Elementi matrice se čuvaju u vektoru u kojem su prvo elementi prve vrste pa druge itd.

```
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>

template<class T>
class Matrica{
  private:
      T *podatak;
      int vrsta, kolona;
  public:
      Matrica(){podatak=0;}//deafoult konstruktor
      Matrica(int,int);//konstruktor
      Matrica(const Matrica&);//konstruktor kopije
```

```
        ~Matrica(){delete []podatak;}
        Matrica &operator=(const Matrica&);
        T& operator()(int,int);
        int velicina(){return vrsta*kolona;}
        int br_vrsta(){return vrsta;}
        int br_kolona(){return kolona;}
        };

template<class T>
Matrica<T>::Matrica(int a,int b):vrsta(a),kolona(b){
        podatak=new T[vrsta*kolona];
        }

template<class T>
Matrica<T>::Matrica(const Matrica& stara):vrsta(stara.vrsta),kolona(stara.kolona){
        podatak=new T[vrsta*kolona];
        for(int i=0;i<vrsta*kolona;i++){
                podatak[i]=stara.podatak[i];
                }
        }

template<class T>
Matrica<T>& Matrica<T>::operator=(const Matrica& stara){
        if(this!=&stara){//ako se ne dodjeljuje a=a
                vrsta=stara.vrsta;
                kolona=stara.kolona;
                delete []podatak;
                podatak=new T[vrsta*kolona];
                for(int i=0;i<vrsta*kolona;i++)
                        podatak[i]=stara.podatak[i];
                }
        return *this;
}

template<class T>
T& Matrica<T>::operator()(int i,int j){
        //if(i<1||i>vrsta&j<1||j>kolona)
                exit(1);
        return podatak[(i-1)*kolona+j-1];
        }

class Kompleks{
        private:
                float real;
                float imag;
        public:
                Kompleks(float a=0,float b=0):real(a),imag(b){}
                ~Kompleks(){}
                void citaj(){cout<<"("<<real<<","<<imag<<")";}
                };

main(){
        cout<<"Koliko vrsta i kolona zelite da ima matrica cijelih brojeva?"<<endl;
        int a,b;
```

```
	cin>>a>>b;
	Matrica<int> MatricaCijelih(a,b);
	cout<<"Unesite elemente matrice"<<endl;
	for(int i=1; i<=a;i++)
		for(int j=1;j<=b;j++){
		cin>>MatricaCijelih(i,j);};
	for(int i=1; i<=a;i++){
		for(int j=1;j<=b;j++){
		cout<<"MatricaCijelih("<<i<<","<<j<<")==";
		cout<<MatricaCijelih(i,j)<<" ";}
		cout<<endl;}
	cout<<"Koliko vrsta i kolona zelite da ima matrica kompleksnih brojeva?"<<endl;
	cin>>a>>b;
	Matrica<Kompleks> MatricaKompleksnih(a,b);
	float a1,b1;
	for(int i=1; i<=a;i++)
		for(int j=1;j<=b;j++){
		cin>>a1>>b1;
		MatricaKompleksnih(i,j)=Kompleks(a1,b1);}
	for(int i=1; i<=a;i++) {
		for(int j=1;j<=b;j++){
		cout<<"MatricaKompleksnih("<<i<<","<<j<<")==";
		MatricaKompleksnih(i,j).citaj();
		cout<<" ";}
		cout<<endl;}
	Matrica<Kompleks> b2(MatricaKompleksnih),b3;
	b3=b2;
	for(int i=1; i<=a;i++){
		for(int j=1;j<=b;j++){
		cout<<"b2("<<i<<","<<j<<")==";
		b2(i,j).citaj();
		cout<<" ";}
		cout<<endl;}
	for(int i=1; i<=a;i++){
		for(int j=1;j<=b;j++){
		cout<<"b3("<<i<<","<<j<<")==";
		b3(i,j).citaj();
		cout<<" ";}
		cout<<endl;}
	getch();
}
```

Ovo je još jedna ilustracija korišćenja šablona. Razlog uvođenja šablona u ovom zadatku je očigledan, veoma često imamo potrebu da smještamo razne podatke u matrice. Način smještanja elemenata i pristupanja istim se ne razlikuje u zavisnosti od tipa tih elemenata. Kada napravimo šablon matrica veoma lako je definisati matricu sa odgovarajućim tipom podataka (*Matrica<int> MatricaCijelih(a,b);*). Kao podatke članove ove klase imamo broj kolona i vrsta, koji će uvijek biti cjelobrojne promjenjive, i pokazivač na početak niza u koji smo smjestili elemente matrice. Elementi matrice mogu biti za različite instance klasa različitih tipova pa smo tip pokazivača zamijenili šablonskim parametrom *T*. Na svakom mjestu u realizaciji klase gdje bude potrebno navoditi tip pokazivača na početak niza ovih elemenata ili tip podataka koji su smješteni u nizu mi ćemo pisati **T,* odnosno *T* . Kompajler zna da treba da generiše klase sa tipovima podataka koje mi navedemo u <>, deklaracijom oblika *Matrica<int>*

MatricaCijelih(a,b); bi generisali matricu sa cjelobrojnim elementima. Što se tiče pojedinih funkcija članica šablonske klase nema razlike u logici same realizacije. Primijetimo da smo prilikom preklapanja *operator()* kao rezultat vratili referencu a ne samo vrijednost podatka. To je bilo moguće jer je to element niza koji je podatak član objekta za koji se poziva ta funkcija pa on postoji i nakon izvršavanja date funkcije. Razlog zašto smo koristili referencu je, sa jedne strane, jer ne znamo tačno kojeg će tipa biti podaci koji se smještaju u ovaj niz pa ne želimo da vršimo bespotrebno kopiranje moguće velikih objekata. Sa druge strane, na ovaj način smo eliminisali potrebu za funkcijom koju bi koristili za dodjeljivanje vrijednosti pojedinim elementima niza. Kada za određene indekse dobijemo upotrebom ovog operatora odgovarajući element mi smo dobili referencu na taj podatak, pa će se sve promjene nad njim odraziti nad podatkom niza, a ne nad njegovom kopijom. Npr. (*cin>>MatricaCijelih(i,j);*), ovdje se vrši dodjeljivanje vrijednosti koju smo mi unijeli sa tastature određenom elementu matrice (niza).

Kada realizujemo neku klasu čiji objekti želimo da su podaci ovog niza moramo obezbijediti mogućnost pravilnog dodjeljivanja vrijednosti jednog objekta drugom. To znači da u pojedinim slučajevima moramo preklopiti operator dodjele. Mi za klasu kompleksnih brojeva to nismo učinili. Zašto?

Primijetimo da se u glavnom programu pristupa pojedinim elementima kao da su zaista smješteni u matrici. Ovo je i cilj programskog jezika C++, korisnik ne zna na koji način smo mi realizovali neku klasu, zna samo kako on može da je koristi. o*perator()* smo realizovali koristeći naredbu *return podatak[(i-1)*kolona+j-1];* Ovo je logično jer se u matrici elementi počinju indeksirati od jedinice a u memoriji ih mi čuvamo u nizu koji se indeksira od nule pa će elementu (1,1), zapravo odgovarati prvi element u našem nizu, dakle element sa indeksom 0. Nakon unošenja potrebnih podataka o broju vrsta i kolona definisali smo matricu kompleksnih brojeva *Matrica<Kompleks> MatricaKompleksnih(a,b);*. Dalji tok glavnog programa protumačite sami.

Primjer rada programa

Koliko vrsta i kolona zelite da ima matrica cijelih brojeva?
2 3
Unesite elemente matrice
1 3 4 5 8 2
MatricaCijelih(1,1)==1 MatricaCijelih(1,2)==3 MatricaCijelih(1,3)==4
MatricaCijelih(2,1)==5 MatricaCijelih(2,2)==8 MatricaCijelih(2,3)==2
Koliko vrsta i kolona zelite da ima matrica kompleksnih brojeva?
2 3
1 2 2 3 5 6 2.3 4 8 2.1 0 10
MatricaKompleksnih(1,1)==(1,2) MatricaKompleksnih(1,2)==(2,3)
MatricaKompleksnih(1,3)==(5,6)
MatricaKompleksnih(2,1)==(2.3,4) MatricaKompleksnih(2,2)==(8,2.1)
MatricaKompleksnih(2,3)==(0,10)
b2(1,1)==(1,2) b2(1,2)==(2,3) b2(1,3)==(5,6)
b2(2,1)==(2.3,4) b2(2,2)==(8,2.1) b2(2,3)==(0,10)
b3(1,1)==(1,2) b3(1,2)==(2,3) b3(1,3)==(5,6)
b3(2,1)==(2.3,4) b3(2,2)==(8,2.1) b3(2,3)==(0,10)

Zadatak 6. 3

Sastaviti šablonsku funkciju koja će eliminisati ponavljanje istih elemenata u nekom nizu (npr. ako imamo niz 2 3 4 3 2 1 5 daje niz 4 3 2 1 5), kao i šablonsku klasu koja će predstavljati niz sa zadatim opsezima indeksa. Napisati glavni program u kojem će se data

funkcija pozivati za niz cijelih i niz realnih brojeva kao i izvršiti inicijalizacija objekata klase niz za cijele i realne elemente nizova dobijenih izvršavanjem šablonske funkcije za niz cijelih ili realnih elemenata (4 3 2 1 5).

```
#include<iostream.h>
#include<conio.h>

template <class S>
int Sazimanje(S* niz,int duz){
      int pom;
      int n=0;
      for(int i=0;i<duz-1;i++){
            pom=0;
            for(int j=i+1;j<=duz-1;j++)
                  if(niz[i]==niz[j]){
                        pom=1;
                        break;
                        }
            if(pom==0) {niz[n]=niz[i]; n=n+1;}
            }
      niz[n]=niz[duz-1];
      return n+1;
      }

template <class T>
class Niz{
  private:
      T* sadrzaj;
      int poc;
      int kraj;
  public:
      Niz(){sadrzaj=0;}
      Niz(T*,int,int=0);
      Niz(const Niz&);
      ~Niz(){delete []sadrzaj; sadrzaj=0;}
      T*Daj_sadrzaj(){return sadrzaj;}
      Niz&operator=(const Niz&);
      T operator[](int i){return sadrzaj[i];}
      void citaj(){for(int i=0;i<kraj-poc+1;i++) cout<<sadrzaj[i]<<" ";cout<<endl;}
      };

template<class T>
Niz<T>::Niz(T * unos,int max,int min):poc(min),kraj(max){
      sadrzaj=new T[max-min+1];
      for(int i=0;i<max-min+1;i++) sadrzaj[i]=unos[i];
      }

template<class T>
Niz<T>::Niz(const Niz& stara):poc(stara.poc),kraj(stara.kraj){
      sadrzaj=new T[kraj-poc+1];
      for(int i=0;i<max-min+1;i++) sadrzaj[i]=stara.sadrzaj[i];
      }
```

```
template<class T>
Niz<T> &Niz<T>::operator=(const Niz<T>& stara){
      if(this!=&stara){
            delete []sadrzaj;
            poc=stara.poc;
            kraj=stara.kraj;
            sadrzaj=new T[kraj-poc+1];
            for(int i=0;i<kraj-poc+1;i++) sadrzaj[i]=stara.sadrzaj[i];
            }
      return *this;
      }

void main(){
     int niz[]={1,2,5,3,2,4,6,3,2};
     Niz<int> Cijeli(niz,10,2);
     float realni[]={1.5,2.4,5,3,2.4,4,6,3,2.4};
     Niz<float> Realni(realni,8);
     int n;
     n=Sazimanje(niz,9);
     for(int i=0;i<n;i++) cout<<niz[i]<<", "; cout<<endl;
     Sazimanje(realni,9);
     Cijeli.citaj();
     Realni.citaj();
     getch();
     }
```

U realizaciji ovog zadatka smo koristili ranije navedena pravila za šablonske funkcije i klase. Što se tiče same funkcije, kao rezultat dobijamo broj elemenata novog niza. Za eliminisanje ponavljanja koristimo pomoćnu promjenjivu koja ima vrijednost 0 ukoliko posmatranog elementa nema do kraja niza, uz taj uslov prepisuje se tako dobijeni element u sažeti niz. Ukoliko je njena vrijednost jednaka jedinici znači da se taj element već pojavljivao i da ga ne treba upisivati. To ćemo učiniti samo kod njegovog poslednjeg pojavljivanja.

Što se tiče klase, preklopili smo *operator[]* jer nam je moguć slučaj da niz objekata konkretne klase, generisane na osnovu date šablonske, bude argument neke generisane funkcije. U tom slučaju bi zahtjevali pristup određenom elementu niza, što smo preklapanjem ovog operatora omogućili. Ovdje smo preklopili i operator dodjele da bi se podsjetili načina na koji se to radi, iako ga nismo nigdje kasnije koristili.

Primjer rada programa

```
1, 5, 4, 6, 3, 2,
1 2 5 3 2 4 6 3 2
1.5 2.4 5 3 2.4 4 6 3 2.4
```

---

7. Izuzeci

Zadatak 7. 1

Projektovati klasu za obradu vektora realnih brojeva sa zadatim opsezima indeksa. Za razriješavanje konfliktnih situacija koristiti mehanizam obrade izuzetaka. Sastaviti glavni program za prikazivanjenje mogućnosti te klase.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class Vektor{
      private:
            float *niz;
            int max_ops;
            int min_ops;
      public:
           enum Greska{OK,     //Kodovi gresaka
                        OPSEG,  //neispravan opseg indeksiranja
                       MEMORIJA,//dodjela memorije nije uspjela,
                       PRAZAN,  //vektor je prazan
                       INDEKS,  //indeks je izvan opsega
                       DUZINA}; //neusaglasene duzine vektora
            Vektor(){niz=0;}
            Vektor(float, float=0);
            Vektor(const Vektor &);
            ~Vektor(){delete []niz;niz=0;}
            Vektor &operator=(const Vektor &v);
            float &operator[](int)const;  //referencu saljem kao rezultat da
            // bi se mogao mijenjati sadrzaj elementa kojem se pristupa pomocu [].
            friend double operator*(const Vektor&,const Vektor &);
            };

Vektor::Vektor(float a,float b):max_ops(a),min_ops(b){
      if(max_ops<min_ops) throw(OPSEG);
      else if(!(niz=new float[max_ops-min_ops+1])) throw MEMORIJA;
      else for(int i=0;i<max_ops-min_ops+1;i++) niz[i]=0;
      }

Vektor::Vektor(const Vektor &a):max_ops(a.max_ops),min_ops(a.min_ops){
      if(!(niz=new float[max_ops-min_ops+1])) throw MEMORIJA;
      else for(int i=0;i<max_ops-min_ops+1;i++) niz[i]=a.niz[i];
      }

Vektor &Vektor::operator=(const Vektor& a){
      if(&a!=this){
            delete []niz;
            niz=0;
            if(!(niz=new float[max_ops-min_ops+1])) throw MEMORIJA;
            else for(int i=0;i<max_ops-min_ops+1;i++) niz[i]=a.niz[i];
```

```
        }
    return *this;
    }

float &Vektor::operator[](int i)const{
    if(!niz) throw PRAZAN;//u niz nije nista upisano,
    // samo je izvrsen default konstruktor
    else if((i<min_ops)||(i>max_ops)) throw INDEKS;
    else return niz[i-min_ops];
    }

double operator*(const Vektor& a1,const Vektor& a2){
    if((!a1.niz)||(!a2.niz)) throw Vektor::PRAZAN;
    else {
        float s=0;
        for(int i=0;i<a1.max_ops-a1.min_ops+1;i++) s+=a1.niz[i]*a2.niz[i];
        return s;
        }
    }

main(){
    while(1){
        try{
            int max, min;
            cout<<"Unesi opseg indeksa prvog vektora"<<endl;
            cin>>max>>min;
            if(cin.eof()) throw 1;
            Vektor v1(max,min);
            cout<<"Komponente prvog vektora"<<endl;
            for(int i=min;i<=max;i++) cin>>v1[i];
            cout<<"Unesi opseg indeksa drugog vektora"<<endl;
            cin>>max>>min;
            Vektor v2(max,min);
            cout<<"Komponente drugog vektora"<<endl;
            for(int i=min;i<=max;i++) cin>>v2[i];
            cout<<"Skalarni proizvod dva zadata vektora je "<<v1*v2<<endl;
            }
            catch(Vektor::Greska g){
                char *poruke[]={"",
                                "Neispravan opseg indeksa!",
                                "Neuspjelo dodjeljivanje memorije!"
                                "Vektor je prazan!",
                                "Indeks je izvan opsega!"
                                };
                cout<<poruke[g]<<endl;
                }
            catch(...){
                cout<<"Kraj unosa"<<endl;
                break;
                }
        }

getch();getch();
}
```

Podsjetimo se, najprije, tipa nabrajanja *enum*. Ime koje smo dali ovom tipu je *Greska* i promjenjive ovog tipa mogu uzeti samo neku od vrijednosti navedenih unutar *{}*. Nabrajanjem smo, zapravo, definisali neke cjelobrojne konstante simboličkih imena *OK, OPSEG,….* Njima u okviru *{}* možemo dodijeliti tačno određene cjelobrojne vrijednosti. Ako to ne učinimo biće im dodijeljene vrijednosti od 0, pa 1 i tako dalje, svaka sledeća ima za jedan veću vrijednost. Identifikator njihovog tipa je *Greska*, a ne *int*, iako imaju cjelobrojnu vrijednost. Znači, *Greska* je ime novog tipa a *OK, OPSEG...*, vrijednosti koje neka promjenjiva tog tipa može imati, isto kao *float* ime tipa, a 2.4 vrijednost koju promijenjiva tog tipa može uzeti.

U ovom zadatku nam je cilj da ilustrujemo generisanje i obradu izuzetaka. Kada se radi sa izuzecima mogu se posmatrati dvije odvojene faze: prva je otkrivanje i generisanje izuzetaka, a druga obrada datog izuzetka. Kada otkrijemo mogućnost pojave neke greške u programu, koja nije sintaksnog i semantičkog tipa, a želimo da ostavimo mogućnost njene obrade, a ne samo prekidanje programa bez ikakvog objašnjenja, možemo postaviti izuzetak. Izuzetak se generiše ključnom riječju *throw* nakon koje pišemo neki podatak određenog tipa (3, 2.4..) koji će biti identifikator tog izuzetka. Naime, ukoliko smo napisali *throw 3,* nastao je izuzetak cjelobrojnog tipa. Izuzetak se prihvata ključnom riječju *catch(),* gdje u zagradama pišemo tip izuzetka koji se tu obrađuje, *catch(int)* u prethodnom slučaju. Dio programa u kojem može nastati izuzetak stavljamo iza ključne riječi *try* u okviru *{}* nakon kojih se nalaze ključne riječi *catch()* koje prihvataju i obrađuju određene izuzetke.

Što se tiče same klase, ona predstavlja niz kod kojeg je tačno definisan opseg, odnosno, minimalna i maksimalna moguća vrijednost indeksa. Greška će nastati već prilikom formiranja niza kada korisnik želi inicijalizovati objekat tako da ima maksimalni opseg manji od minimalnog. To smo uzeli kao drugu vrijednost koju mogu imati podaci tipa *Greska-OPSEG*. Na mjestima na kojima može doći do ovakve greške postavili smo izuzetak *throw OPSEG*. Druga greška nastaje kada želimo da alociramo memoriju za taj niz ali u dinamičkoj zoni memorije nema dovoljno mjesta za cio niz, postavljamo izuzetak *throw MEMORIJA*. Vidimo da će i jedan i drugi da prihvata isti rukovalac *catch* tipa *Greska* ali će u zavisnosti od vrijednosti da odradi različite akcije. Ako *g* ima vrijednost 1 (*OPSEG*), čita *g[1]* string, odnosno, za *g=2* (*MEMORIJA*) čita *g[2]* string. Sledeća greška može nastati prilikom čitanja, ukoliko želimo uzeti nešto iz praznog niza *throw PRAZAN* ili čitati nešto iz elementa sa indeksom van opsega tu postavljamo izuzetak *throw INDEKS.* Još jedan izuzetak može nastati kada želimo preklopiti operator * za dva niza različitih dužina, *throw DUZINA*. Gdje i na koji način biste ga postavili.

Ukoliko u rukovaocu izuzetka ne koristimo vrijednost koja mu je proslijeđena nakon identifikatora tipa ne moramo navoditi ime promjenjive u koju će se kopirati poslata vrijednost, *catch(int).* U tom slučaju nam ta vrijednost treba samo da na osnovu njenog tipa vidimo koji rukovalac će prihvatiti naš izuzetak (odgovara našem izuzetku).

Prođimo jednom kroz cio kod. Imamo klasu *Vektor* sa tri privatna podatka člana, dva nam određuju maksimalnu i minimalnu moguću vrijednost indeksa, a treći predstavlja pokazivač na početak niza. Zatim imamo jedan javni podatak tipa nabrajanja *Greska* koji može imati jednu od vrijednosti navedenih unutar *{}*. Default konstrukor je definisan na ranije korišćen način. Već prilikom definicije prvog konstruktora može doći do greške. Greška je ukoliko zadamo da je vrijednost najmanjeg indeksa manja od vrijednosti najvećeg. Tu postavljamo izuzetak *throw OPSEG*. Njegov tip je *Vektor::Greska* pa ga prihvata rukovalac *throw(Vektor::Greska g)* U ovom slučaju će *g* imati vrijednost 1 i ispisaće se drugi string iz niza stringova na koji ukazuje pokazivač *poruke*. Sledeća greška nastaje ako nije ispravno alocirana memorija, postavljamo odgovarajući izuzetak. Ako je sve u redu inicijalizujemo sve elemente niza na nulu. U konstruktoru kopije nastaje greška ukoliko u dinamičkoj zoni memorije nema dovoljno mjesta za smještanje svih elemenata niza. Ista greška može nastati i prilikom realizacije operatora dodjele

pa smo i u okviru njega postavili isti izuzetak. Prilikom realizacije operatora indeksiranja, *[]*, može nastati greška ukoliko je niz prazan a mi želimo nešto isčitati iz njega. To bi se desilo ukoliko smo neki objekat samo deklarisali gdje je pozvan default konstruktor koji je postavio odgovarajući pokazivač na 0, nije alocirao memoriju niti je odredio opseg vrijednosti indeksa niza. Operator indeksiranja smo realizovali kao u ranijim primjerima. S obzirom da nam najmanji indeks ne mora biti 0 već ima vrijednost smještenu u podatku *min_ops*, kada korisnik želi element sa indeksom *i* mi mu dajemo element koji je smješten na poziciji *i-min_ops* iako korisnik misli da je to element sa indeksom *i*. Ovdje nastaje greška ukoliko je indeks tražene komponente van dozvoljenog opsega. Operator * nam predstavlja skalarni proizvod dva vektora, greška nastupa ukoliko je jedan od vektora prazan. Pokušajte da postavite izuzetak koji bi javljao grešku kada je različita dužina dva vektora koje "množimo". Ovdje smo morali navesti i klasu kojoj pripada simbolička konstanta kojom postavljamo izuzetak jer je ovo funkcija prijatelj klase, a ne članica, pa ona ima pravo pristupa tim podacima ali se mora navesti klasa čiji je taj podatak član. U glavnom programu se nalazi jedna petlja *while(1)* i izvršavaće se sve dok umjesto opsega indeksa prvog vektora ne unesemo znak za kraj datoteke *CTRL-Z*, nakon čega dolazi do izuzetka tipa *int* čiji će rukovalac izvršiti prekidanje *while* petlje (naredba *break;*). Koristili smo ovu petlju da bismo mogli u okviru jedne realizacije programa generisati više izuzetaka. Da nje nema nakon prvog izuzetka došlo bi do preskoka na prvog odgovarajućeg rukovaoca nakon *try* i ne bismo se više vraćali nazad. Išlo bi se na prvu naredbu nakon *catch* dijela a to bi bio kraj programa. Sada je to sledeći ciklus u petlji, dakle unošenje novih podataka.

Primjer rada programa

Unesi opseg indeksa prvog vektora
1 3
Neispravan opseg indeksa!
Unesi opseg indeksa prvog vektora
3 1
Komponente prvog vektora
1 4 5
Unesi opseg indeksa drugog vektora
3 1
Komponente drugog vektora
4 3 2
Skalarni proizvod dva zadata vektora je 26
Unesi opseg indeksa prvog vektora
Kraj unosa

L I T E R A T U R A

[1] D. Milićev: "Objektno-orjentisano programiranje na jeziku C++," Mikroknjiga, 1995.

[2] H. M. Deitel, P. J. Deitel: "C++ how to program," Prentice Hall, 1997.

[3] J. Liberty, D. B. Horvath: "C++ za LINUX," SAMS Publishing (2000), Kompjuter biblioteka (prevedeno izdanje) 2002.

[4] L. Kraus: "Programski jezik C++ sa rešenim zadacima," Mikro knjiga, 1994.

[5] K. Reisdorph: "Teach yourself C++Builder in 21 day", SAMS Publishing.

[6] D. Milićev, LJ. Lazarević, J. Marušić: "Objektno orjentisano programiranje na jeziku C++ - Skripta sa praktikumom," Mikro knjiga 2001.

[7] N. Wirth: "Algorithms + Data structures = Programs," Prentice Hall, 1976.

[8] G. Booch, J. Rumbaugh, I. Jacobson: "UML – vodič za korisnike," Addison Wesley 1999, CET 2000 (prevedeno izdanje).

[9] Jan Skansholm, "C++ from the beginning",.Addison-Wesley, 2003.