УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

Александар Купусинац

ЗБИРКА РЕШЕНИХ ЗАДАТАКА ИЗ ПРОГРАМСКОГ ЈЕЗИКА C++

Нови Сад, 2012.

PREDGOVOR

Zbirka je namenjena studentima druge godine elektrotehnike i računarstva Fakulteta tehničkih nauka u Novom Sadu i kao pomoćni udžbenik obrađuje one delove gradiva koji su potrebni za praćenje vežbi iz programskog jezika C++. Zbirka predstavlja prateći materijal udžbeniku:

> Malbaški, D. : *Objektno orijentisano programiranje kroz programski jezik C++*, Novi Sad: Fakultet tehničkih nauka, 2008.

i zajedno čine celinu koja je potrebna za savladavanje gradiva i spremanje ispita iz predmeta Objektno programiranje i Objektno orijentisano programiranje.

Gradivo je izloženo postupno i obuhvata širok spektar tema iz ove oblasti. Svako poglavlje počinje kratkim teorijskim uvodom. U prvom poglavlju se ukratko izlažu najbitniji delovi gradiva iz programskog jezika C. Drugo poglavlje se bavi delovima programskog jezika C++ koji nisu objektno-orijentisani i koji se mogu shvatiti kao proširenje programskog jezika C. Treće poglavlje upoznaje čitaoca sa realizacijom klase u programskom jeziku C++. Mehanizam preklapanja operatora je izložen u četvrtom poglavlju. Peto i šesto poglavlje razmatraju veze između klasa (klijentske veze i nasleđivanje). U sedmom poglavlju je opisan način realizacije generičkih klasa. Prevencijom otkaza, tačnije rukovanjem izuzecima, bavi se osmo poglavlje. Najzad, u dodatku je ukratko opisan način rada sa programskim okruženjem `Code::Blocks`.

Zahvaljujem se recezentima na korisnim sugestijama koje su uticale da ova zbirka bude još razumljivija i bliža čitaocu.

Pozivam sve čitaoce da svoja zapažanja, primedbe i korekcije pošalju na adresu `sasak@uns.ac.rs` i za to im se unapred zahvaljujem.

U Novom Sadu, avgusta 2012.

Autor

SADRŽAJ

1. PROGRAMSKI JEZIK C

U ovom poglavlju ukratko ćemo ponoviti gradivo iz programskog jezika C i to one teme koje će nam biti neophodne u daljem izlaganju.

1.1 Promenljive

- Podaci mogu biti: promenljive i konstante.
- Deklaracijom se zauzima prostor u memoriji za datu promenljivu, a definicijom se pored zauzimanja memorije vrši još i postavljanje inicijalne vrednosti. Na primer:

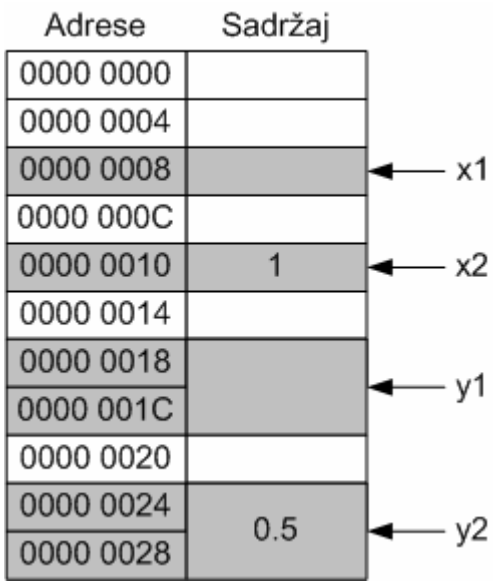
```
int x1;           // deklaracija
int x2=1;         // definicija
double y1;        // deklaracija
double y2=0.5;    // definicija
```

- Tip promenljive određuje koliko će biti memorije zauzeto za datu promenljivu, pri čemu to zavisi od hardvera koji nam je na raspolaganju. Na primer, tip `int` je osnovni celobrojni tip koji može da zauzima 16, ali i 32 bita, što najčešće zavisi od hardvera. Tip `double` je realni tip u dvostrukoj tačnosti i tipično zauzima 64 bita.
- Postoje standardni tipovi (`char`, `int`, `float` itd.) i korisnički tipovi koje programer uvodi naredbom `typedef`. Na primer:

```
typedef int CeoBroj;
```

- Ime promenljive mora biti jedinstveno. Ime promenljive jeste simbolički predstavljena adresa na kojoj se nalazi memorija koja je zauzeta za datu promenljivu. Na primer, neka su memorijske adrese i lokacije veličine 32-bita, tada će memorija posle gore navedenih deklaracija i definicija imati izgled kao što to prikazuje slika 1. Adrese su napisane u heksadecimalnom obliku. Svaka lokacija sadrži 4 bajta i svaki bajt je adresibilan, pa se zbog toga adrese razlikuju za 4. Vidimo da će za promenljive `x1` i `x2` biti zauzeta po jedna lokacija, tj. po 32 bita, jer su obe promenljive tipa `int`, dok za promenljive `y1` i `y2` po 64 bita. Sadržaji promenljivih `x2` i `y2` će biti postavljeni na odgovarajuće inicijalne vrednosti. Posmatrajući sliku zaključujemo da kada se u višem programskom jeziku *govori* o promenljivoj `x1`, računar to tako *razume* da se zapravo radi o promenljivoj na adresi `1C`.
- Unarnim operatorom `sizeof` može se odrediti veličina memorije (u bajtima) koja je potrebna za smeštanje podataka određenog tipa. Na primer:

```
sizeof(int);
sizeof(TTip);
```



Slika 1.

1.2 Pokazivači

- Pokazivač je promenljiva koja sadrži neku adresu.
- Pokazivači se deklarišu na ovaj način:

```
int *iPok;        // pokazivac na podatak tipa int
double *dPok;     // pokazivac na podatak tipa double
void *vPok;       // genericki pokazivac
```

- Pošto pokazivači sadrže adrese, zaključujemo da bez obzira na tip podataka na koje pokazuju, svi pokazivači zauzimaju jednak prostor u memoriji. Na primer, ako su 32-bitne adrese tada će svaki pokazivač zauzimati po 32 bita.
- Kada kažemo da pokazivač *pokazuje* na neki podatak, to zapravo znači da taj pokazivač sadrži adresu datog podatka.
- Unarni operator `&` vraća memorijsku adresu nekog podatka. Unarni operator `*` omogućava da se posredno pristupi nekom podatku pomoću memorijske adrese. Na primer:

```
int x;      // deklaracija promenljive x
int *px;    // deklaracija pokazivaca px
px=&x;      // pokazivac px dobija adresu promenljive x
*px=10;     // promenljiva x dobija vrednost 10
```

Zadatak 1.2.1
Analizirati ispis sledećeg programa:

```
#include <stdio.h>
int main() {
    printf("Velicina memorije (izrazena u bitima) iznosi:");
    printf("\n-za char \t %d",8*sizeof(char));
    printf("\n-za int \t %d",8*sizeof(int));
    printf("\n-za double \t %d",8*sizeof(double));
    printf("\n-za char* \t %d",8*sizeof(char*));
    printf("\n-za int* \t %d",8*sizeof(int*));
    printf("\n-za double* \t %d",8*sizeof(double*));
    printf("\n-za void* \t %d",8*sizeof(void*));
    return 0;
}
```

Operator `sizeof` vraća veličinu memorije koja je potrebna za smeštanje podataka datog tipa, izraženu u bajtima. Pokretanjem programa dobija se sledeći ispis:

```
Velicina memorije (izrazena u bitima) iznosi:
-za char         8
-za int          32
-za double       64
-za char*        32
-za int*         32
-za double*      32
-za void*        32
```

Vidimo da je za smeštanje podataka tipa `char` potrebno 8 bita, za `int` 32 bita, za `double` 64 bita, dok je za pokazivačke tipove `char*`, `int*`, `double*` i `void*` potrebno po 32 bita. Ovaj primer pokazuje da svi pokazivači zauzimaju jednak prostor u memoriji, bez obzira na tip podataka na koje pokazuju, tj. svi pokazivači sadrže adrese koje su veličine 32 bita.

Zadatak 1.2.2
Analizirati ispis sledećeg programa:

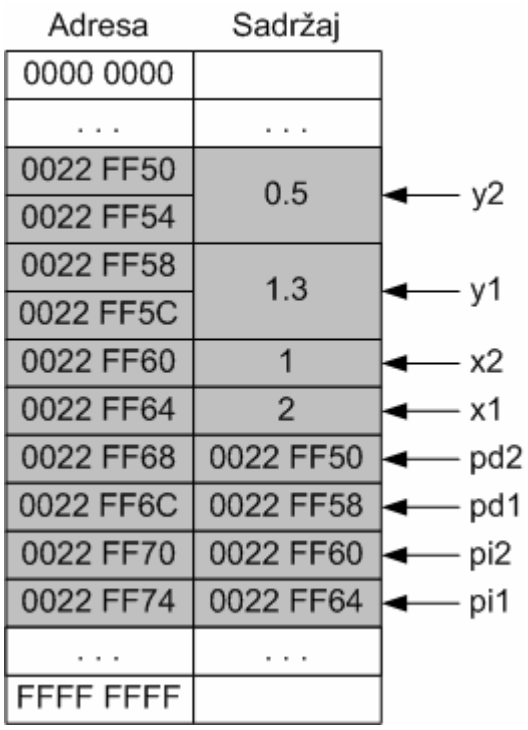
```
#include <stdio.h>
int main(){
    int *pi1;
    int *pi2;
    double *pd1;
    double *pd2;
    printf("\nAdresa pokazivaca pi1: %d",&pi1);
    printf("\nAdresa pokazivaca pi2: %d",&pi2);
    printf("\nAdresa pokazivaca pd1: %d",&pd1);
    printf("\nAdresa pokazivaca pd2: %d",&pd2);
    printf("\n------------------------------------------------\n");
    int x1;
    int x2=1;
```

```
    double y1;
    double y2=0.5;
    pi1=&x1;
    pi2=&x2;
    pd1=&y1;
    pd2=&y2;
    printf("\nSadrzaj pokazivaca pi1: %d",pi1);
    printf("\nSadrzaj pokazivaca pi2: %d",pi2);
    printf("\nSadrzaj pokazivaca pd1: %d",pd1);
    printf("\nSadrzaj pokazivaca pd2: %d",pd2);
    printf("\n-------------------------------------------------\n");
    *pi1=2;
    *pd1=1.3;
    printf("\nVrednost promenljive x1: %d",*pi1);
    printf("\nVrednost promenljive x2: %d",*pi2);
    printf("\nVrednost promenljive y1: %f",*pd1);
    printf("\nVrednost promenljive y2: %f",*pd2);
    printf("\n-------------------------------------------------\n");
    return 0;
}
```

Pokretanjem programa dobijamo sledeći ispis:

```
Adresa pokazivaca pi1: 22ff74
Adresa pokazivaca pi2: 22ff70
Adresa pokazivaca pd1: 22ff6c
Adresa pokazivaca pd2: 22ff68
-------------------------------------------------------

Sadrzaj pokazivaca pi1: 22ff64
Sadrzaj pokazivaca pi2: 22ff60
Sadrzaj pokazivaca pd1: 22ff58
Sadrzaj pokazivaca pd2: 22ff50
-------------------------------------------------------

Vrednost promenljive x1: 2
Vrednost promenljive x2: 1
Vrednost promenljive y1: 1.300000
Vrednost promenljive y2: 0.500000
-------------------------------------------------------
```

Ovde treba napomenuti da dobijeni ispis ovog programa ne mora uvek biti isti, jer izbor slobodnih memorijskih lokacija u kojima će se naći programske promenljive se vrši proizvoljno. Na osnovu gornjeg ispisa možemo zamisliti da će memorija imati izgled kao što to prikazuje slika 2. Promenljive `y1` i `y2` zauzimaju po 64 bita, jer su tipa `double`, dok promenljive `x1` i `x2` po 32 bita, jer su tipa `int`. Pokazivači `pd1`, `pd2`, `pi1` i `pi2` zauzimaju po 32 bita.



Slika 2.

1.3 Nizovi

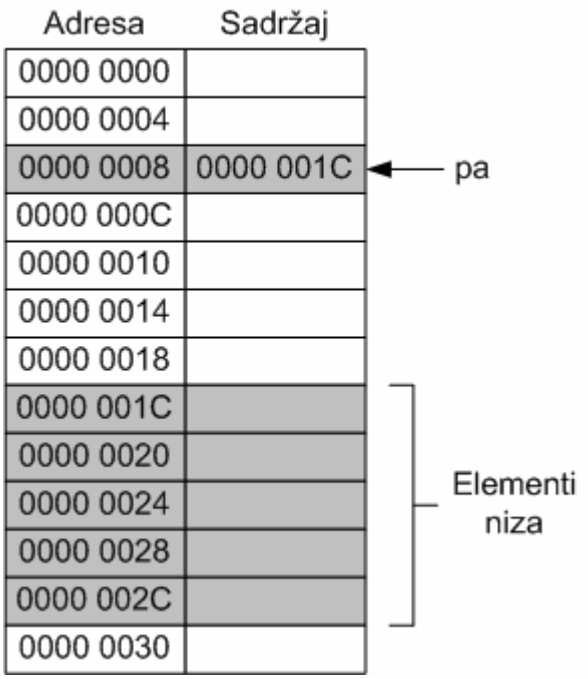
- Elementi niza su istog tipa, a identifikuju se na osnovu rednog broja.
- Memorija za smeštanje elemenata niza može biti zauzeta na statički i dinamički način i shodno tome razlikujemo statičke i dinamičke nizove.
- Statički niz deklarišemo na sledeći način:

```
int a[5];          // niz 5 elemenata tipa int
double b[10];      // niz 10 elemenata tipa double
```

- Ime statičkog niza je konstantan pokazivač i predstavlja početnu adresu niza, tj. adresu prvog elementa niza.
- Dinamički niz dobijamo dinamičkom alokacijom memorije i njenim vezivanjem za neki pokazivač, tako što se adresa alocirane memorije smešta u dati pokazivač. Za dinamičku alokaciju memorije koristimo funkciju `malloc()` koja kao povratnu vrednost vraća adresu alocirane memorije koja je tipa `void*`, pa ukoliko se, na primer, kreira niz tipa `int` potrebno je uraditi eksplicitnu promenu tipa `void*` u tip `int*`.

```
int *pa;     // deklaracija pokazivaca pa
pa=(int*)malloc(5*sizeof(int)); // din. alok. memorije
```

- Na slici 3. je prikazana organizacija dinamičkog niza tipa `int` koji ima 5 elemenata.



Slika 3.

- `i`-tom elementu se može pristupiti sa `a[i]`, odnosno `*(pa+i)`.
- Indeksiranje niza počinje sa brojem 0.

Zadatak 1.3.1
Napisati program koji određuje maksimalni element celobrojnog niza (niz može imati najviše 30 elemenata).

```
#include <stdio.h>
int main() {

    int a[30], n=0, i, max;

    while(n<=0 || n>30) {
        printf("Unesite broj elemenata niza: ");
        scanf("%d",&n);
    }

    for(i=0; i<n; i++) {
        printf("Unesite broj a[%d",i);
        printf("]= ");
        scanf("%d",&a[i]);
    }
```

```
    max=a[0];
    for(i=1; i<n; i++)
        if(a[i]>max)
            max=a[i];

    printf("Maksimalni element niza je: %d",max);

    return 0;
}
```

Zadatak 1.3.2
Napisati program za sortiranje dinamičkog niza celih brojeva u neopadajućem redosledu.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *a;
    int n=0, i, j, t;
    while(n<=0) {
        printf("Unesite broj elemenata niza: ");
        scanf("%d", &n);
    }

    if((a=(int*)malloc(n*sizeof(int)))==NULL)
        exit(0);

    for(i=0; i<n; i++) {
        printf("Unesite broj a[%d",i);
        printf("]= ");
        scanf("%d",a+i);
    }

    for(i=0; i<n-1; i++)
        for(j=i+1; j<n; j++)
            if( *(a+i) > *(a+j) ) {
                t=*(a+i);
                *(a+i)=*(a+j);
                *(a+j)=t;
            }

    printf("Sortiran niz po neopadajucem redosledu: \n");
    for(i=0; i<n; i++) {
        printf("a[%d",i);
        printf("]= %d \n",*(a+i));
    }
    return 0;
}
```

1.4 Funkcije

- Potprogrami predstavljaju mehanizam pomoću kojeg se složeni problemi razbijaju na jednostavnije potprobleme. Programski jezik C formalno poznaje samo jednu vrstu potprograma, a to su **funkcije**.
- Funkcije su potprogrami koji na osnovu argumenata daju jedan rezultat koji se naziva **povratna vrednost funkcije**.
- Deklaracija funkcije ili prototip:

```
Tip ime(TipArg1, TipArg2,..., TipArgN);
```

- Definicija funkcije:

```
Tip ime(TipArg1 arg1, TipArg2 agr2,..., TipArgN argN) {
      // TELO FUNKCIJE
}
```

- Poziv funkcije:

```
ime(arg1, agr2,..., argN);
```

- Razlikujemo prenos argumenata po vrednosti i po adresi.

Zadatak 1.4.1
Analizirati ispis sledećeg programa:

```
#include <stdio.h>

// PROTOTIPOVI FUNKCIJA
void f1(int);
void f2(int*);

int main() {
    int x=5;
    printf("Promenljiva x je: %d\n",x);

    // POZIV FUNKCIJE f1()
    f1(x); // Prenos po vrednosti
    printf("Promenljiva x je: %d\n",x);

    // POZIV FUNKCIJE f2()
    f2(&x); // Prenos po adresi
    printf("Promenljiva x je: %d\n",x);

    return 0;
}

// DEFINICIJE FUNKCIJA
void f1(int a) { a=3; }
void f2(int *a) { *a=3; }
```

Pokretanjem programa dobijamo sledeći ispis:

```
Promenljiva x je: 5
Promenljiva x je: 5
Promenljiva x je: 3
```

Zadatak 1.4.2

Napisati program za rad sa jednostrukospregnutom listom celih brojeva.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct cvor {
    int info;
    struct cvor *next;
    } TCvor;

int unosElementa(TCvor **phead, int el, int pos) {
    TCvor *temp;
    int brojac=1;
    int i;

    temp=*phead;
    while(temp!=NULL) {
        brojac++;
        temp=temp->next;
    }

    if(pos<1 || pos>brojac+1)
        return 0;
    else {
        TCvor *novi;
        novi = (TCvor *)malloc(sizeof(TCvor));
        if(novi==NULL)
            return 0;

        novi->info=el;

        if(pos==1) {
            novi->next=*phead;
            *phead=novi;
        }
        else {
            temp=*phead;
            for(i=1; i<pos-1; i++)
                temp=temp->next;
            novi->next=temp->next;
            temp->next=novi;
        }
        return 1;
    }
}
```

```
int brisiElement(TCvor **phead, int pos) {
    TCvor *temp;
    TCvor *del;
    int brojac=1;
    int i;

    if(*phead==NULL)
        return 0;

    temp=*phead;
    while(temp!=NULL) {
        brojac++;
        temp=temp->next;
    }

    if(pos<1 || pos>brojac)
        return 0;
    else {
         if(pos==1) {
             del=*phead;
            (*phead)=(*phead)->next;
            free(del);
        }
        else {
            temp=*phead;
            for(i=1; i<pos-1; i++)
                temp=temp->next;
            del=temp->next;
            temp->next=del->next;
            free(del);
        }
        return 1;
    }
}


int ispisiListu(TCvor *head) {
    TCvor *temp = head;
    if(head==NULL)
        return 0;
    while(temp!=NULL) {
        printf("%d",temp->info);
        temp=temp->next;
    }
    return 1;
}


void obrisiListu(TCvor **phead) {
    while((*phead)!=NULL)
        brisiElement(phead,1);
    printf("\nLista je obrisana!");
}
```

```
int meni() {
    int x;
    printf("\n\n\n\nIzaberite opciju: ");
    printf("\n1. Unos elementa u listu");
    printf("\n2. Brisanje elementa iz liste");
    printf("\n3. Ispis liste");
    printf("\n4. Obrisi listu");
    printf("\n5. Kraj rada");
    printf("\n\nVas izbor je -> ");
    do {
        scanf("%d",&x);
    } while(x<1 || x>5);
    return x;
}

int main() {
    TCvor *head=NULL;
    int odg, el, pos;
    do {
        odg=meni();
        switch(odg) {
            case 1 : printf("\nUnesite novi element liste -> ");
                     scanf("%d",&el);
                     printf("\nUnesite poziciju -> ");
                     scanf("%d",&pos);
                     if(unosElementa(&head,el,pos))
                        printf("\nElement je unet u listu!");
                     else
                        printf("\nElement nije unet u listu!");
                     break;

            case 2 : printf("\nUnesite poziciju -> ");
                     scanf("%d",&pos);
                     if(brisiElement(&head, pos))
                        printf("\nElement je izbrisan!");
                     else
                        printf("\nElement nije izbrisan!");
                     break;

            case 3 : if(ispisiListu(head)==0)
                        printf("\nLista je prazna!");
                     break;
            case 4 : obrisiListu(&head);
        }
    } while(odg!=5);
    obrisiListu(&head);
    return 0;
}
```

Zadatak 1.4.3
Realizovati *Quick Sort* algoritam.

```
#include <stdio.h>
#define MAXEL 100

void swap(int *a, int *b) {
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

int div(int *a, int n) {
    int b=a[n-1], i=0, j=n-1;
    while(i<j) {
        while(a[i]<b && i<=n-1)
            i++;
        while(a[j]>=b && j>=0)
            j--;
        if(i<j)
            swap(a+i, a+j);
    }
    swap(a+i, a+n-1);
    return i;
}

void qsort(int *a, int n) {
    if(n>1) {
        int i;
        i=div(a,n);
        qsort(a,i);
        qsort(a+i+1, n-i-1);
    }
}

int main() {
    int a[MAXEL];
    int i, n;
    printf("\n Unesite broj elemenata --> ");
    scanf("%d", &n);
    printf("\n Unesite elemente:\n");
    for(i=0; i<n; i++) {
        printf(" a[%d] --> ", i);
        scanf("%d",&a[i]);
    }
    qsort(a,n);
    printf("\n Sortiran niz:\n");
    for(i=0; i<n; i++)
        printf(" %d",a[i]);
    return 0;
}
```

2. UVOD U PROGRAMSKI JEZIK C++

U ovom poglavlju upoznaćemo se sa delovima programskog jezika C++ koji ne pripadaju objektno-orijentisanoj metodologiji, već se mogu shvatiti kao proširenje programskog jezika C.

2.1 Reference

- Referenca ili upućivač u programskom jeziku C++ je alternativno ime za neki podatak (drugo ime za dati podatak).
- Reference nisu podaci. Reference ne zauzimaju prostor u memoriji, pa se ne može tražiti njihova adresa. Reference se pridružuju nekom podatku (promenljivoj ili konstanti) koji se već nalazi u memoriji (na nekoj adresi).
- Ne postoje pokazivači na reference. Ne postoje nizovi referenci.
- Prilikom definisanja, reference moraju da se inicijalizuju nekim podatkom koji se već nalazi na nekoj adresi u memoriji.
- Referenca je čvrsto vezana za podatak. Sve izvršene operacije nad referencama deluju na originalne podatke.
- Referenca se definiše na ovaj način:

```
int x=5;    // promenljiva x
int &rx=x;  // rx je referenca na promenljivu x
double y1;
double y2;
double &ry;       // ne moze!
double &ry=y1;    // moze!
```

- Reference služe da bi se izbeglo korišćenje pokazivača. Mogu se uočiti jasne razlike između referenci i pokazivača.

Sličnost između reference i pokazivača

- Referenca je adresa, odnosno drugo ime za originalni podatak, dok pokazivač je promenljiva koja sadrži adresu. Dakle, njihova ostvarenja su preko adresa podataka kojima su pridruženi.

Razlike između reference i pokazivača

- Referenca je čvrsto vezana za podatak, dok vrednost pokazivača nije, tj. pokazivač može da sadrži adresu jednog podatka i nakon toga može da se promeni njegova vrednost i da onda pokazuje na drugi podatak.

- Dok kod svakog pominjanja reference podrazumeva se posredan pristup podatku na kojeg upućuje, kod pokazivača je potrebno koristiti operator za indirektno adresiranje `*`.
- Dok pokazivači jesu pravi podaci koji imaju adresu, zauzimaju određeni prostor u memoriji i može im se po potrebi promeniti vrednost, reference nisu podaci i treba ih shvatiti samo kao alternativno ime koje se čvrsto vezuje za neki podatak koji već postoji na nekoj adresi u memoriji.

Zadatak 2.1.1
Analizirati ispis sledećeg programa:

```
#include <iostream>
using namespace std;
int main() {
    int x=5;
    int &rx=x;
    int *y;
    y=&x;
    cout<<"Adresa od x je: "<<&x<<endl;
    cout<<"Adresa od rx je: "<<&rx<<endl;
    cout<<"Adresa pokazivaca y je: "<<&y<<endl;
    cout<<"Sadrzaj pokazivaca y je adresa: "<<y<<endl;

    cout<<"x="<<x<<endl;
    rx++; // promena originala x preko reference rx
    cout<<"x="<<x<<endl;
    *y=10; // promena originala x preko pokazivaca y
    cout<<"x="<<x<<endl;
    rx=3;  // ponovo, promena originala x preko reference rx
    cout<<"x="<<x<<endl;

    return 0;
}
```

Uz napomenu da treba očekivati da se dobijene adrese mogu razlikovati na različitim računarima, pokretanjem programa dobijamo sledeći ispis:

```
Adresa od x je: 0x22ff74
Adresa od rx je: 0x22ff74
Adresa pokazivaca y je: 0x22ff6c
Sadrzaj pokazivaca y je: 0x22ff74
x=5
x=6
x=10
x=3
```

2.2 Ulaz i izlaz podataka

- U programskom jeziku C++ dodeljena su nova značenja operatorima `<<` i `>>`. Značenje tih operatora je ostalo isto ukoliko su oba operanda celobrojne vrednosti (služe za pomeranje u levo/desno prvog operanda za onoliko binarnih mesta kolika je vrednost drugog operanda), međutim, ukoliko je prvi operand referenca na tekstualnu datoteku onda te operatore koristimo za ulaz/izlaz podataka.
- Ukoliko je prvi operand operatora `>>` referenca na tekstualnu datoteku tada se operator koristi za čitanje jednog podatka iz te datoteke i smeštanje u drugi operand, uz primenu ulazne konverzije koja odgovara tipu drugog operanda. Na primer, ako je `in` referenca na datoteku, a `x` promenljiva tipa `int`, tada se izrazom `in>>x` čita iz datoteke jedan podatak tipa `int` i smešta u programsku promenljivu `x`.
- Ukoliko je prvi operand operatora `<<` referenca na tekstualnu datoteku tada se operator koristi za upisivanje vrednosti drugog operanda u datu datoteku, uz primenu izlazne konverzije koja odgovara tipu drugog operanda. Na primer, ako je `out` referenca na datoteku, a `x` promenljiva tipa `int`, tada se izrazom `out<<x` upisuje u datoteku vrednost programske promenljive `x`.
- Referenca na glavni ulaz računara (obično je to tastatura) ima identifikator `cin`, a referenca na glavni izlaz računara (obično je to ekran) ima identifikator `cout`.
- Za prelazak u novi red koristi se manipulator `endl`, ali isto tako može da se koristi i karakter `\n`.
- Ispis teksta "`Vrednost promenljive x je: `", a zatim ispis vrednosti promenljive `x` i prelazak u novi red izgleda ovako:

```
cout<<"Vrednost promenljive x je: "<<x<<endl;
```

  ili

```
cout<<"Vrednost promenljive x je: "<<x<<"\n";
```

- Neka su `x` i `y` promenljive tipa `int`, tada unos njihovih vrednosti izgleda ovako:

```
cin>>x>>y;
```

- Potrebne deklaracije za primenu operatora `<<` i `>>` se nalaze u zaglavlju `<iostream>` (o čemu će biti više reči u daljem izlaganju), pa će zbog toga naši programi počinjati sa:

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

Zadatak 2.2.1
Napisati program koji izračunava zbir dva broja.

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

int main() {
    int x, y;
    cout<<"Ovo je program za racunanje zbira dva broja."<<endl;
    cout<<"Unesite brojeve... "<<endl;
    cin>>x>>y;

    cout<<"Broj x je: "<<x<<endl;
    cout<<"Broj y je: "<<y<<endl;
    cout<<"Njihov zbir je: "<<x+y<<endl;
    return 0;
}
```

Zadatak 2.2.2
Napisati program koji formira niz od prvih `n` Fibonačijevih brojeva, pri čemu je poznato da je `n≤100`.

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

int main() {
    int i, n;
    int f[100];
    cout<<"Unesite prirodan broj n, n<=100... "<<endl;
    cin>>n;
    f[0]=1;
    if(n>1)
        f[1]=1;
    if(n>2)
        for(i=2; i<n; i++)
            f[i]=f[i-1]+f[i-2];
    cout<<"Ispis Fibonacijevih brojeva... "<<endl;
    for(i=0; i<n; i++)
        cout<<"f["<<i<<"] = "<<f[i]<<endl;
    return 0;
}
```

Zadatak 2.2.3
Napisati program koji za uneta dva broja `a` i `b` ispisuje veći broj.

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

int main() {
    int a, b;
```

```
    cout<<"Unesite prvi broj --> ";
    cin>>a;
    cout<<"Unesite drugi broj --> ";
    cin>>b;
    if(a>b)
        cout<<"Veci broj je: "<<a<<endl;
    else
        if(b>a)
            cout<<"Veci broj je: "<<b<<endl;
        else
            cout<<"Brojevi su jednaki"<<endl;
    return 0;
}
```

Zadatak 2.2.4
Napisati program koji ispituje da li je dati broj `n` prost broj.

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

#include <math.h>

int main() {
    int n, i;
    double x;
    cout<<"Unesite broj --> ";
    cin>>n;
    i=2;
    x=sqrt(n);
    while(i<=x) {
        if(n%i==0) {
            cout<<"Broj "<<n<<" nije prost."<<endl;
            return 0;
        }
        i++;
    }
    cout<<"Broj "<<n<<" je prost."<<endl;
    return 0;
}
```

2.3 Podrazumevane vrednosti funkcija

- U programskom jeziku C++ postoji mogućnost da se u definiciji funkcije navedu podrazumevane vrednosti formalnih argumenata. Ukoliko pri pozivu tako napisane funkcije nedostaju stvarni argumenti, tada se koriste podrazumevane vrednosti. Na primer:

```
void f(int x=2, int y=3) { . . . }
```

- U listi argumenata uvek se nalaze prvo oni argumenti koji nemaju podrazumevane vrednosti, a zatim oni koji imaju. Na primer:

```
void f(int x, int y=3, int z=4) { . . . } // moze!
void f(int x, int y=3, int z) { . . . }   // ne moze!
```

Zadatak 2.3.1
Potrebno je postaviti pločice na određenoj površini. Napisati program koji na osnovu dimenzija površine koja se popločava izračunava broj potrebnih pločica. Pločice su oblika kvadrata, a ukoliko se ne zada njihova dimenzija, podrazumeva se da ona iznosi 15 *cm*.

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

int brojPlocica(double x, double y, int a=15) {
    double P=x*y;
    cout<<"Povrsina u [cm2] iznosi: "<<P<<endl;
    double p=a*a;
    cout<<"Povrsina jedne plocice u [cm2] iznosi: "<<p<<endl;
    int broj=(int)P/p; // broj potrebnih plocica
    if(broj<(P/p))
        broj++;
    return broj;
}

int main() {

    double x, y, a;
    cout<<"Duzina povrsine koja se poplocava u [cm] --> ";
    cin>>x;
    cout<<"Sirina povrsine koja se poplocava u [cm] --> ";
    cin>>y;

    char odg;
    cout<<"Plocice su oblika kvadrata. "<<endl;
    cout<<"Da li zelite da izaberete dimenzije plocica? [Y/N]... ";
    cin>>odg;
    if(odg=='Y' || odg=='y') {
        cout<<"Navedite duzinu stranice plocice u [cm] --> ";
        cin>>a;
        cout<<"Potreban broj plocica iznosi: "<<brojPlocica(x,y,a);
    }
    if(odg=='N' || odg=='n')
        cout<<"Potreban broj plocica iznosi: "<<brojPlocica(x,y);

    return 0;
}
```

2.4 Preklapanje imena funkcija

- U programskom jeziku C++ postoji mehanizam preklapanja imena funkcija, koji omogućava da se više funkcija nazove istim imenom, ali se one moraju razlikovati po broju i/ili tipovima argumenata i to tako da se obezbedi njihova jednoznačna identifikacija.
- Povratni tipovi funkcija sa istim imenom mogu biti isti.

Zadatak 2.4.1
Analizirati ispis sledećeg programa:

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

bool f(int a, int b) {
    cout<<"Poziv funkcije f() - prva verzija"<<endl;
    if(a>b) return true;
    else return false;
}
bool f(double a, double b) {
    cout<<"Poziv funkcije f() - druga verzija"<<endl;
    if(a>b) return true;
    else return false;
}
bool f(char a, char b) {
    cout<<"Poziv funkcije f() - treca verzija"<<endl;
    if(a>b) return true;
    else return false;
}
int main(){
    cout<<f(5, 3)<<endl;
    cout<<f(7.4, 5.6)<<endl;
    cout<<f('a', 'b')<<endl;
    return 0;
}
```

Pokretanjem programa dobijamo sledeći ispis:

```
Poziv funkcije f() - prva verzija
1
Poziv funkcije f() - druga verzija
1
Poziv funkcije f() - treca verzija
0
```

2.5 Imenski prostori

- Imenski prostori služe za grupisanje globalnih imena u velikim programskim sistemima. Ako se delovi programa stave u različite imenske prostore, tada ne može doći do konflikta sa korišćenjem imena.
- Opšti oblik definisanja imenskog prostora je:

```
namespace Identifikator {/*SADRZAJ IMENSKOG PROSTORA*/}
```

- Identifikatori unutar nekog imenskog prostora mogu da se dohvate iz bilo kog dela programa pomoću operatora `::` (tzv. operatora za razrešenje dosega), odnosno izrazom:

```
Imenski_prostor::identifikator
```

- Takođe, identifikatori iz nekog imenskog prostora mogu da se uvezu naredbom `using`, na sledeći način:

```
using Imenski_prostor::identifikator;
```

ili

```
using namespace Imenski_prostor;
```

Zadatak 2.5.1
Analizirati ispis sledećeg programa:

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

namespace A {
    int x=6;
}
namespace B {
    int x=10;
}
int main(){
    cout<<A::x<<endl;
    cout<<B::x<<endl;
    using namespace A;
    cout<<x<<endl;
    using B::x;
    cout<<x<<endl;
    return 0;
}
```

Pokretanjem programa dobijamo sledeći ispis:

```
6
10
6
10
```

- Po standardu programskog jezika C++ predviđeno je da se globalni identifikatori standardnih zaglavlja nalaze u imenskom prostoru **`std`**.
- Standardom je predviđeno da standardna zaglavlja ne budu u vidu tekstualnih datoteka, pa zbog toga standardna zaglavlja ne sadrže proširenja imena sa **`.h`** (kao što je to slučaj u programskom jeziku C). Jedno od takvih zaglavlja jeste i `<iostream>`. Zbog toga ćemo uvek pisati na početku programa:

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

- Ukoliko ne uključimo imenski prostor **`std`** za ispis promenljive `x` moramo pisati:

```
std::cout<<x;
```

- Ukoliko uključimo imenski prostor **`std`** za ispis promenljive `x` dovoljno je pisati:

```
cout<<x;
```

- Pogledajmo sledeći primer:

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

2.6 Dobijanje izvršne datoteke

- Obrada programa u programskom jeziku C++ obuhvata sledeće faze:
  1.) Unošenje izvornog teksta programa i dobijanje **`.cpp`** datoteka,
  2.) Prevođenje **`.cpp`** datoteka i dobijanje **`.obj`** datoteka,
  3.) Povezivanje prevedenih datoteka u jednu izvršnu datoteku **`.exe`**.
- Pretprocesiranje podrazumeva određene obrade izvornog teksta programa pre početka prevođenja.
- Upoznajmo se sa pojedinačnim pretprocesorskim direktivama:
  - Direktiva `#define` se koristi za zamenu identifikatora sa nizom simbola u programu:

```
#define identifikator niz_simbola
```

- Direktiva `#include` se koristi za umetanje sadržaja datoteke:

```
#include "korisnicko_zaglavlje"
#include <sistemsko_zaglavlje>
```

- Za uslovno prevođenje delova programa koriste se direktive:

```
#ifndef IDENT_DEF /*Ako IDENT_DEF nije definisan*/
#define IDENT_DEF
. . .    /*Definisemo IDENT_DEF*/
#endif /*Kraj uslovnog prevodjenja*/
```

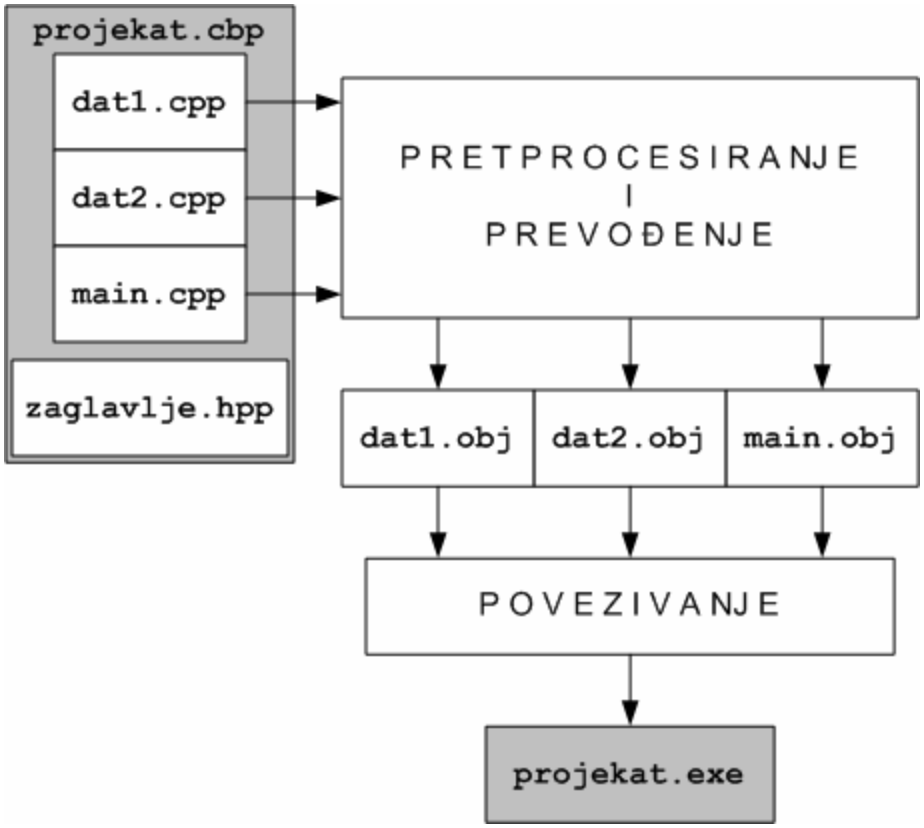
- Glavni program je funkcija `main()`.

Zadatak 2.6.1
Analizirati dobijanje izvršne datoteke na osnovu sledećih datoteka:

```
// Datoteka: zaglavlje.hpp
#ifndef ZAGLAVLJE_DEF
#define ZAGLAVLJE_DEF

#include <iostream>
using namespace std;
#define MAX 100
void f(int, int);
void f(double, double);

#endif



// Datoteka: dat1.cpp
#include "zaglavlje.hpp"
void f(int a, int b) {
    if(a<=MAX && b<=MAX)
        cout<<"Danas je lep dan."<<endl;
    else
        cout<<"Danas nije lep dan."<<endl;
}



// Datoteka: dat2.cpp
#include "zaglavlje.hpp"
void f(double a, double b) {
    if(a<=MAX && b<=MAX)
        cout<<"Danas je lep dan."<<endl;
    else
        cout<<"Danas nije lep dan."<<endl;
}
```

```
// Datoteka: main.cpp
#include "zaglavlje.hpp"
int main() {
    f(3, 2);
    f(4.5, 6.7);
    return 0;
}
```

Pre nego što prevodilac počne sa prevođenjem izvornih datoteka `dat1.cpp`, `dat2.cpp` i `main.cpp` pretprocesor će izvršiti odgovarajuće pripreme teksta, kao što je umetanje odgovarajućeg zaglavlja `zaglavlje.hpp`, a preko njega i standardno zaglavlje `<iostream>`. Zatim će prevodilac prevesti izvorne datoteke, nakon čega nastaju datoteke `dat1.obj`, `dat2.obj` i `main.obj`, iz kojih povezivanjem nastaje izvršna datoteka `.exe`. Ceo proces je prikazan na slici 4.



Slika 4.

3. OSNOVE OBJEKTNOG PROGRAMIRANJA

- Prilikom stvaranja objektno orijentisanog programa softverski inženjeri najviše vremena će posvetiti analizi i modelovanju. Međutim, ovde treba napomenuti činjenicu da se softverski inženjeri bave mislima o učesnicima u informacionom sistemu, odnosno jedinica posmatranja se tretira preko misli o njoj, tačnije preko misli o njenim bitnim karakteristikama. Na primer, projektanti informacionog sistema fakulteta ne barataju sa konkretnim studentima i nastavnicima, već koriste misli o njima, odnosno misli o njihovim bitnim karakteristikama. Na osnovu toga, zaključujemo da su klasa i objekat direktno vezani za misli.
- Misao o bitnim karakteristikama predmeta jeste **pojam** ili **koncept**.
- Reč *predmet* se ovde koristi u najširem smislu i označava predmet posmatranja i/ili razmišljanja. Svaki predmet u najširem smislu poseduje određene karakteristike. Misao o bitnoj karakteristici predmeta naziva se **bitna oznaka**.
- Pojmove možemo klasifikovati na razne načine, ali za naše dalje izlaganje važna je podela na **individualne** i **klasne pojmove**. Individualni pojmovi se odnose na individualne predmete. Individualni predmeti koji imaju zajedničke oznake čine klasu, a misao o datoj klasi jeste klasni pojam. Na primer, individualni pojmovi HAJDN, MOCART i BETOVEN jesu misli o poznatim kompozitorima, kojima je zajedničko to da pripadaju periodu klasicizma i da je njihov stvaralački rad više ili manje vezan za grad Beč. Na osnovu zajedničkih oznaka, oni čine klasu, a misao o toj klasi jeste klasni pojam BEČKI KLASIČAR.
- Sadržaj pojma je skup njegovih bitnih oznaka. Odrediti sadržaj pojma nije nimalo lak posao. Na primer, pojam GRAĐANIN ima bitne oznake *ime i prezime*, *matični broj*, *adresu* i *broj lične karte*, ali isto tako svaki građanin ima *visinu*, *težinu*, *boju kose*, *broj cipela* itd. Upravo ovde se pokazuje opravdanost uvođenja domena problema. Naime, za razliku od logičara, projektant softvera ne mora voditi računa o svim bitnim oznakama pojma, već samo o onima koje su relevantne, tj. one koje su od interesa za dati domen problema koji je predmet analize. Na primer, projektant informacionog sistema poreske uprave će za pojam GRAĐANIN izabrati *ime i prezime*, *matični broj*, *adresu*, *podatke o prihodima* i sl., ali sigurno neće izabrati *visinu* ili *težinu*, iako svaki građanin poseduje pomenute osobine. S druge strane, projektant informacionog sistema zdravstvene ustanove će pored *imena i prezimena*, *matičnog broja*, *adrese* i sl., sigurno izabrati i *visinu* i *težinu*, jer ove osobine su neophodne lekaru prilikom određivanja terapije, ali neće izabrati *podatke o prihodima*.
- Oznake pojma koje su od interesa u datom domenu problema nazivaju se **relevantne oznake pojma**.
- Uvođenjem relevantnih oznaka, umesto bitnih oznaka, rešava se problem donošenja odluke da li je neka oznaka bitna ili ne. Na primer, *matični broj* građanina je oznaka koja je bitna u zemljama u kojima ona postoji, ali postoje zemlje u kojima ona ne postoji pa samim tim nije ni bitna. Dakle, uvođenje relevantnih oznaka koje su vezane za dati domen problema igra ključnu ulogu da naše izlaganje dobije i praktičan karakter, što je naročito važno za softverski inženjering, koji podrazumeva rešavanje praktičnih (inženjerskih) problema. Sada možemo

definisati model, pri čemu treba istaći da sâm termin model, kao homomorfna slika nečeg, ima upotrebu u raznim situacijama (npr. maketa zgrade predstavlja model zgrade koja će biti sagrađena), međutim, softverski inženjer bavi se modelovanjem pojma, tj. modelovanjem misli. Prema tome, ovde ćemo govoriti o softverskom modelovanju, odnosno softverskom modelu.

- Postupak izbora konačnog broja relevantnih oznaka pojma u odnosu na dati domen problema naziva se **softversko modelovanje**, a dobijeni konačni skup relevantnih oznaka naziva se **softverski model**.
- Iz prethodne definicije možemo zapaziti da je modelovanje postupak kojim se dobija uprošćena slika pojma u datom domenu problema, pri čemu taj postupak nije jednoznačno određen, tj. isti pojam se može modelovati na više različitih načina. Na primer, pojam GRAĐANIN se može modelovati konačnim skupom oznaka {*ime i prezime*, *matični broj*}, ali isto tako bi se mogao modelovati konačnim skupom {*ime i prezime*, *matični broj*, *adresa*, *broj pasoša*}. Drugim rečima, projektant je taj koji donosi odluku kako će modelovati neki pojam. Veoma je važno da dobijeni skup relevantnih oznaka bude potrebno i dovoljno deskriptivan, jer to je jedan od ključnih preduslova da softver, kao finalni proizvod, bude kvalitetan. Zbog toga projektant više vremena posvećuje modelovanju, a ne sâmom pisanju kôda.
- **Klasa objekata** (kraće *klasa*) jeste softverski model klasnog pojma.
- **Objekat** je softverski model individualnog pojma.
- Za svaki objekat postoji klasa koja poseduje sve njegove relevantne oznake i tada kažemo da objekat pripada datoj klasi.
- Klasa se realizuje u nekom od objektno-orijentisanih programskih jezika. Klasa se može realizovati na više različitih načina.

3.1 Definicija klase

- Opšti oblik definicije klase u C++ izgleda ovako:

```
class MyClass {
    /* PODACI-CLANOVI */
    /* OBJEKTI-CLANOVI */
    /* FUNKCIJE-CLANICE ili METODE */
};
```

- Ime klase je identifikator. Po konvenciji, ime klase kao i svaka reč koja predstavlja posebnu celinu u okviru imena klase počinje velikim početnim slovom, dok ostala slova su mala. Na primer,

```
MyClass
KompleksniBroj
XYVektor
```

- Podaci-članovi mogu biti bilo kojeg standardnog ili programski definisanog tipa. Na primer:

```
int a;
KorisnickiTip t;
```

- Objekti-članovi se deklarišu navođenjem naziva njihove klase i nazivom imena objekta-člana. Na primer:

```
MyClass m;
```

- Podatke-članove i objekte-članove jednim imenom zovemo **polja**.
- U okviru definicije klase može da se nađe cela definicija metode ili samo njen prototip.
- Sada ćemo napisati klasu `XYPoint` koja modeluje tačku u *xy*-ravni. Klasa će sadržati dva polja tipa `double`:

```
class XYPoint {
    private: /* private-segment */
        /* POLJA */
        double x;
        double y;
    public: /* public-segment */
        /* METODE */
        XYPoint() { x=3; y=4; } // konstruktor
        void setX(double xx) { x=xx; }
        void setY(double yy) { y=yy; }
        double getX() const { return x; }
        double getY() const { return y; }
        double distance() const; // prototip metode
}; /* Tacka-zarez na kraju definicije klase! */
```

- Prava pristupa članu klase mogu biti:
  - `private`
  - `protected`
  - `public`
- Članu klase koji se nalazi u `private`-segmentu može se pristupati samo iz unutrašnjosti klase (npr. iz metoda).
- Članu klase koji se nalazi u `public`-segmentu može se pristupati kako iz unutrašnjosti, tako i iz spoljašnjosti klase (npr. iz metoda, iz programa koji koristi tu klasu itd.).
- Način pristupanja članu klase koji se nalazi u `protected`-segmentu razmotrićemo u poglavlju 6.1.

- Preporučuje se da polja budu u `private` (po B. Mejeru upotreba `public` polja nije objektno-orijentisano programiranje), a metode u `public`-segmentu. Ova preporuka će biti kasnije detaljnije razmotrena na primeru klase `Trougao`.
- NAPOMENA: veoma često studenti zaborave da se definicija klase završava sa znakom `;` (tačka-zarez).
- **Konstruktor** je metoda koja kreira objekat. Osobine konstruktora su sledeće:
  - Konstruktor ima isto ime kao i klasa.
  - Konstruktor može i ne mora da ima parametre.
  - Konstruktor nikad ne vraća vrednost (nema tip i na kraju tela nema naredbu `return`).
- `set`-metoda služi da postavi novu vrednost polja i ima parametar tipa koji odgovara tipu polja kojeg menja.
- `get`-metoda služi da očita vrednost nekog polja i uvek je tipa koji odgovara tipu tog polja.
- Ukoliko metoda menja vrednost bar jednog polja, tada kažemo da ta metoda menja stanje objekta i nazivamo je **modifikatorom**.
- Oznaka `const` označava da metoda ne menja stanje objekta. Na primer:

```
double getX() const { return x; }
```

Zadatak 3.1.1
Napisati klasu `XYPoint` koja modeluje tačku u *xy*-ravni. Napisati test program.

```
// Datoteka: xypoint.hpp
#ifndef XYPOINT_DEF
#define XYPOINT_DEF
#include <math.h>
#include <iostream>
using namespace std;

class XYPoint {
    private:
        double x;
        double y;
    public:
        XYPoint() { x=0; y=0; }
        void setX(double xx) { x=xx; }
        void setY(double yy) { y=yy; }
        double getX() const { return x; }
        double getY() const { return y; }
        double distance() const;
};

#endif
```

Definicija klase `XYPoint` se nalazi između direktiva `#ifndef` i `#endif` da bi se na taj način sprečilo višestruko prevođenje. Metoda `distance()` nije realizovana u **`.hpp`**, već u **`.cpp`** datoteci. Da bi se znalo da je metoda `distance()` članica klase `XYPoint` potrebno je ispred imena metode napisati `XYPoint::` gde je `::` (dva

puta dvotačka) operator za razrešenje dosega. Da bi sve deklarativne naredbe iz **xypoint.hpp** bile dostupne u datoteci **xypoint.cpp** potrebno je napisati:

```
#include "xypoint.hpp"
```

```
// Datoteka: xypoint.cpp
#include "xypoint.hpp"
double XYPoint::distance() const {
    return sqrt(x*x+y*y);
}

// Datoteka: main.cpp
#include "xypoint.hpp"
int main() {
    XYPoint p; // ovde se poziva konstruktor
    cout<<"Tacka ima koordinate: "<<p.getX()<<" i "<<p.getY()<<endl;

    p.setX(3); // ne mozemo pisati p.x=3, jer je x private
    cout<<"Tacka ima koordinate: "<<p.getX()<<" i "<<p.getY()<<endl;

    p.setY(5); // ne mozemo pisati p.y=5, jer je y private
    cout<<"Tacka ima koordinate: "<<p.getX()<<" i "<<p.getY()<<endl;

    return 0;
}
```

Zadatak 3.1.2
Napisati klasu `Trougao`. Napisati test program.

```
// Datoteka: trougao.hpp
#ifndef TROUGAO_DEF
#define TROUGAO_DEF
#include <math.h>
#include <iostream>
using namespace std;

class Trougao {
    private:
    /* POLJA */
        double a;
        double b;
        double c;
    public:
    /* METODE */
        /* Konstruktor */
        Trougao() { a=3; b=4; c=5; }
        /* set-metode */
        void setA(double aa) { a=aa; }
        void setB(double bb) { b=bb; }
        void setC(double cc) { c=cc; }
```

```
        /* get-metode */
        double getA() const { return a; }
        double getB() const { return b; }
        double getC() const { return c; }
        /* ostale metode */
        double getO() const;
        double getP() const;
};
#endif

// Datoteka: trougao.cpp
#include "trougao.hpp"
double Trougao::getO() const {
       return a+b+c;
}
double Trougao::getP() const {
       double s=(a+b+c)/2;
       return sqrt(s*(s-a)*(s-b)*(s-c));
}

// Datoteka: main.cpp
#include "trougao.hpp"
int main() {
    Trougao t;
    cout<<"Duzina stranice a: "<<t.getA()<<endl;
    cout<<"Duzina stranice b: "<<t.getB()<<endl;
    cout<<"Duzina stranice c: "<<t.getC()<<endl;
    cout<<"Obim: "<<t.getO()<<endl;
    cout<<"Povrsina: "<<t.getP()<<endl;
    return 0;
}
```

- Sada ćemo razmotriti zašto se preporučuje da polja budu `private`, a metode `public`. Pretpostavimo da je klasa `Trougao` napisana tako da se polja `a`, `b` i `c` nalaze u `public`-segmentu. Tada se bez ikakvog problema može napisati:

```
Trougao t;
t.a=5;
t.b=1;
t.c=1;
```

- Međutim, trougao sa dužinama stranica 5,1 i 1 ne postoji, jer ne važi 1+1>5. Dakle, stanje objekta klase `Trougao` u kojem polja `a`, `b` i `c` imaju redom vrednosti 5,1 i 1 nije validno. Klasa mora biti napisana tako da se svaki njen objekat nalazi u validnom stanju.
- Iz ovog primera se vidi da je bolje da se ne dozvoli korisniku klase da direktno postavlja nove vrednosti polja, već preko metoda. To se postiže tako što se polja smeste u `private`, a metode u `public`-segment, pri čemu to nije pravilo, već preporuka. Ukoliko se izmena vrednosti polja vrši preko metoda onda se te izmene mogu kontrolisati, kao na primer:

```
    bool setA(double aa) {
        if(aa>0 && aa+b>c && aa+c>b && c+b>aa) {
            a=aa;
            return true;
        } else
            return false;
    }
```

Zadatak 3.1.3

Napisati klasu `Kvadrat`. Napisati test program.

```
// Datoteka: kvadrat.hpp
#ifndef KVADRAT_DEF
#define KVADRAT_DEF
#include <iostream>
using namespace std;
class Kvadrat {
    private:
        double a;
    public:
        Kvadrat() { a=1; }
        void setA(double aa) { a=aa; }
        double getA() const { return a; }
        double getO() const { return 4*a; }
        double getP() const { return a*a; }
};
#endif

// Datoteka: main.cpp
#include "kvadrat.hpp"
int main() {
    Kvadrat k;
    cout<<"Duzina stranice a: "<<k.getA()<<endl;
    cout<<"Obim: "<<k.getO()<<endl;
    cout<<"Povrsina: "<<k.getP()<<endl;
    return 0;
}
```

Zadatak 3.1.4

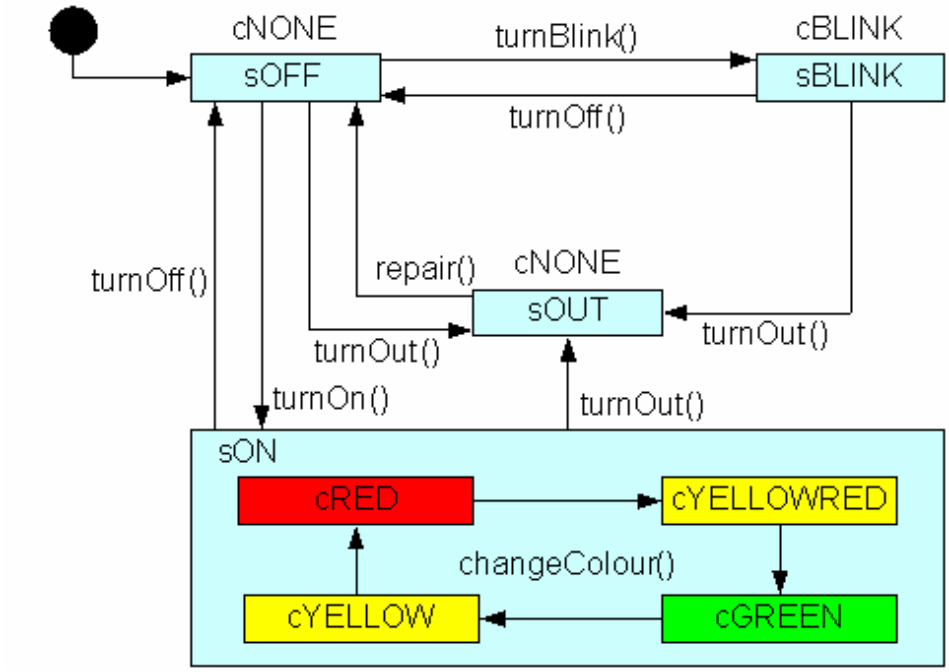
Napisati klasu `Pravougaonik`. Napisati test program.

```
// Datoteka: pravougaonik.hpp
#ifndef PRAVOUGAONIK_DEF
#define PRAVOUGAONIK_DEF
#include <iostream>
using namespace std;
class Pravougaonik {
    private:
        double a, b;
```

```
    public:
        Pravougaonik() { a=1; b=2; }
        void setA(double aa) { a=aa; }
        void setB(double bb) { b=bb; }
        double getA() const { return a; }
        double getB() const { return b; }
        double getO() const { return 2*a+2*b; }
        double getP() const { return a*b; }
};
#endif

// Datoteka: main.cpp
//#include "pravougaonik.hpp"
int main() {
    Pravougaonik p;
    cout<<"Duzina stranice a: "<<p.getA()<<endl;
    cout<<"Duzina stranice b: "<<p.getB()<<endl;
    cout<<"Obim: "<<p.getO()<<endl;
    cout<<"Povrsina: "<<p.getP()<<endl;
    return 0;
}
```

Zadatak 3.1.5

Napisati klasu `Semaphore`. Na slici 5. prikazan je dijagram rada semafora.



Slika 5.

- Semafor može biti u stanju: sOFF, sON, sOUT i sBLINK. Svetlo na semaforu može biti: cNONE, cRED, cYELLOWRED, cGREEN, cYELLOW i cBLINK. Inicijalno stanje semafora je sOFF.
- U zadatku ćemo koristiti nabrojane konstante. Nabrojane konstante su celobrojne konstante (tip `int`) koje se definišu nabrajanjem u naredbama oblika:

```
enum ime_nabrajanja { ime_konstante=vrednost, ... };
```

- Imena konstanti u nabrajanju su identifikatori simboličkih konstanti kojima se dodeljuju vrednosti konstantnih celobrojnih izraza. Ako iza imena neke konstante ne stoji vrednost, dodeliće joj se vrednost koja je za jedan veća od vrednosti prethodne konstante u nizu, odnosno koja je nula ako se radi o prvoj konstanti u nizu.

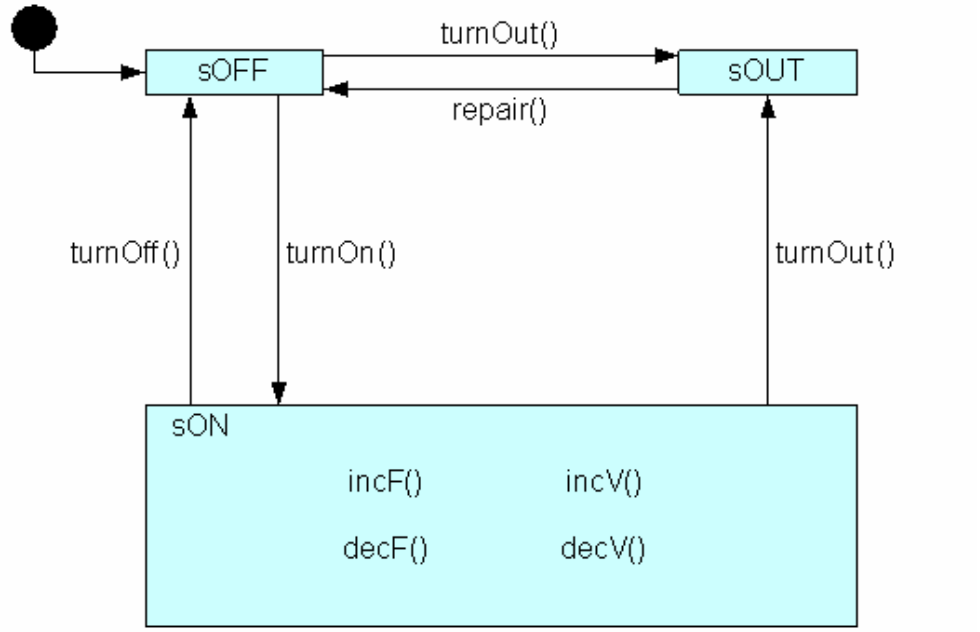
```
// Datoteka: semaphore.hpp
#ifndef SEMAPHORE_DEF
#define SEMAPHORE_DEF

#include <iostream>
using namespace std;

enum States {sON, sOFF, sBLINK, sOUT};
enum Colours {cNONE, cBLINK, cRED, cYELLOWRED, cGREEN, cYELLOW};

class Semaphore {
    private:
        States state;
        Colours colour;
    public:
        Semaphore();
        States getState() const;
        Colours getColour() const;
        bool turnOn();
        bool turnOff();
        bool turnBlink();
        bool turnOut();
        bool repair();
        bool changeColour();
};

#endif

// Datoteka: semaphore.cpp
#include "semaphore.hpp"

Semaphore::Semaphore() {
    state=sOFF;
    colour=cNONE;
}

States Semaphore::getState() const {return state;}
Colours Semaphore::getColour() const {return colour;}
```

```
bool Semaphore::turnOn() {
    if(state==sOFF) {
        state=sON;
        colour=cRED;
        return true;
    }
    else
        return false;
}

bool Semaphore::turnOff() {
    if(state==sON || state==sBLINK) {
        state=sOFF;
        colour=cNONE;
        return true;
    }
    else
        return false;
}

bool Semaphore::turnBlink() {
    if(state==sOFF) {
        state=sBLINK;
        colour=cBLINK;
        return true;
    }
    else
        return false;
}

bool Semaphore::turnOut() {
    if(state!=sOUT) {
        state=sOUT;
        colour=cNONE;
        return true;
    }
    else
        return false;
}

bool Semaphore::repair() {
     if(state==sOUT) {
        state=sOFF;
        colour=cNONE;
        return true;
    }
    else
        return false;
}

bool Semaphore::changeColour() {
    if(state==sON) {
        switch(colour) {
            case cRED : colour=cYELLOWRED; break;
```

```
            case cYELLOWRED : colour=cGREEN; break;
            case cGREEN : colour=cYELLOW; break;
            case cYELLOW : colour=cRED;
        }
        return true;
    }
    else
        return false;
}



// Datoteka: main.cpp
#include "semaphore.hpp"

void printSemaphore(const Semaphore &rs) {
    cout<<"*** Stanje semafora: ";
    switch(rs.getState()) {
        case sON : cout<<"ON"<<endl; break;
        case sOFF : cout<<"OFF"<<endl; break;
        case sOUT : cout<<"OUT"<<endl; break;
        case sBLINK : cout<<"BLINK"<<endl; break;
    }
    cout<<"*** Boja: ";
    switch(rs.getColour()) {
        case cNONE : cout<<"NONE"<<endl; break;
        case cBLINK : cout<<"BLINK"<<endl; break;
        case cRED : cout<<"RED"<<endl; break;
        case cYELLOWRED : cout<<"YELLOW"<<endl; break;
        case cGREEN : cout<<"GREEN"<<endl; break;
        case cYELLOW : cout<<"YELLOW"<<endl; break;
    }
}

char meni() {
    char odg;
    do {
        cout<<"Izaberite opciju: "<<endl;
        cout<<"1. Ukljuci semafor"<<endl;
        cout<<"2. Iskljuci semafor"<<endl;
        cout<<"3. Ukljuci trepcuce zuto"<<endl;
        cout<<"4. Pokvari semafor"<<endl;
        cout<<"5. Popravi semafor"<<endl;
        cout<<"6. Promeni boju"<<endl;
        cout<<"7. Kraj rada"<<endl;
        cin>>odg;
    } while(odg<'1' || odg>'7');
    return odg;
}


int main()
{
    Semaphore s;
    char ch;
```

```
    do {
        ch=meni();
        switch(ch) {
            case '1' :  if(s.turnOn())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printSemaphore(s);
                        break;

            case '2' :  if(s.turnOff())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printSemaphore(s);
                        break;

            case '3' :  if(s.turnBlink())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printSemaphore(s);
                        break;

            case '4' :  if(s.turnOut())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printSemaphore(s);
                        break;
            case '5' :  if(s.repair())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printSemaphore(s);
                        break;

            case '6' :  if(s.changeColour())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printSemaphore(s);
                        break;
        }
    } while(ch!='7');

    return 0;
}
```

Zadatak 3.1.6
Napisati klasu `FMRadio`. Na slici 6. prikazan je dijagram rada.



Slika 6.

- Klasa `FMRadio` sadrži polja: `state`, `frequency` (tipa `double`) i `volume` (tipa `int`). Frekvencija se nalazi u opsegu od 87.5MHz do 108MHz. Promena frekvencije se uvek vrši za korak od 0.5MHz. Jačina zvuka se nalazi u opsegu od 0 do 20. Promena jačine zvuka se uvek vrši za korak 1. Inicijalno stanje je `sOFF`. Frekvencija i jačina zvuka se mogu menjati samo u stanju `sON`. U stanju `sOUT` frekvencija i jačina imaju vrednost -1.

```
// Datoteka: fmradio.hpp
#ifndef FMRADIO_DEF
#define FMRADIO_DEF

#include <iostream>
using namespace std;

#define MINF 87.5
#define MAXF 108.0
#define KF 0.5

#define MINV 0
#define MAXV 20
#define KV 1
```

```
enum States {sON, sOFF, sOUT};

class FMRadio {
    private:
        States state;
        double freq;
        int vol;
    public:
        FMRadio();
        States getState() const;
        double getFreq() const;
        int getVol() const;
        bool turnOn();
        bool turnOff();
        bool turnOut();
        bool repair();
        bool incF();
        bool decF();
        bool incV();
        bool decV();
};
#endif


// Datoteka: fmradio.cpp
#include "fmradio.hpp"

FMRadio::FMRadio() {
    state=sOFF;
    freq=MINF;
    vol=MINV;
}

States FMRadio::getState() const {return state;}
double FMRadio::getFreq() const {return freq;}
int FMRadio::getVol() const {return vol;}

bool FMRadio::turnOn() {
    if(state==sOFF) {
        state=sON;
        return true;
    }
    else
        return false;
}

bool FMRadio::turnOff() {
    if(state==sON) {
        state=sOFF;
        return true;
    }
    else
        return false;
}
```

```
bool FMRadio::turnOut() {
    if(state!=sOUT) {
        state=sOUT;
        freq=-1;
        vol=-1;
        return true;
    }
    else
        return false;
}

bool FMRadio::repair() {
    if(state==sOUT) {
        state=sOFF;
        freq=MINF;
        vol=MINV;
        return true;
    }
    else
        return false;
}

bool FMRadio::incF() {
    if(state==sON && freq+KF<=MAXF) {
        freq+=KF;
        return true;
    }
    else
        return false;
}

bool FMRadio::decF() {
    if(state==sON && freq-KF>=MINF) {
        freq-=KF;
        return true;
    }
    else
        return false;
}

bool FMRadio::incV() {
    if(state==sON && vol+KV<=MAXV) {
        vol+=KV;
        return true;
    }
    else
        return false;
}

bool FMRadio::decV() {
    if(state==sON && vol-KV>=MINV) {
        vol-=KV;
        return true;
    }
```

```
    else
        return false;
}


// Datoteka: main.cpp
#include "fmradio.hpp"

void printFMRadio(const FMRadio &rf) {
    cout<<"*** FMRadio je u stanju: ";
    switch(rf.getState()) {
        case sON : cout<<"ON"<<endl; break;
        case sOFF : cout<<"OFF"<<endl; break;
        case sOUT : cout<<"OUT"<<endl;
    }
    cout<<"*** Frekvencija: "<<rf.getFreq()<<endl;
    cout<<"*** Jacina zvuka: "<<rf.getVol()<<endl;
}

char meni() {
    char odg;
    do {
        cout<<"Izaberite opciju: "<<endl;
        cout<<"1. Ukljuci FMRadio"<<endl;
        cout<<"2. Iskljuci FMRadio"<<endl;
        cout<<"3. Pokvari FMRadio"<<endl;
        cout<<"4. Popravi FMRadio"<<endl;
        cout<<"5. Povecaj frekvenciju"<<endl;
        cout<<"6. Smanji frekvenciju"<<endl;
        cout<<"7. Povecaj jacinu zvuka"<<endl;
        cout<<"8. Smanji jacinu zvuka"<<endl;
        cout<<"9. Kraj rada"<<endl;
        cin>>odg;
    } while(odg<'1' || odg>'9');
    return odg;
}

int main() {
    FMRadio f;
    char ch;
    do {
        ch=meni();
        switch(ch) {
            case '1' :  if(f.turnOn())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printFMRadio(f);
                        break;
            case '2' :  if(f.turnOff())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printFMRadio(f);
                        break;
```

```
            case '3' :  if(f.turnOut())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printFMRadio(f);
                        break;

            case '4' :  if(f.repair())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printFMRadio(f);
                        break;

            case '5' :  if(f.incF())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printFMRadio(f);
                        break;

            case '6' :  if(f.decF())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printFMRadio(f);
                        break;

            case '7' :  if(f.incV())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printFMRadio(f);
                        break;

            case '8' :  if(f.decV())
                            cout<<"Operacija izvrsena!"<<endl;
                        else
                            cout<<"Operacija nije izvrsena!"<<endl;
                        printFMRadio(f);
                        break;
        }
    } while(ch!='9');

    return 0;
}
```

3.2 Konstruktori

- Konstruktor je metoda koja kreira objekat. Konstruktor ima isto ime kao i klasa, može i ne mora da ima parametre i nikad ne vraća vrednost (nema tip i na kraju tela nema naredbu `return`).

- Vrste konstruktora:
  - *Ugrađeni konstruktor* je metoda koja vrši kreiranje objekta, ali ne i njegovu inicijalizaciju. Ugrađeni konstruktor postoji u svakoj klasi, pa ukoliko projektant klase nije napisao svoj konstruktor tada kreiranje objekta vrši ugrađeni konstruktor koji kreira objekat, ali ga ne postavlja u inicijalno stanje (ne inicijalizuje polja).
  - *Konstruktor bez parametara* kreira objekat i postavlja ga u inicijalno stanje koje je unapred odredio projektant klase. Na primer:

    ```
    MyClass() { n=0; }
    ```
  - *Konstruktor sa parametrima* kreira objekat i postavlja ga u inicijalno stanje koje određuje korisnik klase. Na primer:

    ```
    MyClass(int nn) { n=nn; }
    ```
  - *Konstruktor kopije* je metoda koja vrši kreiranje objekta i njegovu inicijalizaciju kopiranjem sadržaja drugog objekta iste klase. Konstruktor kopije ima parametar koji je referenca na objekat iste klase. Na primer:

    ```
    MyClass(const MyClass &rm) { n=rm.n; }
    ```

- Referenca predstavlja alternativno ime originala, a to znači da se u metodi u kojoj postoji prenos objekta po referenci sve vreme radi sa originalom, a koristi njegovo alternativno ime. Zbog toga je važno da se ispred deklaracije parametra metode navede **`const`**, čime se dobija garancija da u okviru metode neće biti promene originala koji se prenosi po referenci u datu metodu. Upravo tako smo uradili i u konstruktoru kopije, tj. zbog toga smo u konstruktoru kopije pisali **`const`** `MyClass &m`.

Zadatak 3.2.1
Analizirati ispis sledećeg programa:

```
// Datoteka: myclass.hpp
#ifndef MYCLASS_DEF
#define MYCLASS_DEF
#include <iostream>
using namespace std;
class MyClass {
    private:
        int x;
    public:
        void setX(int xx) { x=xx; }
        int getX() const { return x; }
};
#endif

// Datoteka: main.cpp
#include "myclass.hpp"
int main() {
    MyClass mc;
    cout<<mc.getX()<<endl;
    return 0;
}
```

Neizvesno je šta će biti ispisano na ekranu, jer u klasi nije napisan konstruktor, pa će ugrađeni konstruktor kreirati objekat `mc` i polje `x` neće biti inicijalizovano.

Zadatak 3.2.2

Napisati klasu `Kocka`. Napisati test program.

```
// Datoteka: kocka.hpp
#ifndef KOCKA_DEF
#define KOCKA_DEF
#include <iostream>
using namespace std;
class Kocka {
    private:
        double a;
    public:
        Kocka();
        Kocka(double);
        Kocka(const Kocka&);
        double getA() const;
        double getP() const;
        double getV() const;
};

// Datoteka: kocka.cpp
#include "kocka.hpp"
Kocka::Kocka() { a=1; cout<<"A"<<endl; }
Kocka::Kocka(double aa) { a=aa; cout<<"B"<<endl; }
Kocka::Kocka(const Kocka &rk) { a=rk.a; cout<<"C"<<endl; }
double Kocka::getA() const { return a; }
double Kocka::getP() const { return 6*a*a; }
double Kocka::getV() const { return a*a*a; }

// Datoteka: main.cpp
#include "kocka.hpp"
int main() {
    Kocka k1, k2(5), k3(k1), k4(k2);
    cout<<"Povrsina kocke k1: "<<k1.getP()<<endl;
    cout<<"Zapremina kocke k1: "<<k1.getV()<<endl;
    cout<<"Povrsina kocke k2: "<<k2.getP()<<endl;
    cout<<"Zapremina kocke k2: "<<k2.getV()<<endl;
    cout<<"Povrsina kocke k3: "<<k3.getP()<<endl;
    cout<<"Zapremina kocke k3: "<<k3.getV()<<endl;
    cout<<"Povrsina kocke k4: "<<k4.getP()<<endl;
    cout<<"Zapremina kocke k4: "<<k4.getV()<<endl;
    return 0;
}
```

Svaki konstruktor ispisuje slovo. Objekat `k1` kreira konstruktor bez parametara, pa će na ekranu biti ispisano slovo `A`. Objekat `k2` kreira konstruktor sa parametrima, pa će na ekranu biti ispisano slovo `B`. Objekte `k3` i `k4` kreira konstruktor kopije, pa će dva puta biti ispisano slovo `C`.

3.3 Destruktor

- Destruktor je metoda koja uništava objekat i ima sledeće karakteristike:
  - Destruktor ima isto ime kao klasa i ispred obavezan znak ~ (tilda).
  - Destruktor nema parametre.
  - Destruktor se ne poziva direktno, već automatski u trenutku kada je potrebno uništiti objekat (na primer, na kraju funkcije je potrebno uništiti sve lokalne objekte).

Zadatak 3.3.1
Analizirati ispis sledećeg programa:

```
// Datoteka: myclass.hpp
#ifndef MYCLASS_DEF
#define MYCLASS_DEF
#include <iostream>
using namespace std;
class MyClass {
    private:
        int x;
    public:
        MyClass() { cout<<"A "; }
        MyClass(int xx) { x=xx; cout<<"B "; }
        MyClass(const MyClass &rm) { x=rm.x; cout<<"C "; }
        void setX(int xx) { x=xx; }
        int getX() const { return x; }
        ~MyClass() { cout<<"D "; }
};
#endif

// Datoteka: main.cpp
#include "myclass.hpp"
int main() {
    MyClass mc1, mc2(3), mc3(mc2);
    return 0;
}
```

Na kraju funkcije `main()` za svaki objekat će biti pozvan destruktor koji će ga uništiti. Pokretanjem programa dobijamo sledeći ispis:

```
A B C D D D
```

- U svakoj klasi postoji ugrađeni konstruktor i ugrađeni destruktor. Ukoliko destruktor nije napisan tada će posao uništavanja objekta obaviti ugrađeni destruktor. Na primer, u ranijim zadacima nismo pisali destruktor, pa je posao uništavanja objekta obavljao ugrađeni destruktor.

- Ukoliko u klasi postoji dinamička alokacija memorije, tada je obavezno da se napiše destruktor koji bi oslobodio zauzetu memoriju i tako sprečio *curenje memorije*.
- Operator **`new`** se koristi za dinamičko zauzimanje memorije i vraća adresu od dobijenog memorijskog segmenta.
- Operator **`delete`** služi za oslobađanje dinamički alociranog memorijskog segmenta.

Zadatak 3.3.2
Analizirati ispis sledećeg programa:

```
// Datoteka: dynclass.hpp
#ifndef DYNCLASS_DEF
#define DYNCLASS_DEF
#include <iostream>
using namespace std;
class DynClass {
    private:
        int *px;
    public:
        DynClass() {
            px=new int; // dinamicka alokacija memorije
            *px=5;
        }
        DynClass(int xx) {
            px=new int; // dinamicka alokacija memorije
            *px=xx;
        }
        DynClass(const DynClass &rd) {
            px=new int; // dinamicka alokacija memorije
            *px=*(rd.px);
        }
        void setX(int xx) { *px=xx; }
        int getX() const { return *px; }
        /* DESTRUKTOR */
        ~ DynClass() {
            delete px;
            cout<<"Memorija oslobodjena!"<<endl;
        }
};
#endif

// Datoteka: main.cpp
#include "dynclass.hpp"
int main() {
    DynClass dc1, dc2(3), dc3(dc2);
    cout<<dc1.getX()<<endl;
    cout<<dc2.getX()<<endl;
    cout<<dc3.getX()<<endl;
    return 0;
}
```

Na kraju funkcije `main()` za svaki objekat će biti pozvan destruktor koji će osloboditi dinamički alociranu memoriju. Pokretanjem programa dobijamo sledeći ispis:

```
5
3
3
Memorija oslobodjena!
Memorija oslobodjena!
Memorija oslobodjena!
```

- U klasi `DynClass` obavezno treba napisati destruktor. Ako u klasi `DynClass` ne bi napisali destruktor tada bi u klasi postojao samo ugrađeni destruktor koji bi uništavao objekat, ali ne bi prethodno oslobodio dinamički zauzetu memoriju koja je vezana za pokazivač `px`, pa bi na taj način dobili program koji *jede* memoriju. Takva pojava je poznata pod imenom *curenje memorije*.

4. MEHANIZAM PREKLAPANJA OPERATORA

- Preklapanje operatora je jedna od tipičnih karakteristika programskog jezika C++. Preklapanje operatora je mehanizam pomoću kojeg ćemo moći da *obučimo* većinu standardnih operatora koje koristimo u programskom jeziku C++ kako da se ponašaju u slučaju da njihovi operandi više nisu standardnih tipova (`int`, `double` i sl.), već klasnih tipova (`MyClass` i sl.).

4.1 Preklapanje metoda

- U poglavlju 2.4 videli smo da u programskom jeziku C++ može da postoji više funkcija koje imaju isto ime.
- Preklapanje metoda je mogućnost da u klasi postoje dve ili više metoda koje mogu imati isto ime.
- Za identifikaciju metode koristi se njeno ime i parametri.

Zadatak 4.1.1
Analizirati ispis sledećeg programa:

```
// Datoteka: myclass.hpp
#ifndef MYCLASS_DEF
#define MYCLASS_DEF
#include <iostream>
using namespace std;
class MyClass {
    private:
        double x;
        double y;
    public:
        void f(int xx, int yy) {
            x=xx;
            y=yy;
            cout<<"Prva verzija"<<endl;
        }
        void f(double xx, double yy) {
            x=xx;
            y=yy;
            cout<<"Druga verzija"<<endl;
        }
};
#endif

// Datoteka: main.cpp
#include "myclass.hpp"
int main() {
    MyClass mc;
    int a=2, b=3;
    double c=2.4, d=4.6;
    mc.f(a, b);
    mc.f(c, d);
    return 0;
}
```

Prvo će biti pozvana prva verzija metode `f(int, int)`, a zatim druga verzija metode `f(double, double)`, pa će ispis na ekran biti ovakav:

```
Prva verzija
Druga verzija
```

4.2 Prijateljske funkcije

- Prijateljska funkcija jeste slobodna funkcija koja je unutar klase proglašena za prijatelja i time ona ima privilegiju da može pristupati svim članovima te klase, pa čak i onim koji se nalaze u `private`-segmentu.
- Neka slobodna funkcija se proglašava prijateljem klase tako što se u definiciji klase navede njen prototip i ispred prototipa reč **`friend`**.
- NAPOMENA: Prijateljska funkcija nije metoda, tj. nije članica klase!

Zadatak 4.2.1
Analizirati sledeći program:

```
// Datoteka: myclass.hpp
#ifndef MYCLASS_DEF
#define MYCLASS_DEF
#include <iostream>
using namespace std;
class MyClass {
    private:
        int x;
    public:
        MyClass() { x=0; }
        int getX() const { return x; }
        friend void fX(const MyClass&); // prijateljska funkcija fX
};
#endif

// Datoteka: main.cpp
#include "myclass.hpp"

void fX(const MyClass &rm) {
/*
Funkcija fX je prijatelj klase MyClass, pa mozemo pisati:
*/
    cout<<rm.x<<endl;
}

void gX(const MyClass &rm) {
/*
Funkcija gX nije prijatelj klase MyClass, pa ne mozemo pisati:
    cout<<rm.x<<endl;
vec mora ovako:
*/
    cout<<rm.getX()<<endl;
}
```

```
int main() {
    MyClass mc;
    fX(mc);
    gX(mc);
    return 0;
}
```

4.3 Preklapanje operatora

- Preklapanje operatora je mogućnost da se za većinu standardnih operatora definiše njihovo ponašanje za slučaj da su operandi klasnih tipova.
- Postoje određena ograničenja prilikom definisanja novih ponašanja operatora:
  - ne može se redefinisati ponašanje operatora za standardne tipove podataka,
  - ne mogu se uvoditi novi simboli za operatore,
  - ne mogu se preklapati operatori:
    - za pristup članu klase `.`
    - za razrešenje dosega `::`
    - za uslovni izraz `?:`
    - za veličinu objekta `sizeof` i
    - za prijavljivanje izuzetka `throw`
  - ne mogu se menjati prioriteti ni smerovi grupisanja pojedinih operatora.
- Uz poštovanje gore navedenih ograničenja, operatori se mogu preklopiti na dva načina:
  - metodom
  - slobodnom funkcijom (prijateljskom funkcijom)
- Preklapanje operatora će biti detaljno objašnjeno na primeru klase `Complex`.

Zadatak 4.3.1

Napisati klasu `Complex` koja modeluje kompleksni broj i sadrži dva polja: `real` i `imag` (tipa `double`):

```
// Datoteka: complex.hpp
#ifndef COMPLEX_DEF
#define COMPLEX_DEF
#include <iostream>
using namespace std;
class Complex {
    private:
        double real;
        double imag;
    public:
        Complex();
        Complex(double, double);
        Complex(const Complex&);
        double getReal() const;
        double getImag() const;
        void setReal(double);
        void setImag(double);
```

```
        Complex& operator=(const Complex&);
        Complex& operator+=(const Complex&);
        Complex& operator-=(const Complex&);
        Complex& operator*=(const Complex&);
        Complex& operator/=(const Complex&);
        const Complex& operator++();
        const Complex operator++(int);
        friend Complex operator+(const Complex&, const Complex&);
        friend Complex operator-(const Complex&, const Complex&);
        friend Complex operator*(const Complex&, const Complex&);
        friend Complex operator/(const Complex&, const Complex&);
        friend bool operator==(const Complex&, const Complex&);
        friend bool operator!=(const Complex&, const Complex&);
        friend ostream& operator<<(ostream&, const Complex&);
        friend istream& operator>>(istream&, Complex&);
};
#endif
```

```
// Datoteka: complex.cpp
#include "complex.hpp"

Complex::Complex() { real=0; imag=0; }
Complex::Complex(double r, double i) { real=r; imag=i; }
Complex::Complex(const Complex &rc) { real=rc.real; imag=rc.imag; }
double Complex::getReal() const { return real; }
double Complex::getImag() const { return imag; }
void Complex::setReal(double r) { real=r; }
void Complex::setImag(double i) { imag=i; }
```

Operator dodele = se preklapa na sledeći način:

```
Complex& Complex::operator=(const Complex &z) {
    real=z.real; imag=z.imag;
    return *this;
}
```

- Operator dodele = se preklapa operatorskom metodom. Operatorska metoda kao argument prihvata po referenci objekat klase `Complex`. Smisao preklapanja operatora dodele je da kad napišemo `z1=z1`, gde su `z1` i `z2` objekti klase `Complex`, zapravo se izvršava metoda `operator=` objekta `z1` za stvarni argument `z2`, tj. kao da je napisano `z1.operator=(z2)`.
- Vidimo da operatorska metoda ima karakterističnо ime po tome što u sebi sadrži reč `operator`, a zatim se navodi operator koji se preklapa.
- U svakom objektu svake klase postoji podatak-član **`this`** koji jeste pokazivač na klasu kojoj taj objekat pripada. Podatak-član **`this`** sadrži adresu objekta kojem pripada.

- Pošto podatak-član **this** jeste pokazivač koji sadrži adresu od objekta kojem pripada, to znači da sa ***this** pristupamo memorijskom prostoru datog objekta. Operatorska metoda kojom je realizovan operator dodele vraća referencu na `*this`. Razlog zašto je tako realizovan operator dodele jeste da bi se kasnije mogao napisati lanac dodela, na primer, `z1=z2=z3=z4` itd. gde su `z1`, `z2`, `z3` i `z4` objekti klase `Complex`. Posmatrajmo izraz `z1=z2=z3=z4`. Redosled izvršavanja će početi sa desna u levo. Prvo će biti izvršena dodela `z3=z4`, a povratna vrednost dodele će biti izmenjena vrednost objekta `z3` (jer metoda vraća `*this`), koji će zatim biti stvarni argument nove dodele `z2=z3` itd. dok najzad ne bude izvršena dodela `z1=z2`.

Operator += se preklapa na sledeći način:

```
Complex& Complex::operator+=(const Complex &z) {
    real+=z.real; imag+=z.imag;
    return *this;
}
```

- Operator `+=` se preklapa operatorskom metodom. Operatorska metoda kao argument prihvata po referenci objekat klase `Complex`. Operatorska metoda kojom je realizovan operator `+=` vraća referencu na `*this`. Razlog zašto je tako realizovan operator `+=` jeste da bi se kasnije moglo napisati `z1+=z2+=z3+=z4`, gde su `z1`, `z2`, `z3` i `z4` objekti klase `Complex`.

Operatori `-=`, `*=` i `/=` se preklapaju na sličan način:

```
Complex& Complex::operator-=(const Complex &z) {
    real-=z.real; imag-=z.imag;
    return *this;
}

Complex& Complex::operator*=(const Complex &z) {
    double r=real*z.real - imag*z.imag;
    double i=real*z.imag + imag*z.real;
    real=r; imag=i;
    return *this;
}

Complex &Complex::operator/=(const Complex &z){
    double r=real;
    double i=imag;
    double d=z.real*z.real+z.imag*z.imag;
    real = (r*z.real + i*z.imag)/d;
    imag = (i*z.real - r*z.imag)/d;
    return *this;
}
```

Prefiksni i postfiksni operator `++` se preklapaju operatorskim metodama:

```
const Complex& Complex::operator++() {
    real++; imag++;
    return *this;
}

const Complex Complex::operator++(int i) {
    Complex w(real, imag);
    real++; imag++;
    return w;
}
```

Operator + se preklapa slobodnom funkcijom, tačnije prijateljskom funkcijom. Unutar klase `Complex` naveden je prototip slobodne operatorske funkcije `operator+` i ispred prototipa stoji reč **`friend`**, a njena realizacija izgleda ovako:

```
Complex operator+(const Complex &z1, const Complex &z2) {
    Complex w(z1.real+z2.real, z1.imag+z2.imag);
    return w;
}
```

- Smisao preklapanja operatora + je da kad napišemo `z1+z2`, gde su `z1` i `z2` objekti klase `Complex`, zapravo se izvršava funkcija `operator+` za stvarne argumente `z1` i `z2`, tj. kao da je napisano `operator+(z1,z2)`.
- Kao što možemo primetiti u telu funkcije se kreira lokalni objekat `w` na osnovu parametara `z1.real+z2.real` i `z1.imag+z2.imag`. Posao kreiranja objekta `w` obavlja konstruktor sa parametrima klase `Complex`.
- Kao povratnu vrednost, funkcija `operator+` vraća po vrednosti (ne po referenci) objekat `w`.
- NAPOMENA: Važno je primetiti da povratni tip funkcije mora biti `Complex`, a ne `Complex&`. Razlog za to je što se ne može vratiti referenca na lokalni objekat koji će biti uništen po završetku izvršavanja funkcije. Dakle, pogrešno je pisati:

```
friend Complex& operator+(const Complex&, const Complex&);
```

Operatori -, * i / se preklapaju na sličan način:

```
Complex operator-(const Complex &z1, const Complex &z2){
    Complex w(z1.real-z2.real, z1.imag-z2.imag);
    return w;
}

Complex operator*(const Complex &z1, const Complex &z2){
    Complex w(z1.real*z2.real - z1.imag*z2.imag,
              z1.imag*z2.real + z1.real*z2.imag);
    return w;
}
```

```
Complex operator/(const Complex &z1, const Complex &z2){
    double d=z2.real*z2.real+z2.imag*z2.imag;
    Complex w((z1.real*z2.real + z1.imag*z2.imag)/d,
              (z1.imag*z2.real - z1.real*z2.imag)/d);
    return w;
}
```

Operatori == i != se preklapaju prijateljskim funkcijama:

```
bool operator==(const Complex &z1, const Complex &z2){
    return (z1.real==z2.real)&&(z1.imag==z2.imag);
}

bool operator!=(const Complex &z1, const Complex &z2){
    return (z1.real!=z2.real)||(z1.imag!=z2.imag);
}
```

Operatori << i >> se preklapaju na ovaj način:

```
ostream &operator<<(ostream &out, const Complex &z){
    if( z.imag == 0)
        out<<z.real;
    if( z.real == 0 && z.imag!=0)
        out<<z.imag<<"i";
    if( z.real != 0 && z.imag>0)
        out<<z.real<<"+"<<z.imag<<"i";
    if( z.real != 0 && z.imag<0)
        out<<z.real<<z.imag<<"i";
    return out;
}

istream &operator>>(istream &in, Complex &z){
    in>>z.real>>z.imag;
    return in;
}
```

Sada ćemo prikazati upotrebu preklopljenih operatora:

```
// Datoteka: main.cpp
#include "complex.hpp"

int main() {
    Complex z1, z2(1,1), z3(2,3), z4(1,0), z5(0,1), z6(0,0);
    cout<<"z1 = "<<z1<<endl;
    cout<<"z2 = "<<z2<<endl;
    cout<<"z3 = "<<z3<<endl;
    cout<<"z4 = "<<z4<<endl;
    cout<<"z5 = "<<z5<<endl;
    cout<<"z6 = "<<z6<<endl;
    z1=z2+z3;
    cout<<"z1 = "<<z1<<endl;
    cout<<"z2 = "<<z2<<endl;
```

```
    cout<<"z3 = "<<z3<<endl;
    cout<<"z4 = "<<z4<<endl;
    cout<<"z5 = "<<z5<<endl;
    cout<<"z6 = "<<z6<<endl;
    z2=z1*z3;
    cout<<"z1 = "<<z1<<endl;
    cout<<"z2 = "<<z2<<endl;
    cout<<"z3 = "<<z3<<endl;
    cout<<"z4 = "<<z4<<endl;
    cout<<"z5 = "<<z5<<endl;
    cout<<"z6 = "<<z6<<endl;
    z2+=z3;
    cout<<"z1 = "<<z1<<endl;
    cout<<"z2 = "<<z2<<endl;
    cout<<"z3 = "<<z3<<endl;
    cout<<"z4 = "<<z4<<endl;
    cout<<"z5 = "<<z5<<endl;
    cout<<"z6 = "<<z6<<endl;
    cout<<"Unesite z1: "<<endl;
    cin>>z1;
    cout<<"Unesite z2: "<<endl;
    cin>>z2;
    cout<<"z1 = "<<z1<<endl;
    cout<<"z2 = "<<z2<<endl;
    z1++;
    ++z2;
    cout<<"z1 = "<<z1<<endl;
    cout<<"z2 = "<<z2<<endl;
    return 0;
}
```

Zadatak 4.3.2

Napisati klasu `Otpornik`.

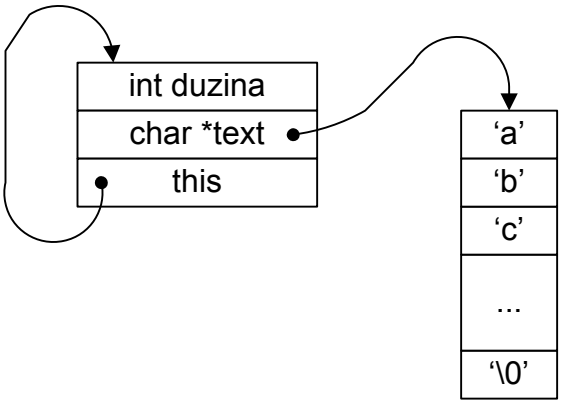
```
// Datoteka: otpornik.hpp
#ifndef OTPORNIK_DEF
#define OTPORNIK_DEF

#include <iostream>
using namespace std;

class Otpornik {
    private:
       double r;
    public:
       Otpornik();
       Otpornik(double);
       Otpornik(const Otpornik&);
       void setR(double);
       double getR() const;
       Otpornik& operator=(const Otpornik&);
       friend Otpornik operator+(const Otpornik&, const Otpornik&);
       friend Otpornik operator||(const Otpornik&, const Otpornik&);
```

```
       friend ostream& operator<<(ostream&, const Otpornik&);
       friend istream& operator>>(istream&, Otpornik&);
};
#endif


// Datoteka otpornik.cpp
#include "otpornik.hpp"

Otpornik::Otpornik() { r=0; }
Otpornik::Otpornik(double rr) { r=rr; }
Otpornik::Otpornik(const Otpornik &x) { r=x.r; }
void Otpornik::setR(double rr) { r=rr; }
double Otpornik::getR() const { return r; }

Otpornik& Otpornik::operator=(const Otpornik &x) {
    r=x.r;
    return *this;
}

Otpornik operator+(const Otpornik &x, const Otpornik &y) {
    Otpornik z(x.r+y.r);
    return z;
}

Otpornik operator||(const Otpornik &x, const Otpornik &y) {
    Otpornik z((x.r*y.r)/(x.r+y.r));
    return z;
}

ostream& operator<<(ostream &out, const Otpornik &x) {
    out<<x.r;
    return out;
}

istream& operator>>(istream &in, Otpornik &x) {
    in>>x.r;
    return in;
}


// Datoteka: main.cpp
#include "otpornik.hpp"

int main() {
    Otpornik r1, r2;
    cout<<"Unesite r1: ";
    cin>>r1;
    cout<<"Unesite r2: ";
    cin>>r2;
    cout<<"Redna veza: "<<(r1+r2)<<endl;
    cout<<"Paralelna veza: "<<(r1||r2)<<endl;
    return 0;
}
```

Zadatak 4.3.3
Napisati klasu `DinString`.

```
// Datoteka: dinstring.hpp
#ifndef DINSTRING_DEF
#define DINSTRING_DEF
#include <iostream>
using namespace std;

class DinString {
  private:
     int duzina;
     char *text;
  public:
     DinString();
     DinString(const char[]);
     DinString(const DinString&);
     ~DinString();
     int length() const;
     char& operator[](int);
     char operator[](int) const;
     DinString& operator=(const DinString&);
     DinString& operator+=(const DinString&);
     friend bool operator==(const DinString&, const DinString&);
     friend bool operator!=(const DinString&, const DinString&);
     friend DinString operator+(const DinString&, const DinString&);
     friend ostream& operator<<(ostream&, const DinString&);
};
#endif
```

- Objekat klase `DinString` sadrži polje `text` koje je pokazivač na tip `char`, polje `duzina` tipa `int` i polje **`this`** koje pokazuje na sam objekat kojem pripada. Na slici 7. prikazan je objekat klase `DinString`. Pokazivač `text` sadrži adresu memorijskog segmenta u kojem se nalaze elementi stringa. Na kraju stringa se nalazi karakter **`NULL`**, tj. `'\0'`. Polje `duzina` sadrži broj karaktera u stringu.



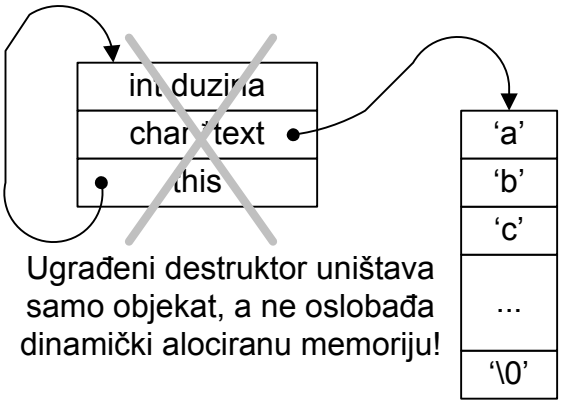
Slika 7.

- Konstruktor bez parametara kreira objekat i postavlja ga u inicijalno stanje:

```
duzina=0; text=NULL;
```

```
// Datoteka: dinstring.cpp
#include "dinstring.hpp"

/* KONSTRUKTORI */
DinString::DinString() {
    duzina=0;
    text=NULL;
}

DinString::DinString(const char ulazniStr[]) {
    duzina=0;
    while(ulazniStr[duzina]!='\0')
        duzina++;
    text=new char[duzina+1];
    for (int i=0; i<duzina; i++)
        text[i]=ulazniStr[i];
    text[duzina]='\0';
}

DinString::DinString(const DinString &ds) {
    duzina=ds.duzina;
    text=new char[duzina+1];
    for (int i=0; i<duzina; i++)
        text[i]=ds.text[i];
    text[duzina]='\0';
}

/* DESTRUKTOR */
DinString::~DinString() {
    delete[] text;
}
```

- U klasi `DinString` obavezno treba napisati destruktor, da bi se sprečilo *curenje memorije*. Slika 8. ilustruje efekat uništavanja objekta klase `DinString` pomoću ugrađenog destruktora, gde vidimo da će biti uništen samo objekat, pri čemu neće doći do oslobađanja dinamički alocirane memorije u kojoj se nalaze karakteri i koja je vezana za polje `text` (koji je pokazivač na tip `char` i koji se nalazi u objektu).



Slika 8.

```
/* DODATNA METODA */
int DinString::length() const {
    return duzina;
}

/* PREKLOPLJENI OPERATORI */
char& DinString::operator[](int i) {
    return text[i];
}

char DinString::operator[](int i) const {
    return text[i];
}
```

- Operator za indeksiranje `[]` u klasi `DinString` se preklapa dva puta, jer pokriva dva slučaja – čitanje sa *i*-te pozicije i pisanje na *i*-tu poziciju u stringu.

```
DinString& DinString::operator=(const DinString &ds) {
    if (this!=&ds){
        delete[] text;
        duzina=ds.duzina;
        text=new char[duzina+1];
        for (int i=0; i<duzina; i++)
            text[i]=ds.text[i];
        text[duzina]='\0';
    }
    return *this;
}
```

- Kod praklapanja operatora dodele = važno je proveriti da li važi `this!=&ds`, da bi se izbeglo da kod izraza `a=a`, gde je `a` objekat klase `DinString`, bude uništen sadržaj objekta `a`. Ukoliko važi `this!=&ds` (a to je primer za slučaj `a=b`) tada se može uništiti string koji je vezan za pokazivač `text`, a zatim prepisati dužina od objekta koji se dodeljuje, zauzeti nova memorija i najzad, prepisati novi elementi.

```
DinString& DinString::operator+=(const DinString &ds) {
    int i;
    char *tempText=new char[duzina+ds.duzina+1];
    for (i=0; i<duzina; i++)
        tempText[i]=text[i];
    for (i=0; i<ds.duzina; i++)
        tempText[duzina+i]=ds.text[i];
    tempText[duzina+ds.duzina]='\0';
       duzina+=ds.duzina;
       delete []text;
       text=tempText;
       return *this;
}
```

- Preklapanje operatora za spajanje (konkatenaciju) stringova += se realizuje tako što se formira lokalni string `tempText` koji sadrži elemente oba stringa i na kraju karakter `NULL`, tj. `'\0'`. Zatim se postavlja nova vrednost dužine stringa, oslobađa memorija koja je vezana za pokazivač `text` i pokazivaču `text` se dodeljuje vrednost iz pokazivača `tempText`, odnosno adresa koja se nalazi u pokazivaču `tempText`. Na taj način se uspostavlja veza pokazivača `text` sa memorijskim segmentom na koji pokazuje `tempText`, tj. pokazivač `text` će pokazivati na spojeni niz, što je i bio cilj.

```
bool operator==(const DinString &ds1, const DinString &ds2) {
    if (ds1.duzina!=ds2.duzina)
        return false;
    for (int i=0; i<ds1.duzina; i++)
        if (ds1.text[i]!=ds2.text[i])
            return false;
    return true;
}

bool operator!=(const DinString &ds1, const DinString &ds2) {
    if (ds1.duzina!=ds2.duzina)
        return true;
    for (int i=0; i<ds1.duzina; i++)
        if (ds1.text[i]!=ds2.text[i])
            return true;
    return false;
}
```

- Preklapanje operatora za proveru jednakost stringova == se realizuje tako što se prvo proverava da li je dužina jednaka, a zatim da li su jednaki elementi stringova na odgovarajućim pozicijama.

```
DinString operator+(const DinString &ds1, const DinString &ds2) {
    DinString temp;
    temp.duzina=ds1.duzina+ds2.duzina;
    temp.text=new char[temp.duzina+1];
    int i;
    for(i=0; i<ds1.duzina; i++)
        temp.text[i]=ds1.text[i];
    for(int j=0; j<ds2.duzina; j++)
        temp.text[i+j]=ds2.text[j];
    temp.text[temp.duzina]='\0';
    return temp;
}

ostream& operator<<(ostream &out, const DinString &ds) {
    if(ds.duzina>0)
        out<<ds.text;
    return out;
}
```

- Za ispis možemo pisati `out<<s.text;` jer na kraju se nalazi karakter `'\0'`.

```
// Datoteka: main.cpp
#include "dinstring.hpp"

int main(){
       DinString a, b("Dobar"), c("dan");
       cout<<"a: "<<a<<endl;
       cout<<"b: "<<b<<endl;
       cout<<"c: "<<c<<endl;
       cout<<endl<<"Testiranje +="<<endl;
       cout<<"a+=b: "<<(a+=b)<<endl;
       cout<<"a+=c: "<<(a+=c)<<endl;
       cout<<endl<<"Testiranje +"<<endl;
       cout<<"a=b+c: "<<(a=b+c)<<endl;
       cout<<"a=b+\" \"+c: "<<(a=b+" "+c)<<endl;
       cout<<endl<<"Testiranje []"<<endl;
       char x=a[5];
       cout<<"x=a[5] x: "<<x<<endl;
       a[5]='Z';
       cout<<"Izvrseno a[5]='Z', sada je a: "<<a<<endl;
       cout<<endl<<"Testiranje ==, = i !="<<endl;
       cout<<"a: "<<a<<endl;
       cout<<"b: "<<b<<endl;
       cout<<"a==b? "<<(a==b)<<endl;
       cout<<"a!=b? "<<(a!=b)<<endl;
       a=b;
```

```
       cout<<"a: "<<a<<endl;
       cout<<"b: "<<b<<endl;
       cout<<"a==b? "<<(a==b)<<endl;
       cout<<"a!=b? "<<(a!=b)<<endl;
       cout<<endl<<"Ocitavanje duzine stringa pomocu length"<<endl;
       cout<<"a: "<<a<<endl;
       cout<<"duzina stringa a: "<<a.length()<<endl;
       a=(a+=(" "+b))+" "+c;
       cout<<"a: "<<a<<endl;
       cout<<"duzina stringa a: "<<a.length()<<endl;
       return 0;
}
```

Zadatak 4.3.4

Napisati klasu `IntArray`.

```
// Datoteka: intarray.hpp
#ifndef INTARRAY_DEF
#define INTARRAY_DEF
#include <iostream>
using namespace std;

class IntArray {
   private:
        int *el;
        int brEl;
   public:
        IntArray() { el=NULL; brEl=0; }
        IntArray(const int[], int);
        IntArray(const IntArray&);
        ~IntArray() { delete[] el; }
        int length() const { return brEl; }
        int& operator[](int);
        int operator[](int) const;
        IntArray& operator=(const IntArray&);
        IntArray& operator+=(const IntArray&);
        friend bool operator==(const IntArray&, const IntArray&);
        friend bool operator!=(const IntArray&, const IntArray&);
        friend IntArray operator+(const IntArray&, const IntArray&);
        friend ostream& operator<<(ostream&, const IntArray&);
};
#endif


// Datoteka: intarray.cpp
#include "intarray.hpp"
IntArray::IntArray(const int ulazniNiz[], int n) {
    brEl=n;
    el=new int[brEl];
    for (int i=0; i<brEl; i++)
        el[i]=ulazniNiz[i];
}
```

```
IntArray::IntArray(const IntArray &ra) {
    brEl=ra.brEl;
    el=new int[brEl];
    for (int i=0; i<brEl; i++)
        el[i]=ra.el[i];
}

int& IntArray::operator[](int i) { return el[i]; }

int IntArray::operator[](int i) const { return el[i]; }

IntArray& IntArray:: operator=(const IntArray &ra) {
    if (this!=&ra){
        delete[] el;
        brEl=ra.brEl;
        el=new int[brEl];
        for (int i=0; i<brEl; i++)
            el[i]=ra.el[i];
    }
    return *this;
}

IntArray& IntArray::operator+=(const IntArray &ra) {
    int i;
    int *tempEl=new int[brEl+ra.brEl];
    for (i=0;i<brEl;i++)
        tempEl[i]=el[i];
    for (i=0;i<ra.brEl;i++)
        tempEl[brEl+i]=ra.el[i];
    brEl+=ra.brEl;
    delete []el;
    el=tempEl;
    return *this;
}

bool operator==(const IntArray &ra1, const IntArray &ra2) {
    if (ra1.brEl!=ra2.brEl)
        return false;
    for (int i=0;i<ra1.brEl;i++)
        if (ra1.el[i]!=ra2.el[i])
            return false;
    return true;
}

bool operator!=(const IntArray &ra1, const IntArray &ra2) {
    if (ra1.brEl!=ra2.brEl)
        return true;
    for (int i=0;i<ra1.brEl;i++)
        if (ra1.el[i]!=ra2.el[i])
            return true;
    return false;
}
```

```
IntArray operator+(const IntArray &ra1, const IntArray &ra2) {
    IntArray temp;
    temp.brEl=ra1.brEl+ra2.brEl;
    temp.el=new int[temp.brEl];
    int i;
    for(i=0;i<ra1.brEl;i++)
        temp.el[i]=ra1.el[i];
    for(int j=0;j<ra2.brEl;j++)
        temp.el[i+j]=ra2.el[j];
    return temp;
}

ostream& operator<<(ostream &out, const IntArray &ra) {
       int i;
       for(i=0; i<ra.brEl; i++)
        out<<ra.el[i]<<" ";
       return out;
}


// Datoteka: main.cpp
#include "intarray.hpp"

int main(){
    const int a1[] ={1,2,3,4,5};
    const int a2[] ={6,7,8};
    const int a3[] ={0,1,2,3,4,5,6,7,8,9};
    const int a4[] ={100,101,102,103};
    IntArray ia0, ia1(a1,5), ia2(a2,3), ia3(a3,10), ia4(a4,4);
    cout<<"ia0: "<<ia0<<endl;
    cout<<"ia1: "<<ia1<<endl;
    cout<<"ia2: "<<ia2<<endl;
    cout<<"ia3: "<<ia3<<endl;
    cout<<"ia4: "<<ia4<<endl;
    IntArray ia5(ia0), ia6(ia1), ia7(ia2), ia8(ia3), ia9(ia4);
    cout<<"ia5: "<<ia5<<endl;
    cout<<"ia6: "<<ia6<<endl;
    cout<<"ia7: "<<ia7<<endl;
    cout<<"ia8: "<<ia8<<endl;
    cout<<"ia9: "<<ia9<<endl;
    cout<<endl<<"Testiranje +="<<endl;
    ia1+=ia2;
    ia3+=ia4;
    cout<<"ia1: "<<ia1<<endl;
    cout<<"ia3: "<<ia3<<endl;
    cout<<endl<<"Testiranje +"<<endl;
    ia0=ia5+ia6;
    cout<<"ia0: "<<ia0<<endl;
    cout<<endl<<"Testiranje []"<<endl;
    int x=ia1[5];
    cout<<"x=ia1[5] x: "<<x<<endl;
    cout<<endl<<"Testiranje ==, = i !="<<endl;
    cout<<"ia4: "<<ia4<<endl;
    cout<<"ia9: "<<ia9<<endl;
```

```
    cout<<"ia4==ia9? "<<(ia4==ia9)<<endl;
    cout<<"ia4!=ia9? "<<(ia4!=ia9)<<endl;
    return 0;
}
```

Zadatak 4.3.5

Napisati klasu `Polinom`, koja modeluje polinom reda `n`:

$$P(x)=a[0]x^0+a[1]x^1+...+a[n]x^n$$

```
// Datoteka: polinom.hpp
#ifndef POLINOM_DEF
#define POLINOM_DEF
#include <iostream>
using namespace std;

class Polinom {
    private:
        double *a;
        int n;
    public:
        Polinom() { a=NULL; n=-1; }
        Polinom(const double[], int);
        Polinom(const Polinom&);
        ~Polinom() { delete[] a; }
        int getN() const { return n; }
        double& operator[](int);
        double operator[](int) const;
        double operator()(double) const;
        Polinom& operator=(const Polinom&);
        Polinom& operator+=(const Polinom&);
        Polinom& operator-=(const Polinom&);
        friend bool operator==(const Polinom&, const Polinom&);
        friend bool operator!=(const Polinom&, const Polinom&);
        friend Polinom operator-(const Polinom&);
        friend Polinom operator+(const Polinom&, const Polinom&);
        friend Polinom operator-(const Polinom&, const Polinom&);
        friend ostream& operator<<(ostream&, const Polinom&);
};
#endif


// Datoteka: polinom.cpp
#include "polinom.hpp"

Polinom::Polinom(const double niz[], int red) {
    n=red;
    a=new double[n+1];
    for (int i=0; i<=n; i++)
        a[i]=niz[i];
}
```

```
Polinom::Polinom(const Polinom &p) {
    n=p.n;
    a=new double[n+1];
    for (int i=0; i<=n; i++)
        a[i]=p.a[i];
}

double& Polinom::operator[](int i) {
    return a[i];
}

double Polinom::operator[](int i) const {
    return a[i];
}

double Polinom::operator()(double x) const {
    double s=0;
    if(n>=0) {
        s=a[n];
        for(int i=n-1; i>=0; i--)
            s=s*x+a[i];
    }
    return s;
}

Polinom& Polinom::operator=(const Polinom &p) {
    if(this!=&p) {
        delete[] a;
        n=p.n;
        a=new double[n+1];
        for (int i=0; i<=n; i++)
            a[i]=p.a[i];
    }
    return *this;
}

Polinom& Polinom::operator+=(const Polinom &p) {
    *this=*this+p;
    return *this;
}

Polinom& Polinom::operator-=(const Polinom &p) {
    *this=*this-p;
    return *this;
}

bool operator==(const Polinom &p, const Polinom &q) {
    if(p.n!=q.n)
        return false;
    for(int i=0; i<=p.n; i++)
        if(p.a[i]!=q.a[i])
            return false;
    return true;
}
```

```
bool operator!=(const Polinom &p, const Polinom &q) {
    if(p.n!=q.n)
        return true;
    for(int i=0; i<=p.n; i++)
        if(p.a[i]!=q.a[i])
            return true;
    return false;
}

Polinom operator-(const Polinom &p) {
    Polinom r;
    if(p.n>0) {
        r.n=p.n;
        r.a=new double[r.n+1];
        for(int i=0; i<=p.n; i++)
            r.a[i]=-p.a[i];
    }
    return r;
}

Polinom operator+(const Polinom &p, const Polinom &q) {
    int i;
    Polinom r;
    if(p.n<=q.n) {
        r=q;
        for(i=0; i<=p.n; i++)
            r[i]+=p[i];
    } else {
        r=p;
        for(i=0; i<=q.n; i++)
            r[i]+=q[i];
    }
    while(r.a[r.n]==0)
        r.n--;
    return r;
}

Polinom operator-(const Polinom &p, const Polinom &q) {
    Polinom r;
    r=p+(-q);
    return r;
}

ostream& operator<<(ostream &out, const Polinom &p) {
    out<<"[";
    if(p.n>=0) {
        for(int i=0; i<p.n; i++)
            out<<" "<<p.a[i]<<",";
        out<<" "<<p.a[p.n]<<" ]";
    } else
        out<<" ]";
    return out;
}
```

```
// Datoteka: main.cpp
#include "polinom.hpp"

int main() {
    const double d1[] = {0.1, 0.2, 0.3, 0.4};
    const double d2[] = {0.5, 0.6, 0.7, 0.8, 0.9};
    const double d3[] = {4, 3, 2, 1, 0, -1};
    Polinom p0, p1(d1,3), p2(d2,4), p3(d3,5);
    cout<<p0<<endl;
    cout<<p1<<endl;
    cout<<p2<<endl;
    cout<<p3<<endl;
    cout<<p0(1)<<endl;
    cout<<p1(0.7)<<endl;
    cout<<p2(3)<<endl;
    cout<<p3(2)<<endl;
    p0=p1+p2;
    cout<<p0<<endl;
    p0=p1-p2;
    cout<<p0<<endl;
    p0=p2-p1;
    cout<<p0<<endl;
    p1=p0;
    cout<<(p1==p0)<<endl;
    cout<<(p2==p0)<<endl;
    p1+=p3;
    cout<<p1<<endl;
    p1-=p3;
    cout<<p1<<endl;
    return 0;
}
```

5. KLIJENTSKE VEZE

- Razni pojmovi stoje u nekom odnosu (npr. trougao sadrži tri temena, automobil sadrži motor itd.). Odnos među pojmovima modelujemo vezama između klasa.
- Veze između klasa mogu se klasifikovati u tri grupe:
  1. **Klijentske veze** u kojima klase-klijenti koriste usluge klasa-snabdevača, a tu spadaju:
     - *asocijacija* koja obuhvata širok spektar logičkih relacija između semantički nezavisnih klasa,
     - *agregacija* koja modeluje odnos celina-deo,
     - *kompozicija* koja takođe modeluje odnos celina-deo, ali uz uslov egzistencijalne zavisnosti dela od celine,
     - *veze korišćenja.*
  2. **Nasleđivanje** koje podrazumeva kreiranje potklase na osnovu preuzimanja sadržaja natklase, uz mogućnost modifikacije preuzetih i dodavanja novih sadržaja.
  3. **Veze zavisnosti**.
- Ovo poglavlje detaljnije razmatra kompoziciju, a naredno nasleđivanje.

5.1 Kompozicija

- Kompozija modeluje odnos *poseduje*. Na primer, pojam AUTOMOBIL *poseduje* pojam MOTOR (pri čemu se u ovom primeru podrazumeva da je automobil isključivo vozilo koje pokreće motor i ne ulazi se u dublje rasprave da li postoje alternativna rešenja). Dakle, pojam AUTOMOBIL predstavlja **celinu** koja sadrži **deo** MOTOR.
- U fazi realizacije klasa koja predstavlja celinu naziva se **vlasnik**, a klasa koja odgovara delu naziva se **komponenta**. Isti termini se koriste i za pojedinačne objekte.
- Kompozicija jeste takva veza klasa, za koju važi to da vlasnik *poseduje* komponentu, pri čemu komponenta ne može postojati pre kreiranja i posle uništenja vlasnika. Drugim rečima, životni vek komponente sadržan je u životnom veku vlasnika.
- Objekat-član je komponenta koja ima jednog vlasnika.
- Ako svaki objekat klase `A` poseduje bar jedan objekat klase `B`, pri čemu stvaranje i uništavanje datog objekta klase `B` zavisi od stvaranja i uništavanja objekta klase `A`, onda se kaže da između klasa `A` i `B` postoji veza kompozicije.

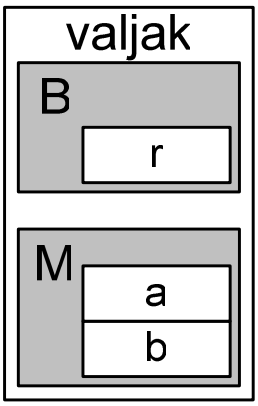
Zadatak 5.1.1

Napisati klase `Krug`, `Pravougaonik` i `Valjak`.

```
// Datoteka: krug.hpp
#ifndef KRUG_DEF
#define KRUG_DEF
#include <math.h>

class Krug {
    private:
        double r;
    public:
        Krug(double rr=1) { r=rr; }
        double getR() const { return r; }
        double getO() const { return 2*r*M_PI; }
        double getP() const { return r*r*M_PI; }
};
#endif

// Datoteka: pravougaonik.hpp
#ifndef PRAVOUGAONIK_DEF
#define PRAVOUGAONIK_DEF

class Pravougaonik {
    private:
        double a;
        double b;
    public:
        Pravougaonik(double aa=1, double bb=2) { a=aa; b=bb; }
        double getA() const { return a; }
        double getB() const { return b; }
        double getO() const { return 2*a+2*b; }
        double getP() const { return a*b; }
};
#endif

// Datoteka: valjak.hpp
#ifndef VALJAK_DEF
#define VALJAK_DEF
#include "krug.hpp"
#include "pravougaonik.hpp"

class Valjak {
    private:
        Krug B;
        Pravougaonik M;
    public:
        Valjak(double rr=1, double hh=1):B(rr), M(2*rr*M_PI, hh){}
        double getR() const { return B.getR(); }
        double getH() const { return M.getB(); }
        double getP() const { return 2*B.getP()+M.getP(); }
        double getV() const { return B.getP()*getH(); }
};
#endif
```

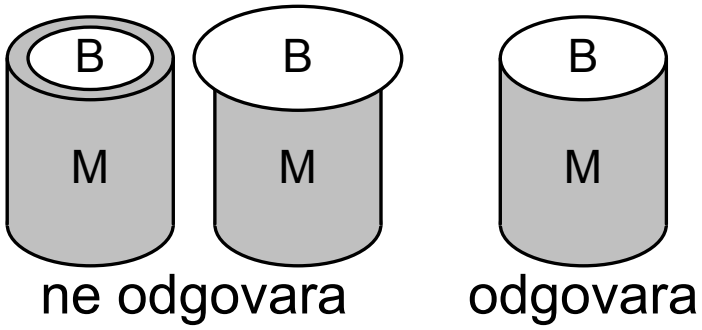
```
// Datoteka: main.cpp
#include "valjak.hpp"
#include <iostream>
using namespace std;

int main() {
    Krug k(3);
    Pravougaonik p(5,8);
    Valjak v(2,4);
    cout<<"Obim kruga: "<<k.getO()<<endl;
    cout<<"Povrsina kruga: "<<k.getP()<<endl;
    cout<<"Obim pravougaonika: "<<p.getO()<<endl;
    cout<<"Povrsina pravougaonika: "<<p.getP()<<endl;
    cout<<"Povrsina valjka: "<<v.getP()<<endl;
    cout<<"Zapremina valjka: "<<v.getV()<<endl;
    return 0;
}
```

- Za računanje površine kruga koristili smo konstantu **`M_PI`** koja se nalazi u standardnom zaglavlju `<math.h>`.
- Objekat klase `Valjak` je objekat-vlasnik koji sadrži dve komponente: `B` je objekat klase `Krug`, a `M` je objekat klase `Pravougaonik`. Kao što vidimo, objekat klase `Valjak` ima složenu strukturu što je ilustrovano na slici 9.



Slika 9.

- Zadatak konstruktora klase `Valjak` je da postavi objekat u takvo inicijalno stanje da osnova odgovara omotaču valjka, jer samo u tom slučaju možemo reći da je objekat u dozvoljenom stanju. Slika 10. ilustruje situacije kada osnova neodgovara, odnosno kada odgovara omotaču valjka.



Slika 10.

- U klasi `Valjak` koristili smo **konstruktor inicijalizator** koji ima posebnu sintaksnu formu, gde posle znaka `:` sledi segment za inicijalizaciju:

```
Valjak(double rr=1, double hh=1) : B(rr), M(2*rr*M_PI, hh) {}
```

- Valjak je određen parametrima `rr` (poluprečnik osnove) i `hh` (visina). Da bi objekat klase `Valjak` došao u dozvoljeno inicijalno stanje, potrebno je da konstruktor klase `Krug` postavi bazu `B` u stanje u kojem polje `r` ima vrednost `rr`, a konstruktor klase `Pravougaonik` postavi omotač `M` u stanje u kojem jedna stranica ima vrednost `2*rr*M_PI`, a druga `hh`.

Zadatak 5.1.2
Napisati klase `JSTrougao` (jednakostranični trougao), `Pravougaonik` i `PP3Prizma` (prava pravilna trostrana prizmu).

```
// Datoteka: jstrougao.hpp
#ifndef JSTROUGAO_DEF
#define JSTROUGAO_DEF
#include <math.h>
class JSTrougao {
    private:
        double a;
    public:
        JSTrougao(double aa=1) { a=aa; }
        double getA() const { return a; }
        double getO() const { return 3*a; }
        double getP() const { return (a*a*sqrt(3))/4; }
};
#endif

// Datoteka: pravougaonik.hpp
#ifndef PRAVOUGAONIK_DEF
#define PRAVOUGAONIK_DEF
class Pravougaonik {
    private:
        double a;
        double b;
```

```
    public:
        Pravougaonik(double aa=1, double bb=2) { a=aa; b=bb; }
        double getA() const { return a; }
        double getB() const { return b; }
        double getO() const { return 2*a+2*b; }
        double getP() const { return a*b; }
};
#endif
```

```
// Datoteka: pp3prizma.hpp
#ifndef PP3PRIZMA_DEF
#define PP3PRIZMA_DEF
#include "jstrougao.hpp"
#include "pravougaonik.hpp"

class PP3Prizma {
    private:
        JSTrougao B;
        Pravougaonik M;
    public:
        PP3Prizma(double aa=1, double hh=1):B(aa), M(3*aa, hh){}
        double getA() const { return B.getA(); }
        double getH() const { return M.getB(); }
        double getP() const { return 2*B.getP()+M.getP(); }
        double getV() const { return B.getP()*getH(); }
};
#endif
```

```
// Datoteka: main.cpp
#include "pp3prizma.hpp"
#include <iostream>
using namespace std;

int main() {
    JSTrougao jst(3);
    Pravougaonik p(5,8);
    PP3Prizma pp3p(2,4);
    cout<<"Obim JS-trougla: "<<jst.getO()<<endl;
    cout<<"Povrsina JS-trougla: "<<jst.getP()<<endl;
    cout<<"Obim pravougaonika: "<<p.getO()<<endl;
    cout<<"Povrsina pravougaonika: "<<p.getP()<<endl;
    cout<<"Povrsina PP3-prizme: "<<pp3p.getP()<<endl;
    cout<<"Zapremina PP3-prizme: "<<pp3p.getV()<<endl;
    return 0;
}
```

Zadatak 5.1.3

Napisati klasu `Osoba`. Koristiti klasu `DinString` iz zadatka 4.3.3. Analizirati redosled poziva konstruktora i destruktora.

```
// Datoteka: osoba.hpp
#ifndef OSOBA_DEF
#define OSOBA_DEF
#include "dinstring.hpp"

#include <iostream>
using namespace std;

class Osoba {
    private:
        DinString ime, prezime;
    public:
        Osoba(const char *s1="", const char *s2="") :
        ime(s1), prezime(s2) {
            cout<<"Osoba: Konstruktor 1."<<endl;
        }
        Osoba(const DinString &ds1, const DinString &ds2) :
        ime(ds1), prezime(ds2) {
            cout<<"Osoba: Konstruktor 2."<<endl;
        }
        Osoba(const Osoba &ro) : ime(ro.ime), prezime(ro.prezime) {
            cout<<"Osoba: Konstruktor 3."<<endl;
        }
        ~Osoba() {
            cout<<"Osoba: Destruktor."<<endl;
        }
        void predstaviSe() const {
            cout<<"Zovem se "<<ime<<" "<<prezime<<"."<<endl;
        }
};

#endif


// Datoteka: main.cpp
#include "osoba.hpp"

int main() {
    const char *s1 = "Petar";
    const char *s2 = "Petrovic";
    const char *s3 = "Jovan";
    const char *s4 = "Jovanovic";
    cout<<"*** Kreiranje objekata ds1, ds2, ds3 i ds4"<<endl;
    DinString ds1(s1), ds2(s2), ds3(s3), ds4(s4);
    cout<<"*** Kreiranje objekta os1"<<endl;
    Osoba os1(s1,s2);
    cout<<"*** Kreiranje objekta os2"<<endl;
    Osoba os2(ds3,ds4);
    cout<<"*** Kreiranje objekta os3"<<endl;
    Osoba os3(os2);
```

```
    cout<<"*** Predstavljanje objekata os1, os2 i os3"<<endl;
    os1.predstaviSe();
    os2.predstaviSe();
    os3.predstaviSe();
    return 0;
}
```

Pokretanjem programa dobijamo sledeći ispis:

```
*** Kreiranje objekata ds1, ds2, ds3 i ds4
DinString: Konstruktor 2.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
DinString: Konstruktor 2.

*** Kreiranje objekta os1
DinString: Konstruktor 2.
DinString: Konstruktor 2.
Osoba: Konstruktor 1.

*** Kreiranje objekta os2
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 2.

*** Kreiranje objekta os3
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.

*** Predstavljanje objekata os1, os2 i os3
Zovem se Petar Petrovic.
Zovem se Jovan Jovanovic.
Zovem se Jovan Jovanovic.

*** Unistavanje objekata
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
```

6. NASLEĐIVANJE

- Nasleđivanje modeluje odnos *izvodi se iz*. Na primer, pojam STUDENT *izvodi se iz* pojma OSOBA, tačnije STUDENT je OSOBA koja ima još neke specifične osobine, kao što je, na primer, posedovanje indeksa.
- **Nasleđivanje** je veza između klasa koja podrazumeva preuzimanje sadržaja osnovnih klasa, tzv. **roditelja** i na taj način, uz mogućnost modifikacije preuzetog sadržaja i dodavanja novog dobija se izvedena klasa, tzv. **potomak**.
- Klasa od koje se preuzima sadržaj naziva se *roditeljska klasa*, *klasa-predak*, *osnovna klasa*, *klasa-davalac* ili *natklasa*. Mi ćemo koristiti termin *natklasa*.
- Klasa koja prima sadržaj uz mogućnost modifikacije i u koju se pored toga može još i dodati nov sadržaj naziva se *klasa-potomak*, *klasa-primalac*, *izvedena klasa* ili *potklasa*. Mi ćemo koristiti termin *potklasa*.
- Potklasa se može izvesti iz jedne ili više natklasa. Ukoliko se potklasa izvodi iz više natklasa tada se takvo nasleđivanje naziva **višestrukim**.
- Objekti potklase poseduju sve sadržaje (sa ili bez modifikacija), koje poseduju i objekti njihove natklase, a pored toga mogu posedovati još i dodati sadržaj koji je karakterističan samo za njih.
- Na osnovu toga, u svakom objektu potklase može se razlikovati **roditeljski deo** i **deo koji je specifičan za samog potomka**.

6.1 Realizacija nasleđivanja

- Sada ćemo razmotriti realizaciju nasleđivanja u programskom jeziku C++. Neka je data klasa `A`:

```
class A {

        private:
                        . . .
        protected:
                        . . .
        public:
                        . . .

};
```

- Klasa `B` se izvodi iz klase `A` na sledeći način:

```
                private
class B  :      protected      A {
                public


          private:
                              . . .
          protected:
                              . . .
          public:
                              . . .
};
```

- Ponašanje članova natklase u zavisnosti od načina izvođenja, koji može biti **`private`**, **`protected`** ili **`public`**, je dato u sledećoj tabeli:

Član u natklasi je:	Način izvođenja je:	Isti član u potklasi je:
public	**public**	**public**
public	**protected**	**protected**
public	**private**	**private**
protected	**public**	**protected**
protected	**protected**	**protected**
protected	**private**	**private**
private	**public**	**nije vidljiv**
private	**protected**	**nije vidljiv**
private	**private**	**nije vidljiv**

- Iz tabele vidimo da se `private` članovima natklase može direktno pristupati u potklasi samo preko metoda koje su ili `protected` ili `public` u natklasi.
- Razlika u pristupanju `private` i `protected` članu klase je sledeća:
  - Članu klase koji je `private` može se direktno pristupati samo iz metoda te klase i njenih prijateljskih funkcija.
  - Članu klase koji je `protected` može se direktno pristupati iz metoda te klase, njenih prijateljskih funkcija i metoda njenih potklasa.
- NAPOMENE:
  - Konstruktori i destruktori se ne nasleđuju.
  - Prijateljstvo se ne nasleđuje.

Zadatak 6.1.1

Napisati klasu `Trougao`. Iz klase `Trougao` izvesti klasu `JKTrougao` (jednakokraki trougao). Iz klase `JKTrougao` izvesti klasu `JSTrougao` (jednakostranični trougao).

```
// Datoteka: trougao.hpp
#ifndef TROUGAO_DEF
#define TROUGAO_DEF
#include <math.h>

class Trougao {
 protected:
   double a, b, c;
 public:
   Trougao() { a=3; b=4; c=5; }
   Trougao(double aa, double bb, double cc) { a=aa; b=bb; c=cc; }
   Trougao(const Trougao &t) { a=t.a; b=t.b; c=t.c; }
   double getA() const { return a; }
   double getB() const { return b; }
   double getC() const { return c; }
   void setA(double aa) { a=aa; }
   void setB(double bb) { b=bb; }
   void setC(double cc) { c=cc; }
   double getO() const { return a+b+c; }
   double getP() const {
       double s=(a+b+c)/2;
       return sqrt(s*(s-a)*(s-b)*(s-c));
   }
};
#endif

// Datoteka: jktrougao.hpp
#ifndef JKTROUGAO_DEF
#define JKTROUGAO_DEF
#include "trougao.hpp"

class JKTrougao : public Trougao {
 public:
   JKTrougao() : Trougao(1,2,2) {}
   JKTrougao(double aa, double bb) : Trougao(aa, bb, bb) {}
   JKTrougao(const JKTrougao &jkt) : Trougao(jkt.a, jkt.b, jkt.c) {}
};
#endif

// Datoteka: jstrougao.hpp
#ifndef JSTROUGAO_DEF
#define JSTROUGAO_DEF
#include "jktrougao.hpp"

class JSTrougao : public JKTrougao {
 public:
   JSTrougao() : JKTrougao(1,1) {}
   JSTrougao(double aa) : JKTrougao(aa, aa) {}
   JSTrougao(const JSTrougao &jst) : JKTrougao(jst.a, jst.b) {}
};
#endif
```

```
// Datoteka: main.cpp
#include "jstrougao.hpp"
#include <iostream>
using namespace std;

int main(){
    Trougao t1(1,4,4);
    JKTrougao jk1(2,3);
    JSTrougao js1(5);
    cout<<t1.getP()<<endl;
    cout<<jk1.getP()<<endl;
    cout<<js1.getP()<<endl;
    return 0;
}
```

Zadatak 6.1.2

Napisati klasu `Pravougaonik`. Iz klase `Pravougaonik` izvesti klasu `Kvadrat`.

```
// Datoteka: pravougaonik.hpp
#ifndef PRAVOUGAONIK_DEF
#define PRAVOUGAONIK_DEF
#include <iostream>
using namespace std;

class Pravougaonik {
    protected:
        double a, b;
    public:
        Pravougaonik(double aa=1, double bb=2) { a=aa; b=bb; }
        Pravougaonik(const Pravougaonik &p) { a=p.a; b=p.b; }
        void setA(double aa) { a=aa; }
        void setB(double bb) { b=bb; }
        double getA() const { return a; }
        double getB() const { return b; }
        double getO() const { return 2*a+2*b; }
        double getP() const { return a*b; }
};
#endif



// Datoteka: kvadrat.hpp
#ifndef KVADRAT_DEF
#define KVADRAT_DEF
#include "pravougaonik.hpp"

class Kvadrat : public Pravougaonik {
    public:
        Kvadrat(double aa=1) : Pravougaonik(aa,aa) {}
        Kvadrat(const Kvadrat &k) : Pravougaonik(k.a, k.b) {}
};
#endif
```

```
// Datoteka: main.cpp
#include "kvadrat.hpp"

int main() {
    Pravougaonik p1(2,5);
    cout<<"Obim p1: "<<p1.getO()<<endl;
    cout<<"Povrsina p1: "<<p1.getP()<<endl;
    Kvadrat k1(4);
    cout<<"Obim k1: "<<k1.getO()<<endl;
    cout<<"Povrsina k1: "<<k1.getP()<<endl;
    return 0;
}
```

- Klasa `Kvadrat` nasleđuje sve sadržaje od natklase `Pravougaonik`. Da bi svaki objekat klase `Kvadrat` bio u dozvoljenom stanju potrebno je da polja `a` i `b`, koja je klasa `Kvadrat` nasledila od klase `Pravougaonik`, budu jednaka. Zbog toga, konstruktor klase `Kvadrat` izgleda ovako:

```
Kvadrat(double aa=1) : Pravougaonik(aa,aa) {}
```

- Konstruktor klase `Kvadrat` ima jedan parametar tipa `double`. To je vrednost koja predstavlja dužinu stranice kvadrata kojeg taj objekat treba da predstavlja. Podrazumevana vrednost za dužinu stranice kvadrata je 1. Ta vrednost se dva puta prosleđuje u konstruktor klase `Pravougaonik` i na taj način se postiže da konstruktor klase `Pravougaonik` izgradi roditeljski deo datog objekta klase `Kvadrat` tako da polja `a` i `b` budu jednaka i to baš parametru koji predstavlja vrednost dužine stranice kvadrata. Drugim rečima, konstuktor klase `Kvadrat` *radi po principu* – kvadrat je pravougaonik čije su sve stranice jednake.
- Konstruktor kopije ima zadatak da od već postojećeg objekta klase `Kvadrat` napravi njegovu kopiju i to radi tako što kopira odgovarajuća polja. Konstruktor kopije klase `Kvadrat` izgleda ovako:

```
Kvadrat(const Kvadrat &k) : Pravougaonik(k.a, k.b) {}
```

- Konstruktor kopije kopira polje `a` iz originala u polje `a` kopije i polje `b` iz originala u polje `b` kopije. Pošto važi da su polja `a` i `b` jednaka, ovaj konstruktor možemo napisati i na sledeća tri načina, a rezultat će biti isti:

```
Kvadrat(const Kvadrat &k) : Pravougaonik(k.a, k.b) {}
Kvadrat(const Kvadrat &k) : Pravougaonik(k.a, k.a) {}
Kvadrat(const Kvadrat &k) : Pravougaonik(k.b, k.b) {}
```

- Sve ostale metode (za računanje obima i površine) nije potrebno ponovo pisati u klasi `Kvadrat`, jer ih je klasa `Kvadrat` nasledila. Te metode ispravno rade i za slučaj *pravougaonika čije su stranice jednake dužine*, tj. kvadrata. Na primer, metoda za računanje površine pravougaonika izračunava proizvod `a*b`. Kod kvadrata su polja `a` i `b` jednaka, pa se to svodi na `a*a` ili `b*b` i rezultat odgovara površini kvadrata.

Zadatak 6.1.3

Napisati klasu `Pravougaonik`. Napisati klasu `Kvadar` koja sadrži `B` (objekat klase `Pravougaonik`) i `M` (objekat klase `Pravougaonik`). Iz klase `Kvadar` izvesti klasu `Kocka`.

```
// Datoteka: pravougaonik.hpp
#ifndef PRAVOUGAONIK_DEF
#define PRAVOUGAONIK_DEF
#include <iostream>
using namespace std;

class Pravougaonik {
    protected:
        double a, b;
    public:
        Pravougaonik(double aa=1, double bb=2) { a=aa; b=bb; }
        Pravougaonik(const Pravougaonik &p) { a=p.a; b=p.b; }
        void setA(double aa) { a=aa; }
        void setB(double bb) { b=bb; }
        double getA() const { return a; }
        double getB() const { return b; }
        double getO() const { return 2*a+2*b; }
        double getP() const { return a*b; }
};

#endif



// Datoteka: kvadar.hpp
#ifndef KVADAR_DEF
#define KVADAR_DEF
#include "pravougaonik.hpp"

class Kvadar {
    protected:
        Pravougaonik B;
        Pravougaonik M;
    public:
        Kvadar(double aa=1,double bb=2,double hh=1) :
        B(aa,bb), M(2*aa+2*bb,hh) {}
        Kvadar(const Kvadar &kv) : B(kv.B), M(kv.M) {}
        double getA() const { return B.getA(); }
        double getB() const { return B.getB(); }
        double getH() const { return M.getB(); }
        double getP() const { return 2*B.getP() + M.getP(); }
        double getV() const { return B.getP()*getH();}
};

#endif
```

```
// Datoteka: kocka.hpp
#ifndef KOCKA_DEF
#define KOCKA_DEF
#include "kvadar.hpp"

class Kocka : public Kvadar {
    public:
        Kocka(double aa=1) : Kvadar(aa,aa,aa) {}
        Kocka(const Kocka &k) : Kvadar((Kvadar)k) {}
};

#endif


// Datoteka: main.cpp
#include "kocka.hpp"

int main(){
    Kvadar kv1(3,4,5), kv2(kv1);
    cout<<"Povrsina kv1: "<<kv1.getP()<<endl;
    cout<<"Zapremina kv1: "<<kv1.getV()<<endl;
    cout<<"Povrsina kv2: "<<kv2.getP()<<endl;
    cout<<"Zapremina kv2: "<<kv2.getV()<<endl;
    Kocka ko1(5), ko2(ko1);
    cout<<"Povrsina ko1: "<<ko1.getP()<<endl;
    cout<<"Zapremina ko1: "<<ko1.getV()<<endl;
    cout<<"Povrsina ko2: "<<ko2.getP()<<endl;
    cout<<"Zapremina ko2: "<<ko2.getV()<<endl;
    return 0;
}
```

- Objekat klase `Kvadar` u sebi sadrži komponente `B` i `M` (objekti klase `Pravougaonik`). Konstruktor klase `Kvadar` ima tri parametra – vrednost dužine, širine i visine kvadra. Konstruktor klase `Kvadar` će izgledati ovako:

```
Kvadar(double aa=1, double bb=2, double hh=1) :
B(aa,bb), M(2*aa+2*bb,hh) {}
```

- Osnova kvadra, odnosno objekat-član `B` jeste objekat klase `Pravougaonik`, čije polje `a` sadrži vrednost dužine (parametar `aa`), a polje `b` sadrži vrednost širine kvadra (parametar `bb`). Omotač kvadra, odnosno objekat-član `M` jeste objekat klase `Pravougaonik`, čije polje `a` sadrži obim osnove (a to je `2*aa+2*bb`), a polje `b` sadrži vrednost visine kvadra (parametar `hh`).
- Konstruktor kopije klase `Kvadar` izgleda ovako:

```
Kvadar(const Kvadar &kv) : B(kv.B), M(kv.M) {}
```

- U konstruktor kopije klase `Kvadar` se po referenci prenosi objekat iste klase i to je original na osnovu kojeg se pravi kopija. Kopija se gradi tako što se objekat-član `B` gradi na osnovu objekta-člana `B` iz originala (u konstruktoru je to `kv.B`), a objekat-član `M` gradi na osnovu objekta-člana `M` iz originala (u kons-

truktoru je to `kv.M`). Izgradnju objekta-člana `B` i objekta-člana `M` (koji su objekti klase `Pravougaonik`) će uraditi konstruktor kopije klase `Pravougaonik`, što se i moglo očekivati, s obzirom da se radi o situaciji kada se od nekog objekta pravi njegova kopija, a tu situaciju pokriva konstruktor kopije u datoj klasi.

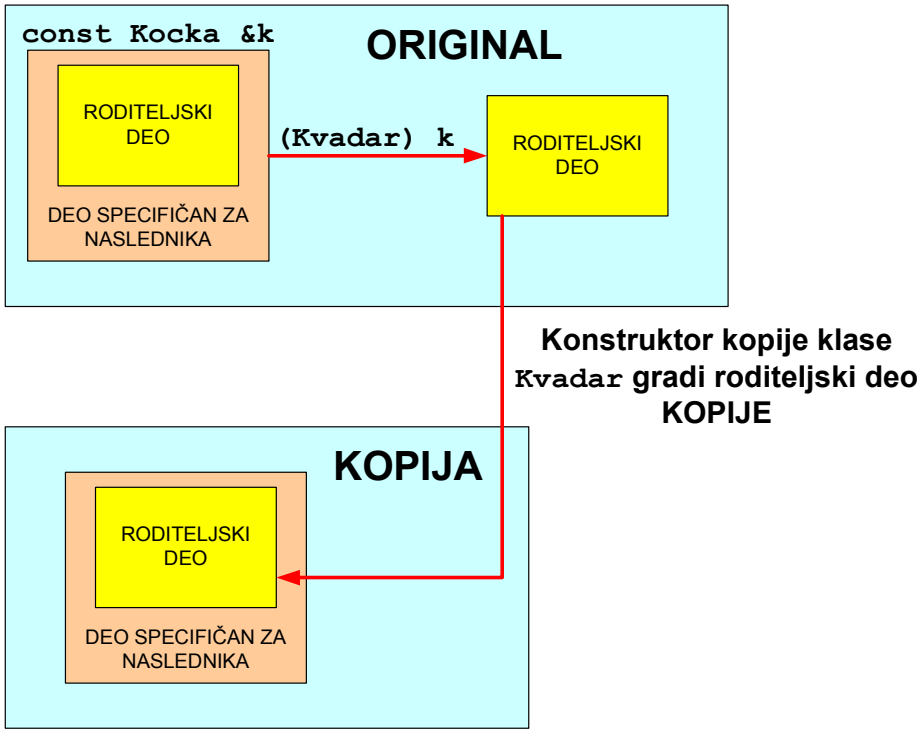
- Klasa `Kocka` nasleđuje sve sadržaje od roditeljske klase `Kvadar`. Konstruktor klase `Kocka` *radi po principu* – kocka je kvadar čije su sve ivice jednake. Zbog toga, konstruktor klase `Kocka` izgleda ovako:

```
Kocka(double aa=1) : Kvadar(aa,aa,aa) {}
```

- Konstruktor kopije klase `Kocka` izgleda ovako:

```
Kocka(const Kocka &k) : Kvadar((Kvadar)k) {}
```

- U konstruktor kopije klase `Kocka` se po referenci prenosi objekat iste klase i to je original na osnovu kojeg se pravi kopija. Eksplicitnom promenom tipa originala `Kocka` u tip `Kvadar` (u konstruktoru je to napisano kao `(Kvadar)k`) koristi se samo roditeljski deo originala (koji inače i pripada klasi `Kvadar` iz koje je izvedena klasa `Kocka`), pa će konstruktor kopije klase `Kvadar` izgraditi roditeljski deo novog objekta, što je ilustrovano na slici 11.



Slika 11.

6.2 Metode izvedene klase

- Potklasa nasleđuje sve sadržaje od natklase. Metode potklase možemo podeliti u tri grupe:
  - *preuzete* ili *nasleđene metode*, koje se bez modifikacija preuzimaju iz natklase,
  - *redefinisane metode*, koje se sa modifikacijama preuzimaju iz natklase,
  - *dodate metode*, koje se dodaju potklasi.
- Prvo ćemo razmotriti pozive konstruktora kod izgradnje objekta potklase i pozive destruktora kod uništavanja objekta potklase, a zatim pozive preuzete, redefinisane i dodate metode.
- Metoda se redefiniše tako što se u potklasi napiše metoda koja ima isto ime i istu listu parametara.

Zadatak 6.2.1
Analizirati ispis sledećeg programa:

```
// Datoteka: klasaa.hpp
#ifndef KLASA_A_DEF
#define KLASA_A_DEF

#include <iostream>
using namespace std;

class A {
    protected:
        int a;
    public:
        A() {
            a=0;
            cout<<"A: Konstruktor 1."<<endl;
        }
        A(int aa) {
            a=aa;
            cout<<"A: Konstruktor 2."<<endl;
        }
        A(const A &x) {
            a=x.a;
            cout<<"A: Konstruktor 3."<<endl;
        }
        ~A() {  cout<<"A: Destruktor."<<endl; }
        void p() {  cout<<"A: Preuzimanje."<<endl; }
        void r() {  cout<<"A: Redefinisanje."<<endl; }

};
#endif


// Datoteka: klasab.hpp
#ifndef KLASA_A_DEF
#define KLASA_A_DEF
```

```
#include "klasaa.hpp"

class B : public A {
    public:
        B() : A(0) { cout<<"B: Konstruktor 1."<<endl; }
        B(int aa) : A(aa) { cout<<"B: Konstruktor 2."<<endl; }
        B(const B &x) : A((A)x) { cout<<"B: Konstruktor 3."<<endl; }
        ~B() {  cout<<"B: Destruktor."<<endl; }

        void r() { cout<<"B: Redefinisanje."<<endl; }
        void d() { cout<<"B: Dodavanje."<<endl; }

};
#endif


// Datoteka: main.cpp
#include "klasab.hpp"

int main() {
    cout<<"*** Kreiranje objekata a1, a2 i a3"<<endl;
    A a1, a2(7), a3(a2);
    cout<<endl<<"*** Kreiranje objekata b1, b2 i b3"<<endl;
    B b1, b2(7), b3(b2);
    cout<<endl<<"*** Metode natklase"<<endl;
    a1.p();
    a2.r();
    cout<<endl<<"*** Metode potklase"<<endl;
    b1.p();
    b2.r();
    b3.d();
    cout<<endl<<"*** Unistavanje objekata"<<endl;
    return 0;
}
```

Pokretanjem programa dobijamo sledeći ispis:

```
*** Kreiranje objekata a1, a2 i a3
A: Konstruktor 1.
A: Konstruktor 2.
A: Konstruktor 3.

*** Kreiranje objekata b1, b2 i b3
A: Konstruktor 2.
B: Konstruktor 1.
A: Konstruktor 2.
B: Konstruktor 2.
A: Konstruktor 3.
B: Konstruktor 3.

*** Metode natklase
A: Preuzimanje.
A: Redefinisanje.
```

```
*** Metode potklase
A: Preuzimanje.
B: Redefinisanje.
B: Dodavanje.

*** Unistavanje objekata
B: Destruktor.
A: Destruktor.
B: Destruktor.
A: Destruktor.
B: Destruktor.
A: Destruktor.
A: Destruktor.
A: Destruktor.
A: Destruktor.
```

- Objekat `a1` kreira konstruktor bez parametara klase `A`. Objekat `a2` kreira konstruktor sa parametrima klase `A`. Objekat `a3` kreira konstruktor kopije klase `A`. Objekat `b1` kreira konstruktor bez parametara klase `B`, koji podrazumeva da će roditeljski deo objekta izgraditi konstruktor bez parametara klase `A`. Objekat `b2` kreira konstruktor sa parametrima klase `B`, koji podrazumeva da će roditeljski deo objekta izgraditi konstruktor sa parametrima klase `A`. Objekat `b3` kreira konstruktor kopije klase `B`, koji podrazumeva da će roditeljski deo objekta izgraditi konstruktor kopije klase `A`.

- Metoda `p()` se preuzima bez modifikacija. Prema tome, poziv ove metode bilo preko objekta natklase `a1.p()`, bilo preko objekta potklase `b1.p()` daje isti ispis:

  ```
  A: Preuzimanje.
  ```

- Metoda `r()` se redefiniše, tj. preuzima sa modifikacijama. Prema tome, poziv ove metode preko objekta natklase `a2.p()` daje ispis:

  ```
  A: Redefinisanje.
  ```

  , dok poziv ove metode preko objekta potklase `b2.p()` daje isti ispis:

  ```
  B: Redefinisanje.
  ```

- Metoda `d()` se dodaje potklasi i može se pozvati jedino preko objekta potklase. Poziv ove metode `b3.d()` daje ispis:

  ```
  B: Dodavanje.
  ```

- Na kraju funkcije `main()` potrebno je uništiti sve postojeće objekte i to je trenutak kada će biti pozvani destruktori.

Zadatak 6.2.2

Iz klase `Osoba` koja je data u zadatku 5.1.3 izvesti klasu `Student`, a zatim iz klase `Student` izvesti klasu `PhDStudent`. Analizirati pozive konstruktora i destruktora.

```
// Datoteka: student.hpp
#ifndef STUDENT_DEF
#define STUDENT_DEF
#include "osoba.hpp"

class Student : public Osoba {
  protected:
      int brojIndeksa;
  public:
      Student(const char *s1="", const char *s2="", int i=0) :
      Osoba(s1,s2), brojIndeksa(i) {
         cout<<"Student: Konstruktor 1."<<endl;
      }
      Student(const DinString &ds1, const DinString &ds2, int i) :
      Osoba(ds1,ds2), brojIndeksa(i) {
         cout<<"Student: Konstruktor 2."<<endl;
      }
      Student(const Osoba &os, int i) : Osoba(os), brojIndeksa(i) {
         cout<<"Student: Konstruktor 3."<<endl;
      }
      Student(const Student &s) : Osoba((Osoba)s),
      brojIndeksa(s.brojIndeksa) {
         cout<<"Student: Konstruktor 4."<<endl;
      }
      ~Student() {
         cout<<"Student: Destruktor."<<endl;
      }
      void predstaviSe() const {
         Osoba::predstaviSe();
         cout<<"Broj mog indeksa je "<<brojIndeksa<<"."<<endl;
      }
};

#endif



// Datoteka: phdstudent.hpp
#ifndef PHDSTUDENT_DEF
#define PHDSTUDENT_DEF

#include "student.hpp"

class PhDStudent : public Student {
  protected:
      double prosecnaOcena;
    public:
      PhDStudent(const char *s1="", const char *s2="", int i=0,
      double po=0) : Student(s1,s2,i), prosecnaOcena(po) {
         cout<<"PhDStudent: Konstruktor 1."<<endl;
      }
```

```
      PhDStudent(const DinString &ds1, const DinString &ds2, int i,
      double po) : Student(ds1,ds2,i), prosecnaOcena(po) {
         cout<<"PhDStudent: Konstruktor 2."<<endl;
      }
      PhDStudent(const Osoba &os, int i, double po) : Student(os,i),
      prosecnaOcena(po) {
         cout<<"PhDStudent: Konstruktor 3."<<endl;
      }
      PhDStudent(const Student &s, double po) : Student(s),
      prosecnaOcena(po) {
         cout<<"PhDStudent: Konstruktor 4."<<endl;
      }
      PhDStudent(const PhDStudent &phds) : Student((Student)phds),
      prosecnaOcena(phds.prosecnaOcena) {
         cout<<"PhDStudent: Konstruktor 5."<<endl;
      }
      ~PhDStudent() {
         cout<<"PhDStudent: Destruktor."<<endl;
      }
      void predstaviSe() const {
         Student::predstaviSe();
         cout<<"Diplomirao sam sa prosecnom ocenom "<<prosecnaOcena;
         cout<<", a sada sam student doktorskih studija"<<endl;
      }
};

#endif


// Datoteka: main.cpp
#include "phdstudent.hpp"

int main() {
    const char *s1 = "Petar";
    const char *s2 = "Petrovic";
    const char *s3 = "Jovan";
    const char *s4 = "Jovanovic";
    cout<<"*** Kreiranje objekata ds1, ds2, ds3 i ds4."<<endl;
    DinString ds1(s1), ds2(s2), ds3(s3), ds4(s4);
    cout<<endl<<"*** Kreiranje objekata os1, os2 i os3."<<endl;
    Osoba os1(s1,s2), os2(ds3,ds4), os3(os2);
    cout<<endl<<"*** Kreiranje objekata st1, st2, st3 i st4."<<endl;
    Student st1(s1,s2,1234), st2(ds1,ds2,1234), st3(os2,1234),
st4(st2);
    cout<<endl<<"*** Kreiranje objekata phds1, phds2, phds3 i
phds4."<<endl;
    PhDStudent phds1(s1,s2,1234,8.56), phds2(ds1,ds2,1234,8.56),
phds3(os3,1234,8.77), phds4(st3,8.77);
    cout<<endl<<"*** Predstavljenje."<<endl;
    os1.predstaviSe();
    os2.predstaviSe();
    os3.predstaviSe();
    st1.predstaviSe();
    st2.predstaviSe();
    st3.predstaviSe();
```

```
    st4.predstaviSe();
    phds1.predstaviSe();
    phds2.predstaviSe();
    phds3.predstaviSe();
    phds4.predstaviSe();
    cout<<endl<<"*** Unistavanje objekata."<<endl;
    return 0;
}
```

Pokretanjem programa dobijamo sledeći ispis:

```
*** Kreiranje objekata ds1, ds2, ds3 i ds4.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
DinString: Konstruktor 2.

*** Kreiranje objekata os1, os2 i os3.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
Osoba: Konstruktor 1.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 2.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.

*** Kreiranje objekata st1, st2, st3 i st4.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
Osoba: Konstruktor 1.
Student: Konstruktor 1.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 2.
Student: Konstruktor 2.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.
Student: Konstruktor 3.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.
Student: Konstruktor 4.

*** Kreiranje objekata phds1, phds2, phds3 i phds4.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
Osoba: Konstruktor 1.
Student: Konstruktor 1.
PhDStudent: Konstruktor 1.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
```

```
Osoba: Konstruktor 2.
Student: Konstruktor 2.
PhDStudent: Konstruktor 2.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.
Student: Konstruktor 3.
PhDStudent: Konstruktor 3.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.
Student: Konstruktor 4.
PhDStudent: Konstruktor 4.

*** Predstavljenje.
Zovem se Petar Petrovic.
Zovem se Jovan Jovanovic.
Zovem se Jovan Jovanovic.
Zovem se Petar Petrovic.
Broj mog indeksa je 1234.
Zovem se Petar Petrovic.
Broj mog indeksa je 1234.
Zovem se Jovan Jovanovic.
Broj mog indeksa je 1234.
Zovem se Petar Petrovic.
Broj mog indeksa je 1234.
Zovem se Petar Petrovic.
Broj mog indeksa je 1234.
Diplomirao sam sa prosecnom ocenom 8.56, a sada sam student
doktorskih studija
Zovem se Petar Petrovic.
Broj mog indeksa je 1234.
Diplomirao sam sa prosecnom ocenom 8.56, a sada sam student
doktorskih studija
Zovem se Jovan Jovanovic.
Broj mog indeksa je 1234.
Diplomirao sam sa prosecnom ocenom 8.77, a sada sam student
doktorskih studija
Zovem se Jovan Jovanovic.
Broj mog indeksa je 1234.
Diplomirao sam sa prosecnom ocenom 8.77, a sada sam student
doktorskih studija

*** Unistavanje objekata.
PhDStudent: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
PhDStudent: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
PhDStudent: Destruktor.
```

```
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
PhDStudent: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
```

Zadatak 6.2.3

Napisati klasu `List` koja modeluje jednostrukospregnutu listu sa generičkim pokazivačem. Iz klase `List` izvesti klasu `IntStack` koja modeluje stek celih brojeva. Način izvođenja klase `IntStack` treba da bude `private`.

```
// Datoteka: list.hpp
#ifndef LIST_DEF
#define LIST_DEF
#include <stdlib.h>
#include <iostream>
using namespace std;
```

```
struct listEl{
    void *content;
    struct listEl *next;
};

class List{
    private:
        listEl *head;
        int noEl;
    public:
        List();
        List(const List&);
        ~List();
        List& operator=(const List&);
        int size() const { return noEl; }
        bool empty() const { return head==NULL; }
        bool addEl(int, void*);
        bool deleteEl(int);
        void* readEl(int) const;
        void clear();
};
#endif


// Datoteka: list.cpp
#include "list.hpp"

List::List() {
    head=NULL;
    noEl=0;
}

List::List(const List &rl) {
    head=NULL;
    noEl=0;
    for(int i=1; i<=rl.noEl; i++)
        addEl(i,rl.readEl(i));
}

List::~List() {
    clear();
}

List& List::operator=(const List &rl) {
    if(this!=&rl) {
        clear();
        head=NULL;
        noEl=0;
        for(int i=1; i<=rl.noEl; i++)
            addEl(i,rl.readEl(i));
    }
    return *this;
}
```

```
bool List::addEl(int n, void *newContent){
    if(n<1 || n>noEl+1)
        return false;
    else {
        listEl *newEl=new listEl();
        if(newEl==NULL)
            return false;
        else {
            newEl->content=newContent;
            if(n==1) {
                newEl->next=head;
                head=newEl;
                noEl++;
            } else {
                listEl *temp=head;
                for(int i=2; i<n; i++)
                    temp=temp->next;
                newEl->next=temp->next;
                temp->next=newEl;
                noEl++;
            }
        return true;
        }
    }
}

bool List::deleteEl(int n){
    if(n<1 || n>noEl)
        return false;
    else {
        if(n==1) {
            listEl *del=head;
            head=head->next;
            delete del;
            noEl--;
        } else {
            listEl *temp=head;
            for(int i=2; i<n; i++)
                temp=temp->next;
            listEl *del=temp->next;
            temp->next=del->next;
            delete del;
            noEl--;
        }
    return true;
    }
}

void* List::readEl(int n)const{
    if(n<1 || n>noEl)
        return NULL;
    else {
        listEl *temp=head;
        for(int i=1; i<n; i++)
            temp=temp->next;
```

```
        return temp->content;
    }
}

void List::clear(){
    while(!empty())
        deleteEl(1);
}



// Datoteka: intstack.hpp
#ifndef INT_STACK_DEF
#define INT_STACK_DEF

#include "list.hpp"

class IntStack : private List{
    private:
        void clear();
    public:
        IntStack() {}
        IntStack(const IntStack&);
        ~IntStack();
        IntStack& operator=(const IntStack&);
        friend ostream& operator<<(ostream&, const IntStack&);
        int top() const;
        void pop();
        void push(int element);
        bool emptyStack() const;
        int stackSize() const;
};
#endif



// Datoteka: intstack.cpp
#include "intstack.hpp"

void IntStack::clear() {
    while(!empty()) {
        int *tmp=(int*)readEl(1);
        delete tmp;
        deleteEl(1);
    }
}


IntStack::IntStack(const IntStack &ds) {
    for(int i=ds.size(); i>0; i--) {
        int toCopy=*(int*)ds.readEl(i);
        push(toCopy);
    }
}
```

```
IntStack::~IntStack(){
    clear();
}


IntStack& IntStack::operator=(const IntStack &ds) {
    if(this!=&ds) {
        clear();
        for(int i=ds.size(); i>0; i--) {
            int toCopy=*(int*)ds.readEl(i);
            push(toCopy);
        }
    }
    return *this;
}


ostream& operator<<(ostream &out, const IntStack &s) {
    if(s.stackSize()>0) {
        out<<"----------------------------------------"<<endl;
        out<<*(int*)(s.readEl(1))<<" <-- VRH STEKA"<<endl;
        for(int i=2; i<=s.stackSize(); i++)
            out<<*(int*)(s.readEl(i))<<endl;
        out<<"----------------------------------------"<<endl;
        } else
            out<<"Stek je prazan!";
    return out;
}


int IntStack::top() const {
    int *retVal=(int*)readEl(1);
    if(retVal==NULL) {
        exit(EXIT_FAILURE);
    } else
        return *retVal;
}


void IntStack::pop() {
    int *retVal=(int*)readEl(1);
    if(retVal==NULL)
        exit(EXIT_FAILURE);
    else{
        delete retVal;
        deleteEl(1);
    }
}


void IntStack::push(int element) {
    int *tmp=new int(element);
    if(tmp==NULL)
        exit(EXIT_FAILURE);
```

```
    else {
        if(!addEl(1,tmp)) {
            delete tmp;
            exit(EXIT_FAILURE);
        }
    }
}


bool IntStack::emptyStack()const{
    return List::empty();
}


int IntStack::stackSize()const{
    return List::size();
}



// Datoteka: main.cpp
#include "intstack.hpp"

int main(){
    IntStack a;
    cout<<a<<endl;
    a.push(1);
    cout<<a<<endl;
    a.push(2);
    a.push(3);
    cout<<a<<endl;
    cout<<"Praznjenje steka."<<endl;
    while(!a.emptyStack()){
        cout<<"Na vrhu steka je:"<<a.top()<<endl;
        a.pop();
    }
    cout<<endl<<"Punjenje steka."<<endl;
    for(int i=0; i<10; i++)
        a.push(i*10);
    cout<<a<<endl;
    return 0;
}
```

6.3 Virtuelne metode

- Razlikujemo **statičke** i **virtuelne** metode.
- Virtualne metode razlikuju se od statičkih po tome što se na mestu poziva u prevedenom kodu ne nalazi direktan skok na njihov početak. Umesto toga, na nivou klase formira se posebna tabela koja, pored još nekih podataka, sadrži adrese svih virtuelnih metoda koje postoje u klasi (ako postoje). Prilikom poziva virtuelnih metoda prethodno se iz te tabele čita njena adresa i tek na bazi te adrese izvršava se instrukcija skoka. Pri tom, svaki objekat sadrži adresu tabele u

sopstvenom memorijskom prostoru, tako da se na bazi sadržaja datog objekta, a ne njegove klase određuje verzija pozvane virtuelne metode. Ove tabele nose naziv *v-tabele*.

- Metoda se proglašava virtuelnom u roditeljskoj klasi tako što se ispred tipa metode navede rezervisana reč **`virtual`**.
- Metoda se proglašava virtuelnom samo jedanput, i to u roditeljskoj klasi i tada u svim klasama naslednicama ta metoda zadržava tu osobinu.
- Redefinisana virtuelna metoda mora imati isti prototip kao originalna.

Zadatak 6.3.1
Analizirati ispis sledećeg programa:

```
// Datoteka: klase.hpp
#ifndef KLASE_DEF
#define KLASE_DEF

#include <iostream>
using namespace std;

class A1 {
    public:
        void f() { cout<<"Klasa A1: f()"<<endl; }
};

class A2 {
    public:
        virtual void f() { cout<<"Klasa A2: f()"<<endl;}
};

class B1 : public A1 {
    public:
        void f() { cout<<"Klasa B1: f()"<<endl; }
};

class B2 : public A2 {
    public:
        void f() { cout<<"Klasa B2: f()"<<endl;}
};

#endif



// Datoteka: main.cpp
#include "klase.hpp"

int main() {

    A1 a1;
    A2 a2;
    B1 b1;
    B2 b2;
```

```
    a1.f();
    a2.f();
    b1.f();
    b2.f();

    A1 *pa1;
    A2 *pa2;
    pa1=&b1;
    pa2=&b2;

    cout<<" *** Pokazivaci *** "<<endl;
    pa1->f();
    pa2->f();

    return 0;
}
```

Pokretanjem programa dobijamo sledeći ispis:

```
Klasa A1: f()
Klasa A2: f()
Klasa B1: f()
Klasa B2: f()
 *** Pokazivaci ***
Klasa A1: f()
Klasa B2: f()
```

- U klasi `A1` nalazi se metoda `f()` koja nije virtuelna, dok u klasi `A2` nalazi se metoda `f()` koja je virtuelna. Iz klase `A1` izvedena je klasa `B1` u kojoj je redefinisana metoda `f()`. Iz klase `A2` izvedena je klasa `B2` u kojoj je redefinisana metoda `f()`. Poziv metode `f()` za objekte `a1` ili `a2` podrazumeva poziv metoda `f()` iz klasa `A1`, odnosno `A2`. Poziv metoda `f()` za objekte `b1` ili `b2` podrazumeva poziv metoda `f()`, iz klasa `B1`, odnosno `B2`. Nakon toga, deklarisana su dva pokazivača na objekte roditeljskih klasa `pa1` i `pa2` i dodeljene su im adrese redom objekata `a1` i `a2`. Tada `pa1->f()` predstavlja poziv roditeljske metode `f()` i biće ispisana poruka: `Klasa A1: f()`, a `pa2->f()` predstavlja poziv potomkove metode `f()` i biće ispisana poruka: `Klasa B2: f()`. Razlog za to jeste u tome što je metoda `f()` u klasi `A2` proglašena virtuelnom.

Zadatak 6.3.2

Izmeniti klasu `Osoba` iz zadatka 5.1.3 tako da metoda `predstaviSe()` bude virtuelna, a zatim iz klase `Osoba` izvesti klasu `Student` (kao u zadatku 6.2.2) i klasu `Zaposleni` koja sadrži još i polja: `brojRadnihSati` i `vrednostSata` I metodu `izracunajPlatu()`, koja izračunava i vraća vrednost proizvoda:

`brojRadnihSati*vrednostSata`.

Napisati slobodnu funkciju `predstavljanje(const Osoba&)`.

```
// Datoteka: osoba.hpp
#ifndef OSOBA_DEF
#define OSOBA_DEF
#include "dinstring.hpp"
#include <iostream>
using namespace std;

class Osoba {
    protected:
        DinString ime, prezime;
    public:
        Osoba(const char *s1="", const char *s2="") :
        ime(s1), prezime(s2) {
            cout<<"Osoba: Konstruktor 1."<<endl;
        }
        Osoba(const DinString &ds1, const DinString &ds2) :
        ime(ds1), prezime(ds2) {
            cout<<"Osoba: Konstruktor 2."<<endl;
        }
        Osoba(const Osoba &ro) : ime(ro.ime), prezime(ro.prezime) {
            cout<<"Osoba: Konstruktor 3."<<endl;
        }
        ~Osoba() {
            cout<<"Osoba: Destruktor."<<endl;
        }
        virtual void predstaviSe() const {
            cout<<"Zovem se "<<ime<<" "<<prezime<<"."<<endl;
        }
};
#endif



// Datoteka: student.hpp
#ifndef STUDENT_DEF
#define STUDENT_DEF
#include "osoba.hpp"

class Student : public Osoba {
  protected:
      int brojIndeksa;
  public:
      Student(const char *s1="", const char *s2="", int i=0) :
      Osoba(s1,s2), brojIndeksa(i) {
          cout<<"Student: Konstruktor 1."<<endl;
      }
      Student(const DinString &ds1, const DinString &ds2, int i) :
      Osoba(ds1,ds2), brojIndeksa(i) {
          cout<<"Student: Konstruktor 2."<<endl;
      }
      Student(const Osoba &os, int i) : Osoba(os), brojIndeksa(i) {
          cout<<"Student: Konstruktor 3."<<endl;
      }
```

```
      Student(const Student &s) : Osoba((Osoba)s),
      brojIndeksa(s.brojIndeksa) {
          cout<<"Student: Konstruktor 4."<<endl;
      }
      ~Student() {
          cout<<"Student: Destruktor."<<endl;
      }
      void predstaviSe() const {
          cout<<"Zovem se "<<ime<<" "<<prezime;
          cout<<" i moj broj indeksa je "<<brojIndeksa<<"."<<endl;
      }
};
#endif


// Datoteka: zaposleni.hpp
#ifndef ZAPOSLENI_DEF
#define ZAPOSLENI_DEF
#include "osoba.hpp"

class Zaposleni : public Osoba {
  protected:
      int brojRadnihSati;
      double vrednostSata;
  public:
      Zaposleni(const char *s1="", const char *s2="", int brs=0,
double vs=0) : Osoba(s1,s2), brojRadnihSati(brs), vrednostSata(vs) {
          cout<<"Zaposleni: Konstruktor 1."<<endl;
      }
      Zaposleni(const DinString &ds1, const DinString &ds2, int brs,
double vs) : Osoba(ds1,ds2), brojRadnihSati(brs), vrednostSata(vs) {
          cout<<"Zaposleni: Konstruktor 2."<<endl;
      }
      Zaposleni(const Osoba &os, int brs, double vs) : Osoba(os),
brojRadnihSati(brs), vrednostSata(vs) {
          cout<<"Zaposleni: Konstruktor 3."<<endl;
      }
      Zaposleni(const Zaposleni &z) : Osoba((Osoba)z),
brojRadnihSati(z.brojRadnihSati), vrednostSata(z.vrednostSata) {
          cout<<"Zaposleni: Konstruktor 4."<<endl;
      }
      ~Zaposleni() {
          cout<<"Zaposleni: Destruktor."<<endl;
      }
      double izracunajPlatu() const {
          return brojRadnihSati*vrednostSata;
      }
     void predstaviSe() const {
         cout<<"Zovem se "<<ime<<" "<<prezime;
         cout<<" i moja plata iznosi "<<izracunajPlatu()<<"."<<endl;
     }
};
#endif
```

```
// Datoteka: main.cpp
#include "student.hpp"
#include "zaposleni.hpp"

void predstavljanje(const Osoba &os) {
    os.predstaviSe();
}

int main() {
    const char *s1 = "Petar";
    const char *s2 = "Petrovic";
    const char *s3 = "Jovan";
    const char *s4 = "Jovanovic";
    cout<<"*** Kreiranje objekata ds1, ds2, ds3 i ds4."<<endl;
    DinString ds1(s1), ds2(s2), ds3(s3), ds4(s4);
    cout<<endl<<"*** Kreiranje objekata os1, os2 i os3."<<endl;
    Osoba os1(s1,s2), os2(ds3,ds4), os3(os2);
    cout<<endl<<"*** Kreiranje objekata st1, st2, st3 i st4."<<endl;
    Student   st1(s1,s2,1234),   st2(ds1,ds2,1234),   st3(os2,1234),
st4(st2);
    cout<<endl<<"*** Kreiranje objekata z1, z2, z3 i z4."<<endl;
    Zaposleni         z1(s1,s2,176,250),         z2(ds1,ds2,176,250),
z3(os3,176,250), z4(z3);
    cout<<endl<<"*** Predstavljenje."<<endl;
    predstavljanje(os1);
    predstavljanje(os2);
    predstavljanje(os3);
    predstavljanje(st1);
    predstavljanje(st2);
    predstavljanje(st3);
    predstavljanje(st4);
    predstavljanje(z1);
    predstavljanje(z2);
    predstavljanje(z3);
    predstavljanje(z4);
    cout<<endl<<"*** Unistavanje objekata."<<endl;
    return 0;
}
```

Pokretanjem programa dobijamo sledeći ispis:

```
*** Kreiranje objekata ds1, ds2, ds3 i ds4.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
DinString: Konstruktor 2.

*** Kreiranje objekata os1, os2 i os3.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
Osoba: Konstruktor 1.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 2.
DinString: Konstruktor 3.
```

```
DinString: Konstruktor 3.
Osoba: Konstruktor 3.

*** Kreiranje objekata st1, st2, st3 i st4.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
Osoba: Konstruktor 1.
Student: Konstruktor 1.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 2.
Student: Konstruktor 2.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.
Student: Konstruktor 3.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.
Student: Konstruktor 4.

*** Kreiranje objekata z1, z2, z3 i z4.
DinString: Konstruktor 2.
DinString: Konstruktor 2.
Osoba: Konstruktor 1.
Student: Konstruktor 1.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 2.
Student: Konstruktor 2.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.
Student: Konstruktor 3.
DinString: Konstruktor 3.
DinString: Konstruktor 3.
Osoba: Konstruktor 3.
Student: Konstruktor 4.

*** Predstavljenje.
Zovem se Petar Petrovic.
Zovem se Jovan Jovanovic.
Zovem se Jovan Jovanovic.
Zovem se Petar Petrovic i moj broj indeksa je 1234.
Zovem se Petar Petrovic i moj broj indeksa je 1234.
Zovem se Jovan Jovanovic i moj broj indeksa je 1234.
Zovem se Petar Petrovic i moj broj indeksa je 1234.
Zovem se Petar Petrovic i moja plata iznosi 44000.
Zovem se Petar Petrovic i moja plata iznosi 44000.
Zovem se Jovan Jovanovic i moja plata iznosi 44000.
Zovem se Jovan Jovanovic i moja plata iznosi 44000.

*** Unistavanje objekata.
Student: Destruktor.
Osoba: Destruktor.
```

```
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Student: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
Osoba: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
```

6.4 Apstraktne klase

- **Apstraktna metoda** je virtuelna metoda koja nema telo (tj. nema realizaciju).
- Virtuelna metoda se proglašava apstraktnom tako što se na kraju njenog prototipa napiše **=0**, odnosno prototip apstraktne metode izgleda ovako:

```
virtual Tip_metode ime_metode(lista_parametara)=0;
```

- Klasa koja ima bar jednu apstraktnu metodu naziva se **apstraktna klasa**.
- Apstraktna klasa se ne može instancirati, tj. ne može se kreirati objekat apstraktne klase.

Zadatak 6.4.1
Napisati apstraktnu klasu `Figura` i iz nje izvesti klase `Trougao` i `Pravougaonik`.

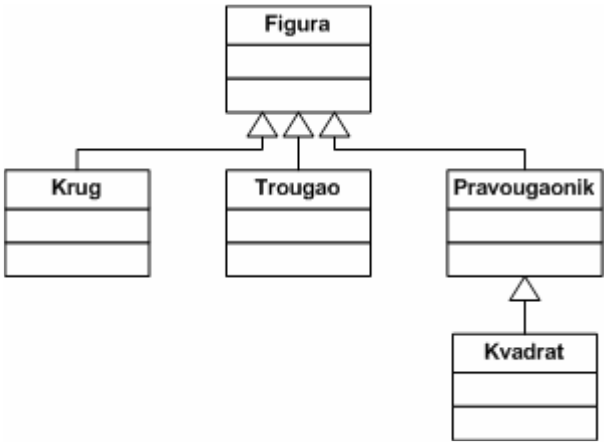
```
// Datoteka: figure.hpp
#ifndef FIGURE_DEF
#define FIGURE_DEF
#include <math.h>
class Figura {
    public:
        virtual double getO() const=0;
        virtual double getP() const=0;
};

class Trougao : public Figura {
    private:
        double a, b, c;
    public:
        Trougao(double aa=3, double bb=4, double cc=5) {
            a=aa; b=bb; c=cc;
        }
        Trougao(const Trougao &t) { a=t.a; b=t.b; c=t.c; }
        double getO() const { return a+b+c; }
        double getP() const {
            double s=(a+b+c)/2;
            return sqrt(s*(s-a)*(s-b)*(s-c));
        }
};

class Pravougaonik : public Figura {
    private:
        double a, b;
    public:
        Pravougaonik(double aa=1, double bb=2) { a=aa; b=bb; }
        Pravougaonik(const Pravougaonik &p) { a=p.a; b=p.b; }
        double getO() const { return 2*a + 2*b; }
        double getP() const { return a*b; }
};
#endif

// Datoteka: main.cpp
#include "figure.hpp"
#include <iostream>
using namespace std;

void printFigura(const Figura &f) {
    cout<<"Obim: "<<f.getO()<<endl;
    cout<<"Povrsina: "<<f.getP()<<endl;
}
```

```
int main() {
    Trougao t1, t2(2,5,5);
    Pravougaonik p1, p2(5,6);
    printFigura(t1);
    printFigura(t2);
    printFigura(p1);
    printFigura(p2);
    return 0;
}
```

6.5 Zajednički članovi klase

- **Zajednički član klase** se dobija tako što se ispred njegove deklaracije navede rezervisana reč **`static`**.
- Zajedničko polje je zajedničko za sve objekte date klase. To znači da u svakom trenutku vrednost tog polja je jednaka za sve objekte date klase.

Zadatak 6.5.1
Napisati apstraktnu klasu `Figura` i realizovati hijerarhiju klasa sa slike 12.



Slika 12.

```
// Datoteka: figure.hpp
#ifndef FIGURE_DEF
#define FIGURE_DEF
#include <math.h>

class Figura {
    private:
        static int count;
        static int id;
```

```
    public:
        Figura() { count++; }
        ~Figura() { count--; }
        virtual int getCount() const { return count; }
        virtual int getId() const { return id; }
        virtual double getO() const=0;
        virtual double getP() const=0;
};


class Krug : public Figura {
    private:
        double r;
        static int count;
        static int id;
    public:
        Krug(double rr=1) { r=rr; count++; }
        ~Krug() { count--; }
        double getR() const { return r; }
        int getCount() const { return count; }
        int getId() const { return id; }
        double getO() const { return 2*r*M_PI; }
        double getP() const { return r*r*M_PI; }
};


class Trougao : public Figura {
    private:
        double a, b, c;
        static int count;
        static int id;
    public:
        Trougao(double aa=3, double bb=4, double cc=5) {
            a=aa; b=bb; c=cc; count++;
        }
        Trougao(const Trougao &t) { a=t.a; b=t.b; c=t.c; count++; }
        ~Trougao() { count--; }
        double getA() const { return a; }
        double getB() const { return b; }
        double getC() const { return c; }
        int getCount() const { return count; }
        int getId() const { return id; }
        double getO() const { return a+b+c; }
        double getP() const {
            double s=(a+b+c)/2;
            return sqrt(s*(s-a)*(s-b)*(s-c));
        }
};


class Pravougaonik : public Figura {
    private:
        double a, b;
        static int count;
        static int id;
```

```
    public:
        Pravougaonik(double aa=1, double bb=2) {
            a=aa; b=bb; count++;
        }
        Pravougaonik(const Pravougaonik &p) {
            a=p.a; b=p.b; count++;
        }
        ~Pravougaonik() { count--; }
        double getA() const { return a; }
        double getB() const { return b; }
        int getCount() const { return count; }
        int getId() const { return id; }
        double getO() const { return 2*a + 2*b; }
        double getP() const { return a*b; }
};

class Kvadrat : public Pravougaonik {
    private:
        static int count;
        static int id;
    public:
        Kvadrat(double aa=1) : Pravougaonik(aa,aa) {
            count++;
        }
        Kvadrat(const Kvadrat &k) : Pravougaonik((Pravougaonik)k) {
            count++;
        }
        ~Kvadrat() { count--; }
        int getCount() const { return count; }
        int getId() const { return id; }
};

#endif
```

```
// Datoteka: figure.cpp
#include "figure.hpp"
int Figura::id=0;
int Figura::count=0;
int Krug::id=1;
int Krug::count=0;
int Trougao::id=2;
int Trougao::count=0;
int Pravougaonik::id=3;
int Pravougaonik::count=0;
int Kvadrat::id=4;
int Kvadrat::count=0;
```

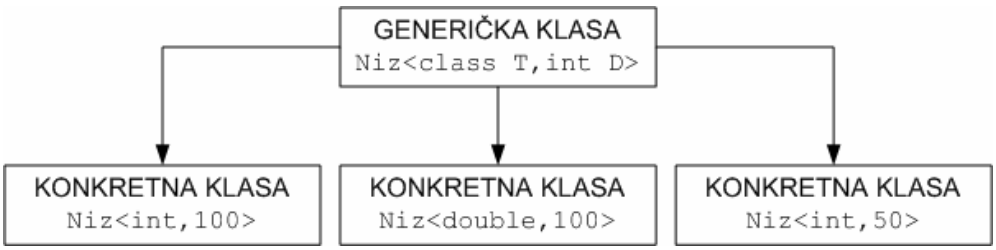
```
// Datoteka: main.cpp
#include "figure.hpp"
#include <iostream>
using namespace std;
```

```
void printFigura(const Figura &f) {
    cout<<"Vrsta figure: ";
    switch(f.getId()){
        case 0 : cout<<"FIGURA"<<endl; break;
        case 1 : cout<<"KRUG"<<endl; break;
        case 2 : cout<<"TROUGAO"<<endl; break;
        case 3 : cout<<"PRAVOUGAONIK"<<endl; break;
        case 4 : cout<<"KVADRAT"<<endl; break;
    }
    cout<<"Obim: "<<f.getO()<<endl;
    cout<<"Povrsina: "<<f.getP()<<endl;
    cout<<"Broj aktuelnih objekata: "<<f.getCount()<<endl;
}

int main() {
    Krug k1, k2(4);
    Trougao t1, t2(2,5,5);
    Pravougaonik p1, p2(5,6);
    Kvadrat kv1, kv2(5);
    printFigura(k1);
    printFigura(k2);
    printFigura(t1);
    printFigura(t2);
    printFigura(p1);
    printFigura(p2);
    printFigura(kv1);
    printFigura(kv2);
    return 0;
}
```

- Svaka od klasa sadrži dva **`static`** polja: `count` i `id`. Polje `count` *broji* aktuelne objekte u programu, tj. svaki poziv konstruktora će polje `count` uvećati, dok će svaki poziv destruktora smanjiti za jedan. Polje `id` jeste identifikator na osnovu kojeg se može zaključiti o kojoj klasi je reč. Ukoliko je polje `id` jednako 0 radi se o klasi `Figura`, ukoliko je polje `id` jednako 1 radi se o klasi `Krug`, ukoliko je polje `id` jednako 2 radi se o klasi `Trougao`, ukoliko je polje `id` jednako 3 radi se o klasi `Pravougaonik` i najzad, ukoliko je polje `id` jednako 4 radi se o klasi `Kvadrat`.

7. GENERIČKE KLASE

- Razmotrimo jedan primer – posmatrajmo dva niza od 100 elemenata. Neka su elementi prvog niza tipa `int`, a drugog niza tipa `double`. Operacije koje definišemo za prvi niz mogli bismo iskoristiti i za drugi niz, s tim da uvek uzimamo u obzir to da ovde radimo sa elementima tipa `double`. Na primer, operator za indeksiranje za slučaj prvog niza bi vratio vrednost elementa na *i*-toj poziciji, pri čemu taj element je tipa `int`. Slično, taj isti operator za slučaj drugog niza bi vratio vrednost elementa na *i*-toj poziciji, pri čemu taj element je tipa `double`. Međutim, ukoliko bi nam trebala oba niza mi bismo morali da napišemo obe klase, bez obzira što su veoma slične.
- Iz prethodnog primera zaključujemo da bi u ovakvim situacijama bilo korisno imati mehanizam pomoću kojeg možemo napisati klasu koja modeluje niz čiji su elementi nekog tipa `T`, a kasnije ukoliko želimo niz celih brojeva tada da možemo reći da je tip `T` ustvari tip `int`, odnosno ako želimo niz realnih brojeva tada da možemo reći da je tip `T` ustvari tip `double`.
- U programskom jeziku C++ postoji mehamizam pomoću kojeg možemo napisati **šablon** (eng. *template*) kojim opisujemo opšti slučaj (bez upotrebe konkretnih tipova).
- Klasa koja je napisana pomoću šablona naziva se **generička klasa**. Kada se šablonu navedu konkretni tipovi dobijamo konkretne klase.
- Na slici 13. ilustrovana je generička klasa `Niz<class T, int D>`, koja opisuje opšti slučaj – niz koji ima elemente tipa `T` i dimenzije je `D`. Na primer, jedna konkretizacija tog opšteg niza je niz celih brojeva dimenzije 100, a to je na slici konkretna klasa `Niz<int,100>`.



Slika 13.

7.1 Primeri generičkih klasa

- U ovom poglavlju razmotrićemo neke od osnovnih generičkih klasa, kao što su `Pair<class T1,class T2>`, `Niz<class T,int D>` i `List<class T>`. Takođe, u ovom poglavlju razmotrićemo upotrebu napisanih generičkih klasa.

Zadatak 7.1.1

Napisati generičku klasu `Pair<class T1, class T2>`.

```
// Datoteka: pair.hpp
#ifndef PAIR_DEF
#define PAIR_DEF

#include <iostream>
using namespace std;

template<class T1, class T2>
class Pair {
    private:
        T1 first;
        T2 second;
    public:
        Pair(const T1&, const T2&);
        T1 getFirst() const;
        T2 getSecond() const;
        void setFirst(const T1&);
        void setSecond(const T2&);
        Pair<T1,T2>& operator=(const Pair<T1,T2>&);
        void printPair() const;
};



template<class T1, class T2>
Pair<T1,T2>::Pair(const T1 &f, const T2 &s) : first(f), second(s) {}

template<class T1, class T2>
T1 Pair<T1,T2>::getFirst() const { return first; }

template<class T1, class T2>
T2 Pair<T1,T2>::getSecond() const { return second; }

template<class T1, class T2>
void Pair<T1,T2>::setFirst(const T1 &f) { first=f; }

template<class T1, class T2>
void Pair<T1,T2>::setSecond(const T2 &s) { second=s; }

template<class T1, class T2>
Pair<T1,T2>& Pair<T1,T2>::operator=(const Pair<T1,T2> &p) {
    first=p.first;
    second=p.second;
    return *this;
}

template<class T1, class T2>
void Pair<T1,T2>::printPair() const {
    cout<<"( "<<first<<", "<<second<<" )"<<endl;
}
```

```
template<class T1, class T2>
bool operator==(const Pair<T1,T2> &p1, const Pair<T1,T2> &p2) {
    if((p1.getFirst()==p2.getFirst()) &&
       (p1.getSecond()==p2.getSecond()))
        return true;
    else
        return false;
}

#endif



// Datoteka: main.cpp
#include "pair.hpp"

typedef Pair<int,int> IIPair;
typedef Pair<int,double> IDPair;
typedef Pair<double,double> DDPair;

int main() {
    Pair<int,int> x1(1,2);
    IDPair x2(1,2.3), x3(5,7.8);
    x1.printPair();
    x2.printPair();
    x3.printPair();
    x3=x2;
    x2.printPair();
    x3.printPair();
    if(x2==x3)
        cout<<"x2 i x3 su jednaki"<<endl;
    else
        cout<<"x2 i x3 nisu jednaki"<<endl;
    return 0;
}
```

Zadatak 7.1.2

Za klasu `DinString` iz zadatka 4.3.3 testirati konkretizaciju

```
Pair<DinString, DinString>
```

```
// Datoteka: main.cpp
#include "pair.hpp"
#include "dinstring.hpp"

int main() {
    DinString ds1("Petar");
    DinString ds2("Petrovic");
    Pair<DinString,DinString> p1(ds1,ds2);
    p1.printPair();
```

```
    Pair<DinString,DinString> p2("Jovan","Jovanovic");
    p2.printPair();
    p2=p1;
    p2.printPair();
    if(p1==p2)
        cout<<"p1 i p2 su jednaki"<<endl;
    else
        cout<<"p1 i p2 nisu jednaki"<<endl;
    return 0;
}
```

Zadatak 7.1.3

Napisati generičku klasu `Niz<class T, int D>`.

```
// Datoteka: niz.hpp
#ifndef NIZ_DEF
#define NIZ_DEF

#include <iostream>
using namespace std;

template <class T, int D>
class Niz {
    private:
        T el[D];
        int brEl;
    public:
        Niz() { brEl=0; }
        ~Niz() {}
        int getBrEl() const { return brEl; }
        T operator[](int i) const { return el[i]; }
        T& operator[](int i) { return el[i]; }
        Niz<T,D>& operator=(const Niz<T,D>&);
        void printNiz() const;
        bool insertNiz(const T&);
};



template <class T, int D>
Niz<T,D>& Niz<T,D>::operator=(const Niz<T,D> &rn) {
    for(brEl=0; brEl<rn.brEl; brEl++)
        el[brEl]=rn[brEl];
    return *this;
}

template <class T, int D>
void Niz<T,D>::printNiz() const {
    cout<<"( ";
    for(int i=0; i<brEl-1; i++)
        cout<<el[i]<<", ";
    cout<<el[brEl-1]<<" )"<<endl;
}
```

```
template <class T, int D>
bool Niz<T,D>::insertNiz(const T &t) {
    if(brEl<D) {
        el[brEl]=t;
        brEl++;
        return true;
    }
    else
        return false;
}

template <class T, int D>
bool operator==(const Niz<T,D> &rn1, const Niz<T,D> &rn2) {
    if(rn1.getBrEl()!=rn2.getBrEl())
        return false;
    for(int i=0; i<rn1.getBrEl(); i++)
        if(rn1[i]!=rn2[i])
            return false;
    return true;
}



template <class T, int D>
bool operator!=(const Niz<T,D> &rn1, const Niz<T,D> &rn2) {
    if(rn1.getBrEl()!=rn2.getBrEl())
        return true;
    for(int i=0; i<rn1.getBrEl(); i++)
        if(rn1[i]!=rn2[i])
            return true;
    return false;
}

#endif


// Datoteka: main.cpp
#include "niz.hpp"

int main() {
    Niz<int,10> iNiz1, iNiz2;
    iNiz1.insertNiz(1);
    iNiz1.insertNiz(2);
    iNiz1.insertNiz(3);
    iNiz1.insertNiz(4);
    iNiz1.insertNiz(5);
    iNiz1.insertNiz(6);
    iNiz1.printNiz();
    iNiz2=iNiz1;
    if(iNiz1==iNiz2)
        cout<<"iNiz1 i iNiz2 su jednaki"<<endl;
    else
        cout<<"iNiz1 i iNiz2 nisu jednaki"<<endl;
    return 0;
}
```

Zadatak 7.1.4
Za klasu `DinString` iz zadatka 4.3.3 testirati konkretizaciju

```
Niz<DinString, 10>
```

```
// Datoteka: main.cpp
//#include "niz.hpp"
#include "dinstring.hpp"

int main() {
    Niz<DinString,10> dsNiz1, dsNiz2;
    dsNiz1.insertNiz("Jedan");
    dsNiz1.insertNiz("Dva");
    dsNiz1.insertNiz("Tri");
    dsNiz1.insertNiz("Cetiri");
    dsNiz1.insertNiz("Pet");
    dsNiz1.insertNiz("Sest");
    dsNiz1.printNiz();
    dsNiz2=dsNiz1;
    if(dsNiz1==dsNiz2)
        cout<<"dsNiz1 i dsNiz2 su jednaki"<<endl;
    else
        cout<<"dsNiz1 i dsNiz2 nisu jednaki"<<endl;
    return 0;
}
```

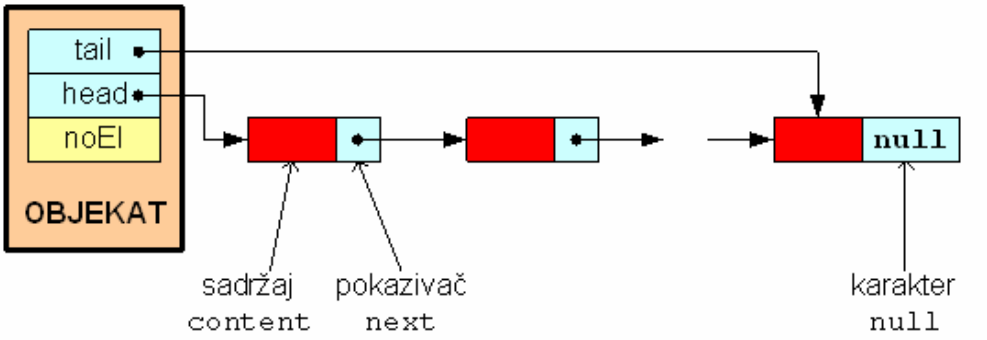
Zadatak 7.1.5
Napisati generičku klasu `List`, koja modeluje jednostrukospregnutu listu čiji su elementi tipa `T`, a zatim za klasu `Complex` iz zadatka 4.3.1 testirati konkretizaciju

```
List<Complex>
```

```
// Datoteka: list.hpp
#ifndef LIST_DEF
#define LIST_DEF
#include <stdlib.h>
#include <iostream>
using namespace std;


template <class T>
class List{
      private:
            struct listEl{
                  T content;
                  struct listEl *next;
            };
            listEl *head;
            listEl *tail;
            int noEl;
```

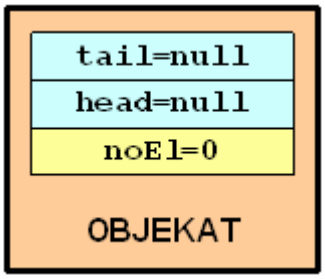
```
        public:
                List(){
                        head=tail=NULL;
                        noEl=0;
                }
                List(const List<T>&);
                List<T>& operator=(const List<T>&);
                virtual ~List();
                int size() const { return noEl; }
                bool empty() const { return head==NULL?1:0; }
                bool add(int, const T&);
                bool remove(int);
                bool read(int, T&)const;
                void clear();
};
```

- U klasi `List` je definisan izgled strukture `listEl`, tako da će svaki čvor liste sadržati `content` (sadržaj tipa `T`) i pokazivač `next` (pokazivač na sledeći čvor liste). Objekat klase `List` sadrži sledeća polja: dva pokazivača (`head` i `tail`) i `noEl` (tipa `int` koji čuva tekući broj čvorova liste). Pokazivač `head` pokazuje na prvi čvor liste, a pokazivač `tail` pokazuje na poslednji čvor liste. Pokazivač `tail` nije obavezan, ali ukoliko postoji može da poveća efikasnost rada sa listom. Slika 14. prikazuje organizaciju jednostrukospregnute liste.



Slika 14.

- Konstruktor bez parametara postavlja inicijalne vrednosti polja `head=tail=null` i `noEl=0`. Izgled objekta kojeg je kreirao konstruktor bez parametara prikazuje slika 15.



Slika 15.

```
template <class T>
ostream& operator<<(ostream & out, const List<T> &rl) {
	out<<endl;
	out<<"--------"<<endl;
	for(int i=1; i<=rl.size(); i++){
		if(i!=1) out<<", ";
		T res;
		rl.read(i,res);
		out<<res;
	}
	out<<endl<<"--------"<<endl;
	return out;
}
```

- Operator za ispis na ekran `<<` realizujemo pomoću slobodne funkcije (koja nije **`friend`** funkcija) i zbog toga se mora voditi računa da se direktno ne pristupi nekom `private` članu. Za čitanje sadržaja čvora na *i*-toj poziciji koristi se metoda `read()` (koja je `public`). U metodu `read()` prosleđuju se sledeći stvarni argumenti: pozicija `i` (tipa `int`) i po referenci objekat `res` klase `T`. U metodi `read()` se pronalazi *i*-ti čvor liste i njegov sadržaj tj. `content` (tipa `T`) se dodeljuje objektu `res`, jer je objekat `res` u metodu prosleđen po referenci i na taj način taj sadržaj biva vidljiv po završetku izvršavanja metode `read()`.

```
template <class T>
List<T>::List(const List<T> &rl) {
	head=NULL; tail=NULL;
	noEl=0;
	for(int i=1; i<=rl.noEl; i++){
		T res;
		if(rl.read(i,res))
			add(i,res);
	}
}

template <class T>
List<T>& List<T>::operator=(const List<T> &rl) {
	if(this!=&rl) {
		clear();
		head=NULL; tail=NULL;
		noEl=0;
		for(int i=1; i<=rl.noEl; i++){
			T res;
			if(rl.read(i,res))
				add(i,res);
		}
	}
	return *this;
}
```

- Konstruktor kopije i operator dodele u sebi sadrže *for*-ciklus u kojem se metodom `read()` smešta sadržaj *i*-tog čvora objekta `rl`, u objekat `res`, a zatim se sadržaj objekta `res` prosleđuje u metodu `add()`. Na taj način se pravi kopija objekta `rl`.

```
template <class T>
List<T>::~List() {
	while(!empty())
		remove(1);
}

template <class T>
bool List<T>::add(int n, const T &newContent) {
	if(n<1 || n>noEl+1)
		return false;
	else {
		listEl *newEl=new listEl;
		if(newEl==NULL)
			return false;
		else {
			newEl->content=newContent;
			if(n==1) {
				newEl->next=head;
				head=newEl;
			} else if(n==noEl+1) {
				newEl->next=NULL;
				tail->next=newEl;
			} else {
				listEl *temp=head;
				for(int i=2; i<n; i++)
					temp=temp->next;
				newEl->next=temp->next;
				temp->next=newEl;
			}
			noEl++;
			if(newEl->next==NULL)
				tail=newEl;
			return true;
		}
	}
}

template <class T>
bool List<T>::remove(int n){
	if(n<1 || n>noEl)
		return false;
	else {
		if(n==1) {
			listEl *del=head;
			head=head->next;
			if(tail==del)
				tail=NULL;
			delete del;
			noEl--;
```

```
		} else {
			listEl *temp=head;
			for(int i=2;i<n;i++)
				temp=temp->next;
			listEl *del=temp->next;
			temp->next=del->next;
			if(tail==del)
				tail=temp;
			delete del;
			noEl--;
		}
		return true;
	}
}

template <class T>
bool List<T>::read(int n,T& retVal) const {
	if(n<1 || n>noEl)
		return false;
	else {
		if(n==1)
			retVal=head->content;
		else if(n==noEl)
			retVal=tail->content;
		else {
			listEl *temp=head;
			for(int i=1; i<n; i++)
				temp=temp->next;
			retVal=temp->content;
		}
		return true;
	}
}

template <class T>
void List<T>::clear() {
	while(!empty())
		remove(1);
}
#endif

// Datoteka: main.cpp
#include "list.hpp"
#include "complex.hpp"

typedef List<Complex> ComplexList;

int main() {
    List<int> iList;
    iList.add(1,5);
    iList.add(2,7);
    cout<<iList<<endl;
    Complex z1,z2(3,5),z3(0,-6);
    ComplexList cList;
    cList.add(1,z1);
```

```
    cList.add(1,z2);
    cList.add(2,z3);
    cout<<cList<<endl;
    cList.remove(3);
    cout<<cList<<endl;
    cList.read(1,z1);
    cout<<z1<<endl;
    return 0;
}
```

Zadatak 7.1.6

Iz klase `List` izvesti klasu `LinkedQueue` (red ili FIFO memorija).

```
// Datoteka: queue_lnk.hpp
#ifndef QUEUE_DEF
#define QUEUE_DEF

#include "list.hpp"

template <class T>
class LinkedQueue;

template <class T>
void printOut(const LinkedQueue<T> &);

template <class T>
class LinkedQueue : private List <T> {
public:
       LinkedQueue(){};
       bool readFromQueue(T&)const;
       void removeFromQueue() { remove(1); }
       void addToQueue(const T &El){ add(size()+1,El); }
       bool empty() const { return List<T>::empty(); }
       int size() const { return List<T>::size(); }
       friend void printOut<>(const LinkedQueue<T>&);
       virtual ~LinkedQueue(){}
};


template <class T>
void printOut(const LinkedQueue<T> &rlq){
       cout<<endl;
       cout<<"\tVelicina reda: "<<rlq.size()<<endl;
       cout<<"\tSadrzaj reda je: ";
       T retVal;
       for(int i=1;i<=rlq.size();i++){
              if(i>1) cout<<", ";
              rlq.read(i,retVal);
              cout<<retVal;
       }
       cout<<endl<<endl;
}
```

```
template <class T>
bool LinkedQueue<T>::readFromQueue(T& retVal) const {
	return List<T>::read(1,retVal);
}
#endif


// Datoteka: main.cpp
#include "queue_lnk.hpp"

typedef LinkedQueue<int> IntQueue;

int main(){
	IntQueue a;
	a.addToQueue(1);
	a.addToQueue(2);
	a.addToQueue(3);
	cout<<"Queue: ";
	printOut(a);
	cout<<endl;
	cout<<"--------------------"<<endl;
	cout<<"Obavlja se uklanjanje iz reda"<<endl;
	a.removeFromQueue();
	cout<<"Queue: ";
	printOut(a);
	cout<<endl;
	cout<<"--------------------"<<endl;
	cout<<"Obavlja se uklanjanje iz reda"<<endl;
	a.removeFromQueue();
	cout<<"Queue: ";
	printOut(a);
	cout<<endl;
	cout<<"--------------------"<<endl;
	cout<<"Dodavanje u red broja 45"<<endl;
	a.addToQueue(45);
	cout<<"Queue: ";
	printOut(a);
	cout<<endl;
	cout<<"--------------------"<<endl;
	cout<<"Pokusaj samo citanja (ne i uklanjanja) iz reda"<<endl;
	int ret;
	if(a.readFromQueue(ret))
		cout<<"Obavljeno citanje iz reda: "<<ret<<endl;
	else
		cout<<"Citanje nije obavljeno - Prazan red!!!"<<endl;
	cout<<"--------------------"<<endl;
	cout<<"Obavlja se uklanjanje iz reda"<<endl;
	a.removeFromQueue();
	cout<<"Queue: ";
	printOut(a);
	cout<<endl;
	cout<<"--------------------"<<endl;
	cout<<"Obavlja se uklanjanje iz reda"<<endl;
	a.removeFromQueue();
```

```
        cout<<"Queue: ";
        printOut(a);
        cout<<endl;
        cout<<"--------------------"<<endl;
        cout<<"Pokusaj samo citanja (ne i uklanjanja) iz reda"<<endl;
        if(a.readFromQueue(ret))
                cout<<"Obavljeno citanje iz reda: "<<ret<<endl;
        else
                cout<<"Citanje nije obavljeno - Prazan red!!!"<<endl;
        cout<<"--------------------"<<endl;
        cout<<"Dodavanje u red broja 100"<<endl;
        a.addToQueue(100);
        cout<<"Queue: ";
        printOut(a);
        return 0;
}
```

Zadatak 7.1.7

Data je klasa `DinString` iz zadatka 4.3.3 i generička klasa `List` iz zadatka 7.1.5. Napisati klasu `Student` koja sadrži polja: `brojIndeksa` (`int`), `ime` (`DinString`), `prezime` (`DinString`) i `ocena` (`int`) i klasu `SpisakStudenata` koja sadrži polja: `nazivPredmeta` (`DinString`), `datumPolaganja` (`DinString`) i `lista` (`List<Student>`).

```
// Datoteka: student.hpp
#ifndef STUDENT_DEF
#define STUDENT_DEF

#include "dinstring.hpp"

class Student {
    protected:
        int brojIndeksa;
        DinString ime;
        DinString prezime;
        int ocena;

    public:
        Student(int i=0, const char *s1="", const char *s2="", int
oc=5) : brojIndeksa(i), ime(s1), prezime(s2), ocena(oc) {}
        Student(int i, const DinString &ds1, const DinString &ds2,
int oc) : brojIndeksa(i), ime(ds1), prezime(ds2), ocena(oc) {}
        Student(const Student &s) : brojIndeksa(s.brojIndeksa),
ime(s.ime), prezime(s.prezime), ocena(s.ocena) {}
        ~Student() {}
        void setBrojIndeksa(int bri) { brojIndeksa=bri; }
        void setIme(const DinString &ds) { ime=ds; }
        void setPrezime(const DinString &ds) { prezime=ds; }
        void setOcena(int x) { ocena=x; }
        int getBrojIndeksa() const { return brojIndeksa; }
        int getOcena() const { return ocena; }
```

```
        Student& operator=(const Student&);
        friend ostream& operator<<(ostream&, const Student&);
};
#endif


// Datoteka: student.cpp
#include "student.hpp"

Student& Student::operator=(const Student &s) {
    brojIndeksa=s.brojIndeksa;
    ime=s.ime;
    prezime=s.prezime;
    ocena=s.ocena;
    return *this;
}

ostream& operator<<(ostream &out, const Student &s) {
    out<<"   "<<s.brojIndeksa;
    out<<"   "<<s.ime;
    out<<"   "<<s.prezime;
    out<<"   "<<s.ocena;
    return out;
}


// Datoteka: spisak.hpp
#ifndef SPISAK_STUDENATA_DEF
#define SPISAK_STUDENATA_DEF
#include "student.hpp"
#include "list.hpp"

class SpisakStudenata {
    private:
        DinString nazivPredmeta;
        DinString datumPolaganja;
        List<Student> lista;
    public:
        SpisakStudenata() {}
        void setNaziv(const DinString &ds) { nazivPredmeta=ds; }
        void setDatumPolaganja(const DinString &ds) {
            datumPolaganja=ds;
        }
        void insertStudent();
        void sortPoOceni();
        void sortPoIndeksu();
        void printSpisak() const;
};

#endif



// Datoteka: spisak.cpp
#include "spisak.hpp"
```

```
void SpisakStudenata::insertStudent() {
    Student st;
    int x;
    char s[100];
    cout<<"Unesite broj indeksa: ";
    cin>>x;
    st.setBrojIndeksa(x);
    cout<<"Unesite ime studenta: ";
    cin>>s;
    st.setIme(s);
    cout<<"Unesite prezime studenta: ";
    cin>>s;
    st.setPrezime(s);
    cout<<"Unesite ocenu: ";
    cin>>x;
    st.setOcena(x);
    cout<<"******************************************************";
    cout<<endl;
    cout<<st<<endl;
    cout<<"******************************************************";
    cout<<endl;
    lista.add(1,st);
}

void SpisakStudenata::sortPoOceni() {
    Student s1, s2;
    for(int i=1; i<=lista.size()-1; i++)
        for(int j=i+1; j<=lista.size(); j++) {
            lista.read(i,s1);
            lista.read(j,s2);
            if(s1.getOcena()<s2.getOcena()) {
                lista.remove(i);
                lista.add(i,s2);
                lista.remove(j);
                lista.add(j,s1);
            }
        }
}


 void SpisakStudenata::sortPoIndeksu() {
    Student s1, s2;
    for(int i=1; i<=lista.size()-1; i++)
        for(int j=i+1; j<=lista.size(); j++) {
            lista.read(i,s1);
            lista.read(j,s2);
            if(s1.getBrojIndeksa()>s2.getBrojIndeksa()) {
                lista.remove(i);
                lista.add(i,s2);
                lista.remove(j);
                lista.add(j,s1);
            }
        }
}
```

```
void SpisakStudenata::printSpisak() const {
    cout<<"Datum: "<<datumPolaganja<<endl;
    cout<<"Predmet: "<<nazivPredmeta<<endl;
    cout<<" ----------------- REZULTATI ISPITA ----------------- ";
    cout<<endl;
    Student st;
    for(int i=1; i<=lista.size(); i++) {
        lista.read(i,st);
        cout<<st<<endl;
    }
    cout<<" ------------------------------------------------------ ";
    cout<<endl;
}



// Datoteka: main.cpp
//#include "spisak.hpp"

int main() {
    char odg='d';
    char s[100];
    SpisakStudenata spisak;
    cout<<"\n\n\n\n\n";
    cout<<"Unesite datum polaganja ispita: ";
    cin>>s;
    spisak.setDatumPolaganja(s);
    cout<<"Unesite naziv ispita: ";
    cin>>s;
    spisak.setNaziv(s);
    cout<<"Unesite podatke o studentima... "<<endl;
    cout<<endl;
    while((odg=='d')||(odg=='D')) {
        spisak.insertStudent();
        cout<<"Da zelite jos da unosite podatke?[D/N]";
        cin>>odg;
    }

    cout<<"Da li da spisak bude sortiran po ocenama?[D/N]";
    cin>>odg;
    if((odg=='d')||(odg=='D'))
       spisak.sortPoOceni();
    else
    {
       cout<<"Da li da spisak bude sortiran po broju indeksa?[D/N]";
       cin>>odg;
       if((odg=='d')||(odg=='D'))
           spisak.sortPoIndeksu();
    }
    cout<<"\n\n\n\n\n\n\n\n\n\n";
    spisak.printSpisak();
    return 0;
}
```

- Kad pokrenemo program i unesemo sledeće podatke:

Datum polaganja ispita: 15.02.2011.
Naziv ispita: OOP

12454 Petar Petrovic 7
12789 Jovan Jovanovic 9
12656 Stevan Stevanovic 6
12854 Stojan Stojanovic 6
11789 Vasa Vasic 10
10289 Marko Markovic 7
12987 Lazar Lazarevic 9

i ukoliko izaberemo opciju da spisak bude sortiran po ocenama, dobijamo sledeći ispis na ekranu:

```
Datum: 15.02.2011.
Predmet: OOP
 ----------------- REZULTATI ISPITA -----------------
   11789   Vasa   Vasic   10
   12987   Lazar   Lazarevic   9
   12789   Jovan   Jovanovic   9
   10289   Marko   Markovic   7
   12454   Petar   Petrovic   7
   12854   Stojan   Stojanovic   6
   12656   Stevan   Stevanovic   6
 ------------------------------------------------------
```

Ukoliko pak izaberemo opciju da spisak bude sortiran po broju indeksa, dobijamo sledeći ispis na ekranu:

```
Datum: 15.02.2011.
Predmet: OOP
 ----------------- REZULTATI ISPITA -----------------
   10289   Marko   Markovic   7
   11789   Vasa   Vasic   10
   12454   Petar   Petrovic   7
   12656   Stevan   Stevanovic   6
   12789   Jovan   Jovanovic   9
   12854   Stojan   Stojanovic   6
   12987   Lazar   Lazarevic   9
 ------------------------------------------------------
```

8. PREVENCIJA OTKAZA

- Rezultat greške pojavljuje se u programu kao defekt (eng. *fault*). Ako se u toku izvršavanja programa naiđe na defekt, dolazi do otkaza (eng. *failure*). Otkaz može da se manifestuje (prekid programa, pad operativnog sistema itd.), ali i ne mora.
- Postoje tri načina za predupređivanje otkaza, od kojih su prva dva karakteristična za procedurno programiranje:
  - prekid izvršavanja programa putem standardnih rutina (`exit` ili `abort` u programskom jeziku C/C++),
  - upravljanje preko izlazne vrednosti potprograma,
  - rukovanje izuzecima (eng. *exception handling*)
- Zajednička karakteristika sva tri načina jeste **prevencija otkaza**, odnosno sprečavanje izvršavanja naredbi koje izazivaju otkaz.

8.1 Rukovanje izuzecima

- **Izuzetak** je namerno izazvan događaj čija je svrha predupređivanje otkaza i koji ostavlja program u zatečenom stanju.
- Prijavljivanje izuzetka se vrši naredbom `throw`, a može se naći u dve varijante:

```
throw izraz;
throw poziv_konstruktora;
```

- Prihvatanje i obrada izuzetaka se vrše u *try-catch* bloku:

```
try {
      naredbe
}
catch(kvaziparametar1) {
      naredbe
}
catch(kvaziparametar2) {
      naredbe
}
. . .
catch(kvaziparametarn) {
      naredbe
}
```

Zadatak 8.1.1
Napisati klasu `Stack` koja modeluje celobrojni sekvencijalni stek. U zadatku koristiti mehanizam rukovanja izuzecima.

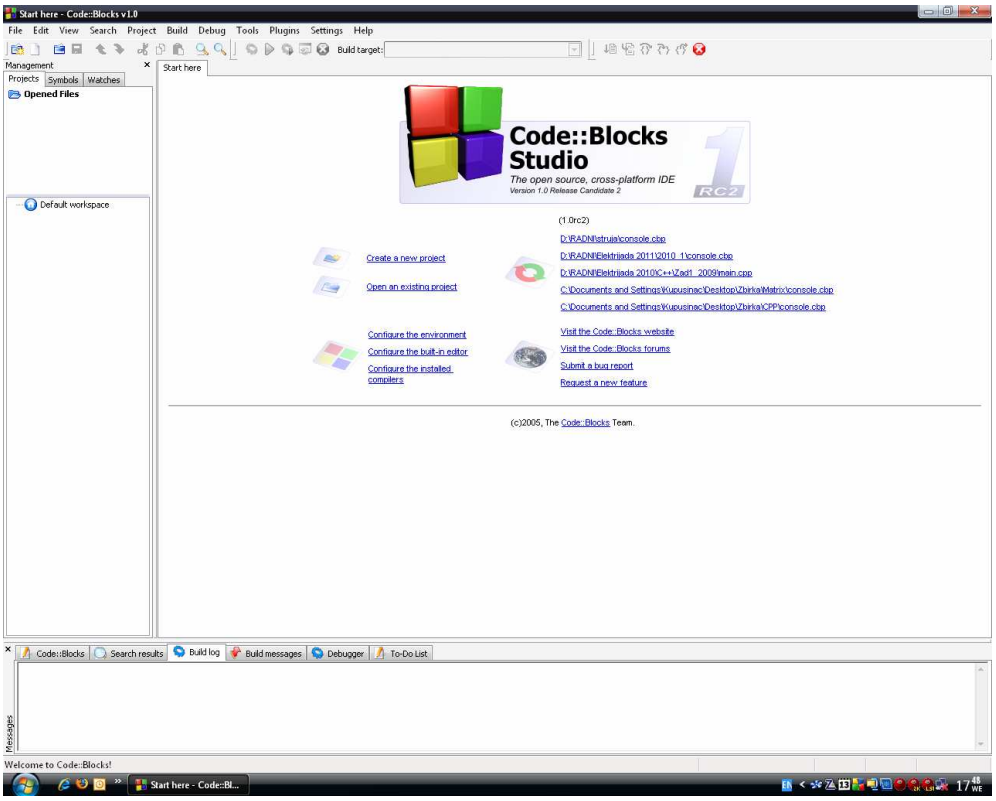
```
// Datoteka: stack.hpp
#ifndef INTSTACK_DEF
#define INTSTACK_DEF

#include <iostream>
using namespace std;

#define CAPACITY 100

enum ErrorType {UNDERFLOW=1, OVERFLOW};

class Stack {
    private:
        int t;
        int s[CAPACITY];
    public:
        Stack() { t=-1; }
        bool empty() const { return t<0; }
        bool full() const { return t==CAPACITY-1; }
        int top() const;
        void pop();
        void push(int el);
        friend ostream& operator<<(ostream&, const Stack&);
};

#endif



// Datoteka: stack.cpp
#include "stack.hpp"

int Stack::top() const {
    if(t<0)
        throw UNDERFLOW;
    return s[t];
}

void Stack::pop() {
    if(t<0)
        throw UNDERFLOW;
    t--;
}

void Stack::push(int el) {
    if(t==CAPACITY-1)
        throw OVERFLOW;
    s[++t]=el;
}
```

```
ostream& operator<<(ostream &out, const Stack &rs) {
    out<<endl<<"----------------------"<<endl;
    if(rs.t<0)
        out<<"Empty stack!";
    else {
        out<<rs.s[rs.t]<<" <-- TOP"<<endl;
        for(int i=rs.t-1; i>=0; i--)
            out<<rs.s[i]<<endl;
    }
    return out;
}
```

```
// Datoteka: main.cpp
#include "stack.hpp"

int main() {
    Stack s;
    try {
        //s.pop(); // Stack underflow.
        s.push(1);
        s.push(2);
        s.push(3);
        cout<<s<<endl;
        s.pop();
        s.pop();
        s.pop();
        //s.top(); // Stack underflow.
        for(int i=0; i<CAPACITY; i++)
            s.push(i);
        cout<<s<<endl;
        //s.push(100); // Stack overflow.
    }
    catch(ErrorType er) {
        switch(er) {
            case UNDERFLOW : cout<<"ERROR: Stack underflow."<<endl;
                             break;
            case OVERFLOW : cout<<"ERROR: Stack overflow."<<endl;
        }
    }
    return 0;
}
```

A. PROGRAMSKO OKRUŽENJE `Code::Blocks`

- Instalacija programskog okruženja `Code::Blocks` se može preuzeti sa sajta:
  - http://www.codeblocks.org/downloads/binaries
- Uputstvo za rad sa programskim okruženjem `Code::Blocks` se može preuzeti sa sajta:
  - http://www.codeblocks.org/user-manual
- Pokretanjem programskog okruženja `Code::Blocks` dobijamo izgled ekrana kao što to prikazuje slika 16.



Slika 16.

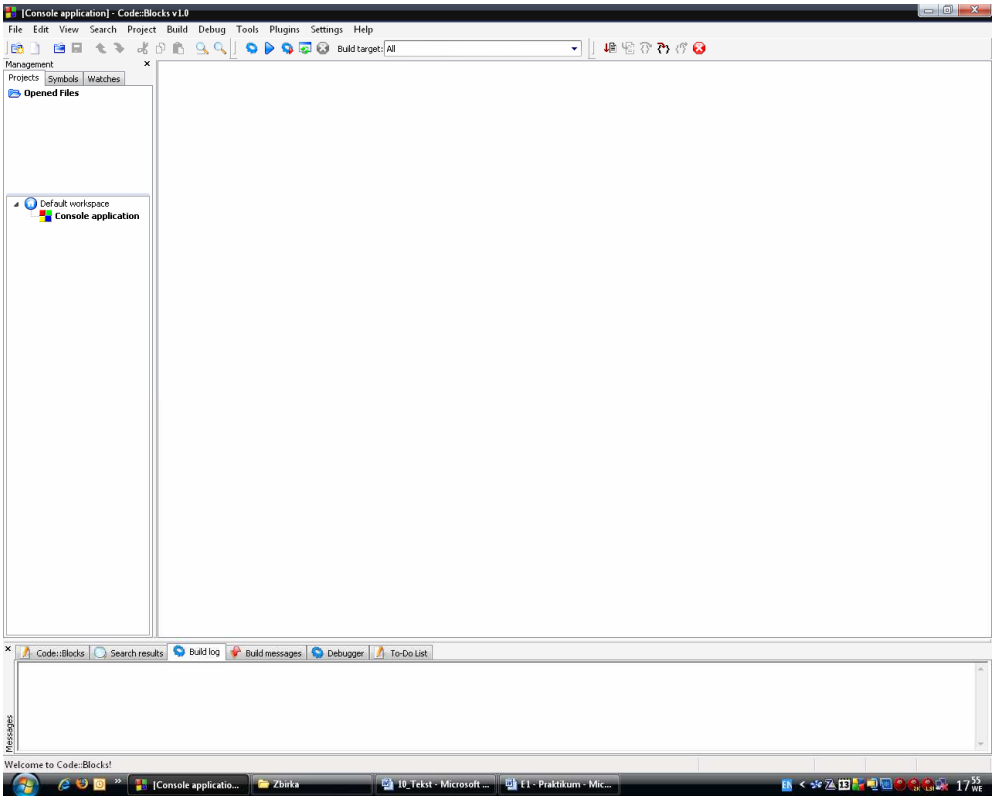
- U glavnom meniju biramo opciju `Settings`, a zatim biramo opciju `Compiler` i dobijamo prozor `Compiler Settings` u kojem biramo `GNU GCC Compiler`.

A.1 Kreiranje novog projekta

- Potrebno je napraviti radni direktorijum u kojem ćemo čuvati projekte. Neka je to, na primer, direktorijum `C:\Temp\e11111`.
- U glavnom meniju biramo opciju `File`, zatim biramo opciju `New Project` i dobijamo prozor u okviru kojeg klikom biramo `Console Application` kao što prikazuje slika 17. Ukoliko naš projekat treba da bude prazan potrebno je izabrati opciju `Do not create any files`, a zatim kliknuti na dugme `Create`, posle čega dobijamo prozor u kojem treba dati ime projekta i kliknuti na dugme `Save`.

Slika 17.

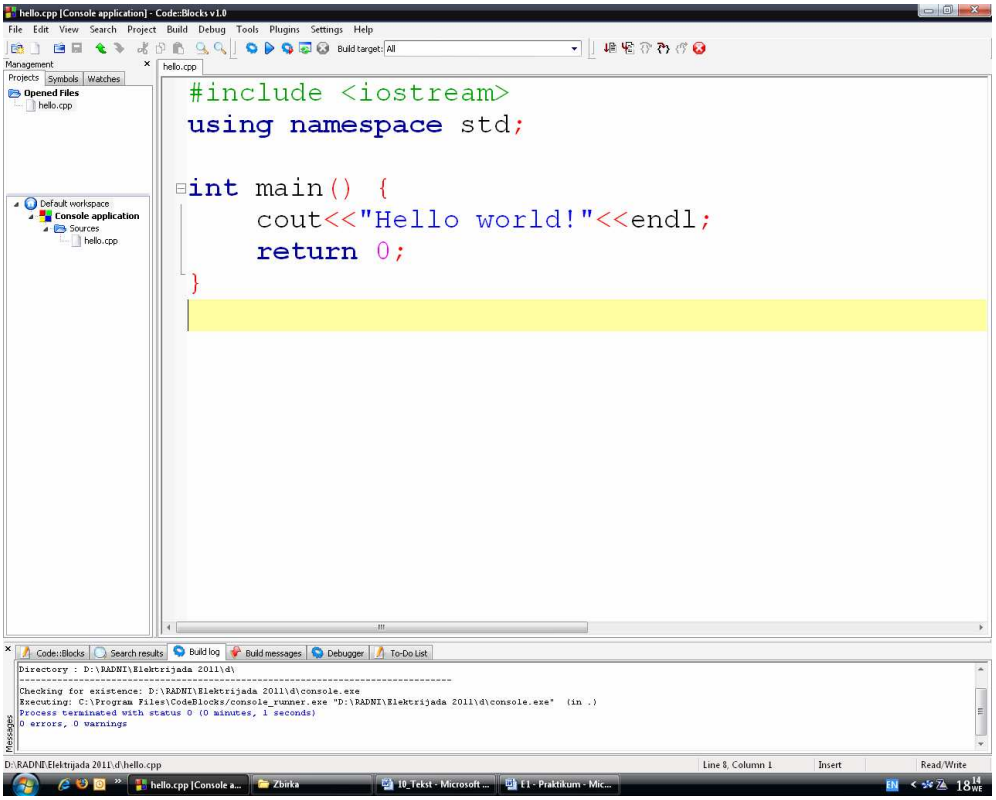
- Ukoliko je sve pravilno urađeno dobijamo izgled ekrana kao što je prikazano na slici 18.



Slika 18.

- U glavnom meniju biramo opciju `File`, a zatim `New File`, posle čega se pojavljuje prozor u kojem treba da navedemo ime datoteke (npr. `hello.cpp`) i zatim kliknemo na dugme `Save`. Na ekranu će se pojaviti pitanje da li želimo da se datoteka ubaci u trenutno aktivan projekat (`Do you want to add this file in the active project?`). Kliknemo na dugme `Yes` i dobijamo izgled ekrana kao što je prikazano na slici 19. U desnom delu prozora se nalazi prostor gde ćemo ukucati tekst našeg programa:

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

int main() {
    cout<<"Hello world!"<<endl;
    return 0;
}
```



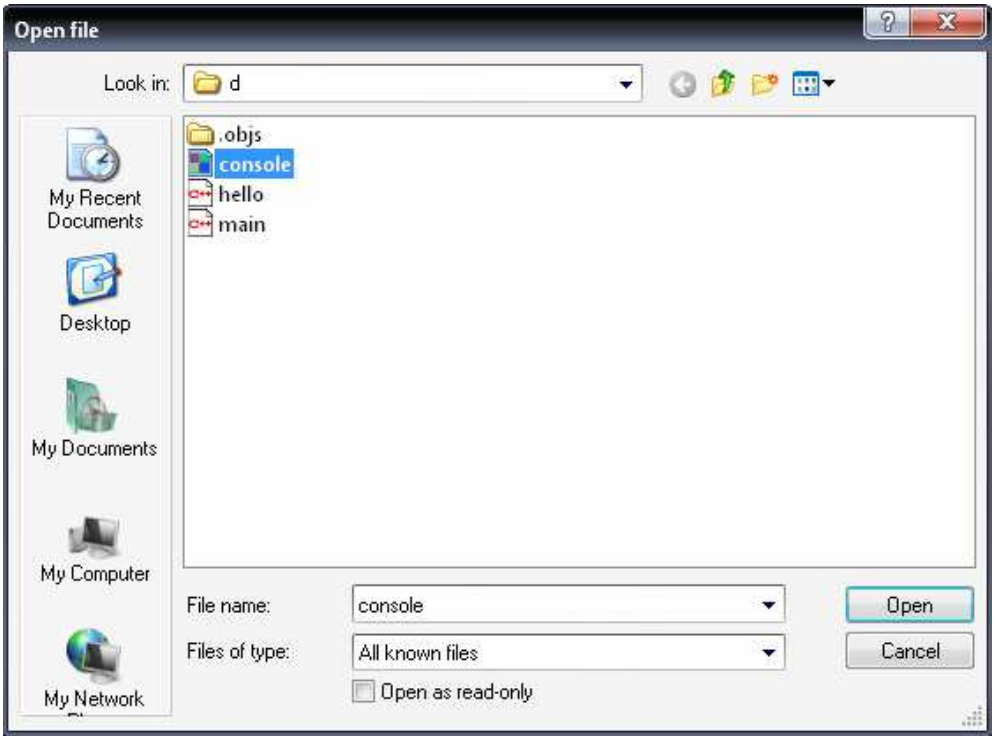
Slika 19.

- Kada je tekst programa napisan potrebno ga je snimiti. Biramo opciju `File`, a zatim opciju `Save`.
- Program se pokreće tako što se u glavnom meniju izabere opcija `Build` i dobija padajući meni u okviru koje se prvo bira opcija `Build` (ili `Ctrl+F9`), a zatim opcija `Run` (ili `Ctrl+F10`). Takođe, postoji i opcija `Build&run` (ili `F9`) koja objedinjuje opcije `Build` i `Run`. Dakle, kada je naš program napisan i snimljen, pritiskom na `F9` dobijamo poruku:

```
Hello world!

Press ENTER to continue.
```

A.2 Otvaranje postojećeg projekta

- Već postojeći projekat se može pokrenuti tako što se u glavnom meniju izabere opcija `File`, a zatim opcija `Open` i dobija se prozor koji je prikazan na slici 20. Projekat ima ekstenziju `.cbp`.



Slika 20.

A.3 Ubacivanje/izbacivanje datoteke

- Već napisana `.hpp` ili `.cpp` datoteka se može ubaciti u trenutno aktivni projekat tako što se u glavnom meniju bira opcija `Project`, a nakon toga opcija `Add Files...`.
- Datoteka se može izbaciti u trenutno aktivnog projekta tako što se u glavnom meniju bira opcija `Project`, a zatim opcija `Remove Files...`.

LITERATURA

[1.] Malbaški, D. : *Objektno orijentisano programiranje kroz programski jezik C++*, Novi Sad: Fakultet tehničkih nauka, 2008.
[2.] Malbaški, D. : *Objekti i objektno programiranje kroz programske jezike C++ i pascal*, Novi Sad: Fakultet tehničkih nauka, 2006.
[3.] Meyer, B. : *Object-Oriented Software Construction*, *(2nd ed.)*, Prentice Hall, 1997.
[4.] Kraus, L. : *Programski jezik C++ sa rešenim zadacima*, Beograd: Akademska misao, 2004.
[5.] Kraus, L. : *Programski jezik C sa rešenim zadacima*, Beograd: Akademska misao, 2001.
[6.] Milić, M., Bibić, R. i Bekut, D. : *Primena računara u elektroenergetici: C++ zbirka rešenih zadataka*, Novi Sad: Fakultet tehničkih nauka, 2007.
[7.] Milićev, D., Lazarević, Lj. i Marušić, J. : *Objektno orijentisano programiranje na jeziku C++: skripta sa praktikumom*, Beograd: Mikro knjiga, 2001.
[8.] Stroustrup, B. : *The C++ Programming Language*, *(3rd ed.)*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.
[9.] Eckel, B. : *Thinking in C++*, *(2nd ed.)*, Prentice Hall, New Jersey, 2000.
[10.] Kupusinac A., Malbaški D. : Invarijanta klase u objektno orijentisanom programiranju i njena primena, *15. TELFOR*, Beograd: Društvo za telekomunikacije, 20-22 Novembar, 2007, pp. 589-592, ISBN 978-86-7466-301-1.
[11.] Kupusinac A., Malbaški D. : Class composition and correctness, *12. Serbian Mathematical Congress*, Novi Sad: PMF, 28-2 Avgust, 2008, pp. 119-123.
[12.] Kupusinac A., Malbaški D. : Korektnost potklase, *16. TELFOR*, Beograd: Društvo za telekomunikacije, 25-27 Novembar, 2008, pp. 747-750, ISBN 978-86-7466-337-0.
[13.] Malbaški D., Kupusinac A. : Konceptualna definicija klase i objekta, *8. International Symposium on Industrial Electronics (INDEL)*, Banja Luka: ETF, 4-6 Novembar, 2010, pp. 368-371, ISBN 978-99955-46-03-8.
[14.] Malbaški D., Kupusinac A. : Klasifikacija invarijanata u klasi, *18. TELFOR*, Beograd: Društvo za telekomunikacije, 23-25 Novembar, 2010, pp. 1220-1223, ISBN 978-86-7466-392-9.
[15.] Malbaški D., Kupusinac A. : Elegancija i složenost algoritma, 18. TELFOR, Beograd: Društvo za telekomunikacije, 23-25 Novembar, 2010, pp. 1201-1203, ISBN 978-86-7466-392-9.
[16.] http://www.codeblocks.org