Mr Aleksandar Kupusinac

Dr Dušan Malbaški

Univerzitet u Novom Sadu

Fakultet tehničkih nauka

Departman za računarstvo i automatiku

Katedra za primenjene računarske nauke

21000 Novi Sad, Trg Dositeja Obradovića 6

PRAKTIKUM ZA VEŽBE IZ OBJEKTNO ORIJENTISANOG PROGRAMIRANJA

Novi Sad
2010

Sadržaj

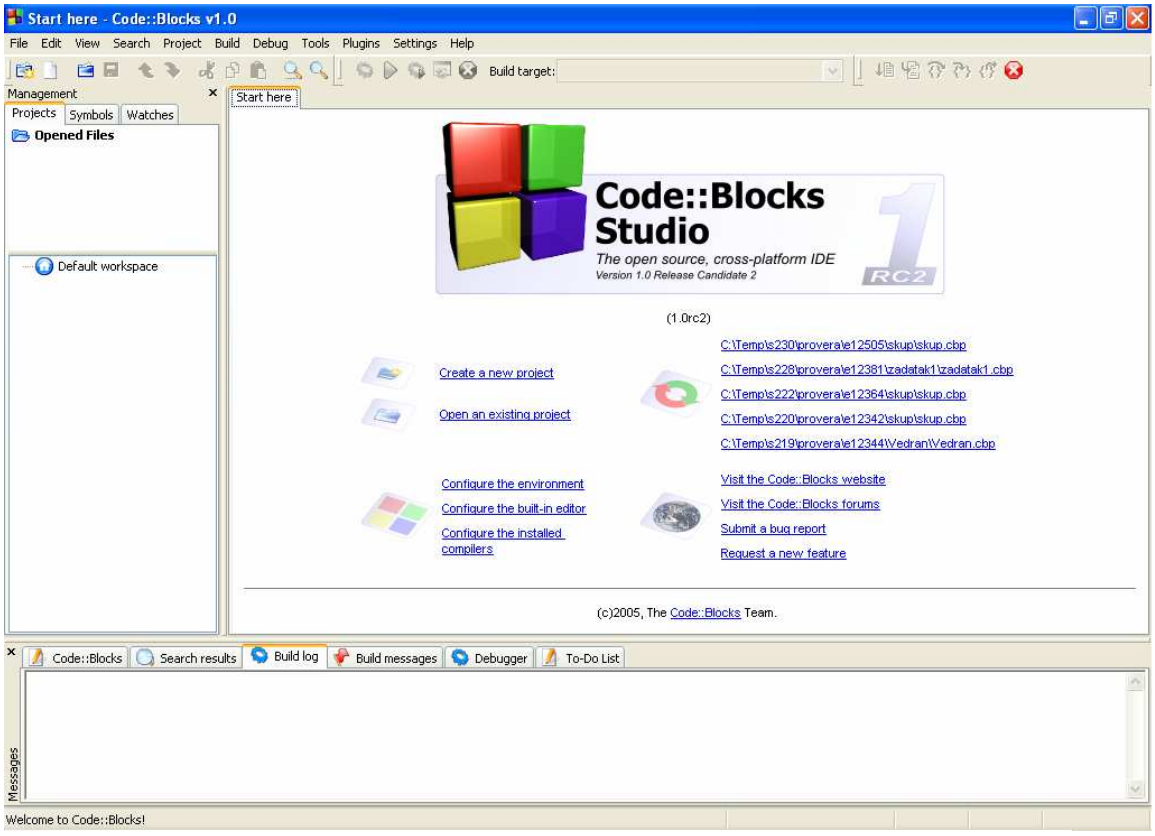
Vežbe 1 3
PROGRAMSKO OKRUŽENJE Code::Blocks 3
PROMENLJIVE 6
POKAZIVAČI 7
NIZOVI 9
FUNKCIJE 11
Vežbe 2 13
REFERENCE 13
ULAZ I IZLAZ PODATAKA 14
Vežbe 3 22
OSNOVE OBJEKTNOG PROGRAMIRANJA 22
DEFINICIJA KLASE U C++ 22
KLASA `Semaphore` 27
Vežbe 4 29
KLASA `FMRadio` 29
KOMPOZICIJA KLASE 30
Vežbe 5 35
UGRAĐENI KONSTRUKTOR 35
KONSTRUKTOR KOPIJE 36
PREKLAPANJE METODA 37
PREKLAPANJE OPERATORA 38
Vežbe 6 43
KLASA `DinString` 43
Vežbe 7 48
NASLEĐIVANJE 48
Vežbe 8 51
NASLEĐIVANJE - ZADACI 51
METODE IZVEDENE KLASE 55
KLASE `Person`, `Student` i `PhDStudent` 57
Vežbe 9 67
VIRTUELNE METODE 67
KLASE `Person`, `Student` i `PhDStudent` 69
APSTRAKTNE KLASE 71
ZAJEDNIČKI ČLANOVI KLASE 72
Vežbe 10 77
GENERIČKE KLASE 77
Vežbe 11 82
GENERIČKE KLASE - ZADACI 82

Vežbe 1

PROGRAMSKO OKRUŽENJE Code::Blocks

Na vežbama ćemo koristiti programsko okruženje `Code::Blocks`. Instalacija programskog okruženja `Code::Blocks` se može besplatno preuzeti na sajtu http://www.codeblocks.org/ . Na sajtu je dostupna instalaciona verzija za operativni sistem MS Windows i verzija za operativni sistem Linux (pogledajte http://www.codeblocks.org/downloads/binaries). Takođe, na sajtu se može preuzeti uputstvo za rad sa programskim okruženjem `Code::Blocks` (pogledajte http://www.codeblocks.org/user-manual).

Sada ćemo se ukratko upoznati kako ćemo pisati i pokretati programe u programskom okruženju `Code::Blocks`. Pokretanjem programskog okruženja `Code::Blocks` dobijamo izgled ekrana kao na slici 1.1.

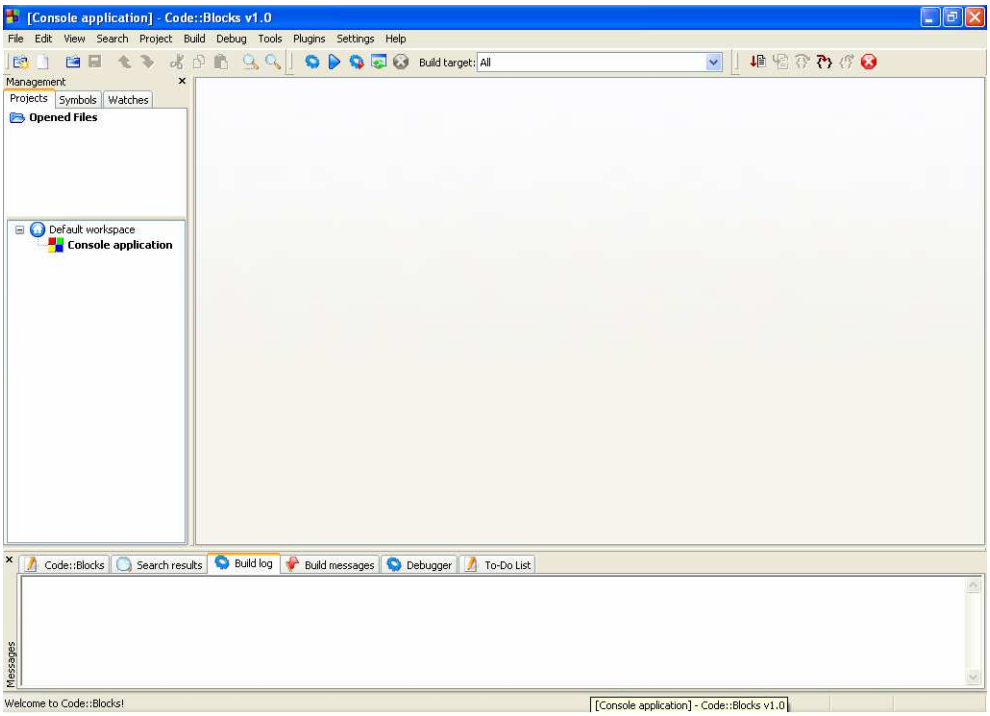


Slika 1.1.

Potrebno je izabrati radni direktorijum u kojem ćemo čuvati projekte. Neka je to, na primer, direktorijum `C:\Temp\e11111`. Sada ćemo napraviti projekat. U glavnom meniju biramo opciju `File`, zatim biramo opciju `New Project` i dobijamo prozor u okviru kojeg klikom biramo `Console Application` kao što prikazuje sklika 1.2. 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`.

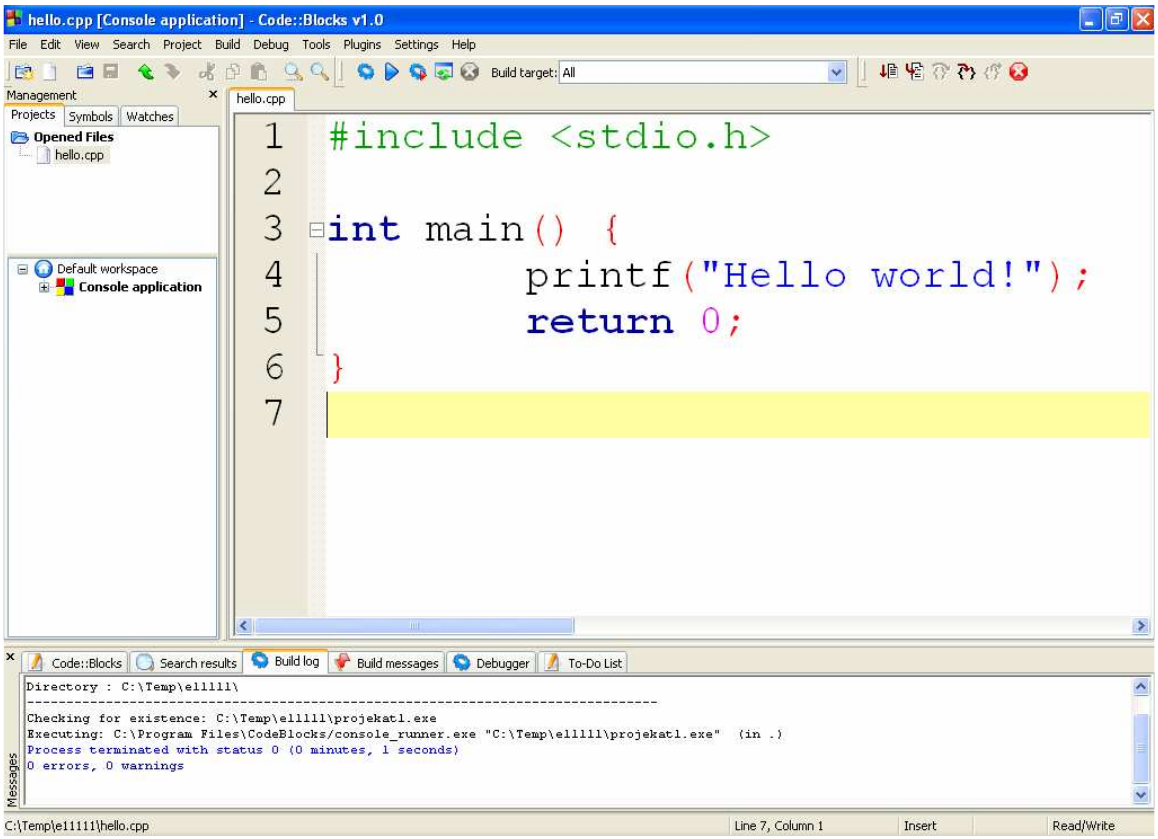
Slika 1.2.

Ukoliko je sve pravilno urađeno dobijamo izgled ekrana kao što je prikazano na slici 1.3.



Slika 1.3.

U glavnom meniju biramo opciju `File`, a zatim `New File`, posle čega se pojavljuje prozor u kojem treba da navedemo ime fajla (na primer `hello.cpp`) i zatim kliknemo na dugme `Save`. Na ekranu će se pojaviti pitanje da li želimo da se naš fajl 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 1.4. U desnom delu prozora se nalazi prostor gde ćemo ukucati tekst našeg programa.



Slika 1.4.

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, kada se u glavnom meniju izabere opcija `Build` u padajućem meniju postoji opcija `Build&run` (ili `F9`) koja objedinjuje opciju `Build` i `Run`. Dakle, kada je naš program napisan i snimljen, pritiskom na `F9` dobijamo korisnički ekran na kojem vidimo da je ispisana poruka `Hello world!`, što je i bio cilj ove vežbe.

Već postojeći projekat se može pokrenuti tako što se u glavnom meniju izabere opcija `File`, a zatim opciju `Open`. Projekat ima ekstenziju `.cbp`.

PROMENLJIVE

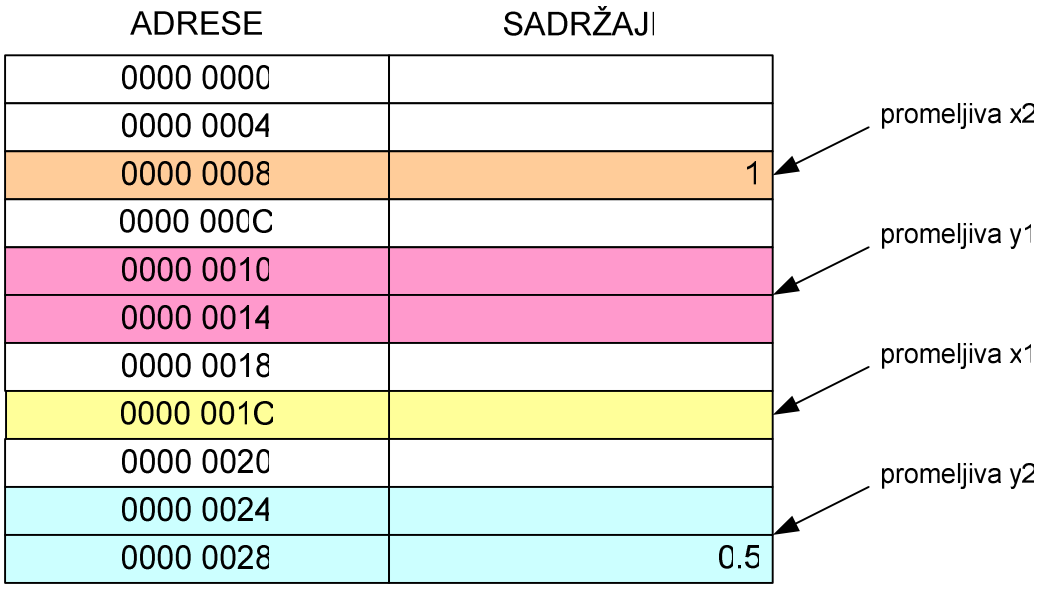
Sada ćemo se podsetiti pojedinih delova gradiva koje smo naučili baveći se proučavanjem programskog jezika C, a koja su nam neophodna za dalji rad.

- 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, što zavisi od hardvera koji nam je na raspolaganju. Na primer, ako imamo na raspolaganju 32-bitno adresiranje, tada za promenljivu tipa `int` biće zauzeto 32 bita, za promenljivu tipa `double` biće zauzeto 64 bita itd.

- Ime promenljive mora biti jedinstveno. Ime promenljive jeste simbolički predstavljena adresa na kojoj se nalazi memorija koja je zauzeta za datu promenljivu. Na primer, za 32-bitne adrese i lokacije će biti 32 bita, pa možemo zamisliti da će memorija posle gore navedenih deklaracija izgledati kao na slici 1.5. Adrese su napisane heksadecimalno. Svaka lokacija sadrži 4 bajta i svaki bajt je adresibilan, pa se adrese razlikuju za 4 bajta. Vidimo da za promenljive `x1` i `x2` će biti zauzeta po jedna lokacija, tj. po 32 bita, jer je promenljiva 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, možemo zaključiti da kada se u višem programskom jeziku govori o promenljivoj `x1`, u našem primeru računar to razume tako da se zapravo radi o promenljivoj na adresi 1C.



Slika 1.5.

- Veličina memorije u bajtovima, koja je potrebna za smeštanje podatka određenog tipa može se utvrditi primenom prefiksnog unarnog operatora `sizeof`.

POKAZIVAČI

- Pokazivač je promenljiva koja sadrži neku adresu.

- Pokazivač se deklariše na ovaj način:

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

- Pošto pokazivač sadrži adresu možemo zaključiti da bez obzira na koji tip pokazuju, svi pokazivači su istog formata, tj. veličina memorija svih pokazivača je ista (ako su adrese 32 bita, onda će svaki pokazivač zauzimati 32 bita).

Zadatak 1.1

Napisati program koji primenom operatora `sizeof` odrediti veličinu memorije u bitima koja potrebna za smeštanje podataka tipa: `char`, `int`, `double`, `char*`, `int*`, `double*` i `void*`.

Rešenje.

```
#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 za dati tip izraženu u bajtima, pa zbog toga taj broj množimo sa 8. Pokretanjem programa dobija sledeći ispis na ekranu:

```
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
```

- Kada kažemo da neki pokazivač «pokazuje» na neku promenljivu, to zapravo znači da adresa te promenljive je sadržaj pokazivača.

- Unarni operator `&` vraća memorijsku adresu nekog podatka (na primer, `&x`).

- Primenom unarnog operatora `*` može se posredno pristupiti nekom podatku pomoću memorijske adrese (na primer, `*p`).

Zadatak 1.2

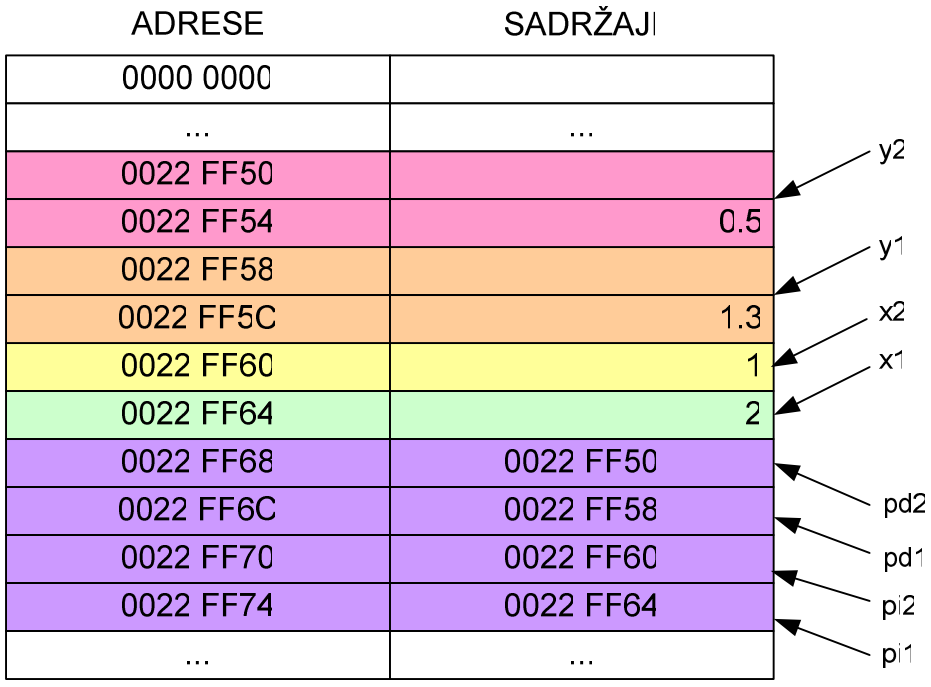
Analizirati ispis sledećeg programa:

```
#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 je adresa promenljive x1: %d",pi1);
    printf("\nSadrzaj pokazivaca pi2 je adresa promenljive x2: %d",pi2);
    printf("\nSadrzaj pokazivaca pd1 je adresa promenljive y1: %d",pd1);
    printf("\nSadrzaj pokazivaca pd2 je adresa promenljive y2: %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;
}
```

Izvršavanjem programa na dobijamo sledeći ispis na ekranu:

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

Sadrzaj pokazivaca pi1 je adresa promenljive x1: 22ff64
Sadrzaj pokazivaca pi2 je adresa promenljive x2: 22ff60
Sadrzaj pokazivaca pd1 je adresa promenljive y1: 22ff58
Sadrzaj pokazivaca pd2 je adresa promenljive y2: 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 adresa slobodnih memorijskih lokacija je proizvoljan. Na osnovu gornjeg ispisa možemo zamisliti da će memorija imati izgled kao što je prikazano na slici 1.6. Za promenljive `y1` i `y2` je zauzeto po 64 bita, jer su tipa `double`, a za promenljive `x1` i `x2` po 32 bita, jer su tipa `int`. Za pokazivače `pd1`, `pd2`, `pi1` i `pi2` je zauzeto po 32 bita.



Slika 1.6.

NIZOVI

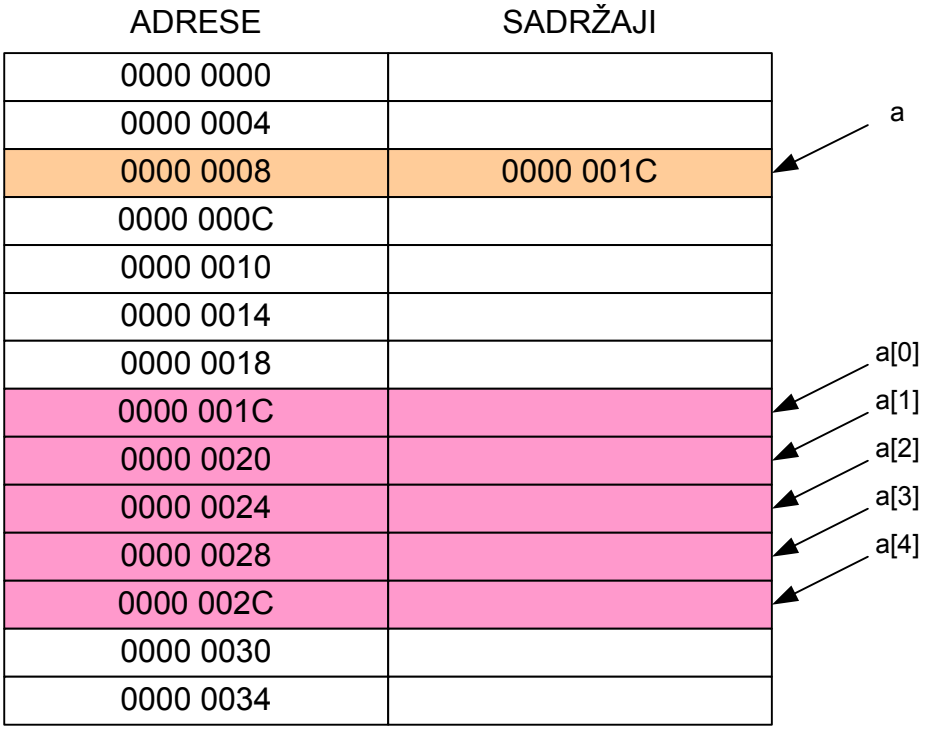
- Ime niza je pokazivač koji sadrži adresu prvog elementa niza.
- Memorija za elemente niza može biti zauzeta statički i dinamički, pa razlikujemo statičke i dinamičke nizove.
- Statički niz deklarišemo na sledeći način:

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

- Dinamički niz dobijamo na sledeći način:

```
int *a;  // pokazivac a
a=(int*)malloc(5*sizeof(int));  // dinamicko zauzimanje
```

- Na slici 1.7 je prikazana organizacija u memoriji celobrojnog niza niza `a` koji ima 5 elemenata. Ime niza je pokazivač koji sadrži adresu početka niza.



Slika 1.7.

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

Zadatak 1.3

Napisati program za pronalaženje maksimalnog elementa statičkog niza celih brojeva (niz može imati najviše 30 elemenata).

Rešenje.

```
#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 1.4

Napisati program za sortiranje dinamičkog niza celih brojeva po neopadajućem redosledu.

Rešenje.

```
#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;
}
```

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**.

- Funkcijama su potprogrami koji na osnovu argumenata daju jedan rezultat koji se naziva **vrednost funkcije**.
- Deklaracija funkcije ili prototip:

```
TipFunkc imeFunkc (TipArg_1, TipArg_2, ... TipArg_n);
```

- Definicija funkcije:

```
TipFunkc imeFunkc (Arg_1, Arg_2, ... Arg_n) {

      // TELO FUNKCIJE
}
```

- Poziv funkcije: `imeFunkc(izraz_1, izraz_2, ... izraz_n);`
- Zadatak koji sledi će ilustrovati razliku između prenosa argumenata po vrednosti i po adresi.

Zadatak 1.5

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;
}

void f1(int a) {
    a=3;
}

void f2(int *a) {
    *a=3;
}
```

Pokretanjem programa na ekranu se dobija sledeći ispis:

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

Vežbe 2

REFERENCE

- Referenca ili upućivač u programskom jeziku C++ je alternativno ime za neki podatak (drugo ime za neki podatak).

- Reference nisu podaci. Reference ne zauzimaju prostor u memoriji, pa se ne može tražiti njihova adresa. Reference 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.

- Reference ne mogu da promene vrednost. Sve izvršene operacije nad referencama deluju na originalne podatke.

- Referenca se definiše na ovaj način:

```
int x=5;
int &rx=x; //rx je referenca inicijalizovana promenljivom x

double y1;
double y2;
double &ry; //greska, ovo ne moze
double &ry=y1; //sad je u redu
ry=y2; //greska, jer referenca ne moze da menja vrednost
```

- Reference služe da bi se izbeglo korišćenje pokazivača. Reference su slične pokazivačima, ali postoje određene razlike.

- SLIČNOST REFERENCI SA POKAZIVAČIMA:

    - Referenca je adresa (drugo ime za originalni podatak), a vrednost pokazivača je takođe adresa. Dakle, njihova ostvarenja su ppreko adresa podataka kojima su pridruženi.

- RAZLIKE IZMEĐU REFERENCI I POKAZIVAČA:

    - Referenca je čvrsto vezana za podatak, dok vrednost pokazivača nije čvrsto vezana za podatak, tj. pokazivač može da sadrži adresu jednog podatka i nakon toga može da se promeni njegova vrednost i da 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 određen prostor u memoriji i može im se po potrebi promeni 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

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 dobijene vrednosti adresa će se razlikovati, na ekranu 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
```

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 levog operanda za onoliko binarnih mesta koliko je vrednost desnog 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 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 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: `", zatim ispis vrednosti promenljive `x` i prelazak u novi red će se pisati:

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

- Neka su `x` i `y` promenljive tipa `int`, tada unos dva podatka tipa `int` i njihovo smeštanje u promenljive `x` i `y` će se pisati:

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

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

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

Zadatak 2.2

Napisati program koji izračunava i ispisuje zbir dva unešena 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 2.3

Napisati program koji ispisuje niz od prvih `n` Fibonačijevih brojeva, gde je `n` prirodan broj koji nije veći od `100`.

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

int main() {
    int i, n;
    int f[100];
```

```
    cout<<"Unesite prirodan broj n koji nije veci od 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;
}
```

PODRAZUMEVANE VREDNOSTI FUNKCIJA

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

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

- Ako se u listi argumenata za neki argument želi navesti podrazumevana vrednost, onda je potrebno navesti podrazumevane vrednosti za sve argumente posle njega.

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

Zadatak 2.4

Potrebno je postaviti pločice u kuhinji. Napisati program koji za zadate dimenzije površine izračunava broj potrebnih pločica da bi se data površina pokrila. Prodavac isporučuje samo pločice oblika kvadrata i uslužuje kupca tako što ukoliko kupac ne navede željene dimenzije pločica prodavac isporučuje pločice koje uvek ima u magacinu i koje su oblika kvadrata sa stranicom `15cm`.

```
#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
    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<<"Da li zelite da Vi izaberete dimenzije plocica? [Y/N]... ";
    cin>>odg;
    if(odg=='Y' || odg=='y') {
        cout<<"Plocice su oblika kvadrata. ";
        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;
}
```

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. Tipovi funkcija sa istim imenom mogu biti isti.

Zadatak 2.5

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 napisanog programa dobija se ispis:

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

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 `::` (operatora za razrešenje dosega), odnosno izrazom:

```
Imenski_prostor::identifikator
```

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

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

ili

```
using namespace Imenski_prostor;
```

Zadatak 2.6

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 na ekranu se dobija ispis:

```
6
10
6
10
```

- Po Standardu programskog jezika C++ je predviđeno da sva standardna zaglavlja sve globalne identifikatore stavljaju u imenski prostor **`std`**. Standardom je predviđeno da standardna zaglavlja ne budu u vidu tekstualnih datoteka, pa zbog toga standardna zaglavlja ne sadrže proširenje imena sa **.h** (kao što je to slučaj u C-u). Jedno od takvih zaglavlja jeste i `<iostream>`. Zbog toga ćemo uvek pisati na početku programa:

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

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

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

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

```
cout<<x;
```

- Na primer:

```
#include <iostream>

int main() {
    int x=5;

    std::cout<<x;

    using namespace std;
    cout<<x;

    return 0;
}
```

DOBIJANJE IZVRŠNE DATOTEKE

- Obrada programa u programskom jeziku C++ obihvata sledeće faze:
  - Unošenje izvornog teksta programa i dobijanje **`.cpp`** datoteka,
  - Prevođenje (kompajliranje) i dobijanje **`.obj`** datoteka,
  - Povezivanje prevedenih datoteka u jednu izvršnu datoteku **`.exe`**.

- Pretprocesiranje podrazumeva određene obrade izvornog teksta programa pre početka prevođenja.
- Pretprocesorski direktivama se može postići sledeće:
  - Zamena identifikatora sa nizom simbola u programu sa direktivom **`#define`**

    ```
    #define identifikator niz_simbola
    ```

  - Umetanje sadržaja datoteke sa direktivom **`#include`**

    ```
    #include "korisnicka_datoteka"
    #include <sistemska_datoteka>
    ```

  - Uslovno prevođenje delova programa:

    ```
    #ifndef identifikator  /*Ako identifikator nije definisan*/
    #endif /*Kraj uslovnog prevođenja*/
    ```

- Zaglavlje (*header*) je **`.hpp`** datoteka u kojoj se nalaze razne deklarativne naredbe. Zaglavlje se pretprocesorskom direktivom **`#include`** uključuje u izvorni program (u datoteku **`.cpp`**). Da bi se sprečilo višestruko prevođenje sadržaja datoteke u toku jednog prevođenje preporučuje se da struktura zaglavlja bude sledeća:

  ```
  // datoteka proba.hpp
  #ifndef PROBA_DEF
  #define PROBA_DEF

  /* Definicije koje smeju da se prevedu samo jedanput. */

  #endif
  ```

- Glavni program je funkcija `main()` i po Standardu bi trebalo da bude tipa `int`.

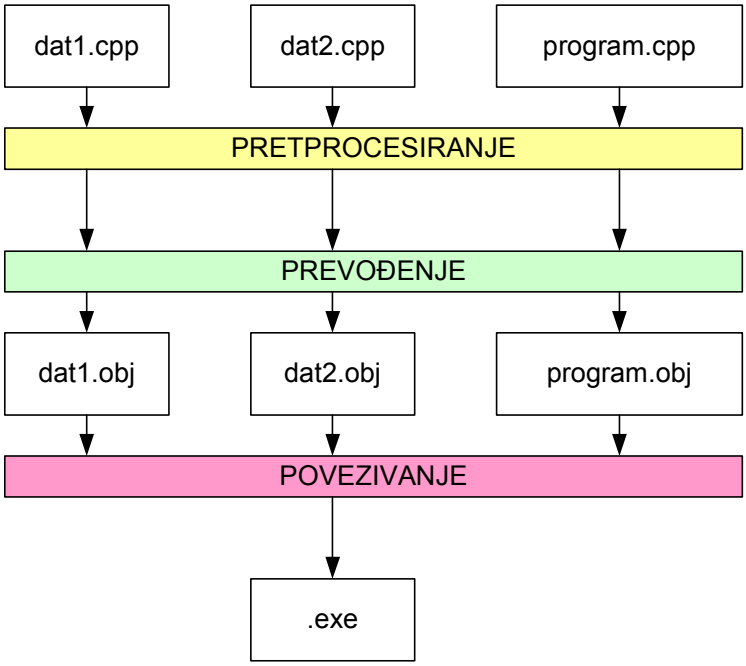
Zadatak 2.6

Analizirati sledeće datoteke:

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

#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: program.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 `program.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 (kompajler) prevesti izvorne datoteke u nastaće datoteke `dat1.obj`, `dat2.obj` i `program.obj` iz kojih će se povezivanjem (linkovanjem) dobiti izvršna datoteka. Ceo tok je ilustrovan na sledećoj slici:



Vežbe 3

OSNOVE OBJEKTNOG PROGRAMIRANJA

- Prilikom rešavanja nekog problema, objektni programer najviše će vremena posvetiti analizi i modelovanju problema, sa ciljem da identifikuje potrebne elemente, tzv. **entitete** i njihove međusobne veze u okviru domena problema. Da bi se dobilo uređeno znanje o nekom domenu problema, entiteti se grupišu u kolekcije koji se nazivaju **klase entiteta**. Terminom entitet se označava da nešto postoji, a to nešto ima neka svojstva i stoji u nekom odnosu sa drugim entitetima. Termin entitet se uveliko ustalio u prirodnim i inženjerskim naukama.

- Klasa entiteta je kolekcija entiteta, koji su prema nekom kriterijumu, međusobno slični.

- Model entiteta u okviru datog domena problema gradi se na osnovu skupa odabranih **relevantnih osobina entiteta**. Entitet poseduje sve osobine, a njegov model samo odabrane. Jedan isti entitet može imati različite relevantne osobine u različitim domenima problema, na primer, za evidenciju studenata u Studentskoj službi relevantne osobine su – ime i prezime, broj indeksa, broj položenih ispita i sl., dok osobine – visina, težina, broj cipela i sl. nisu. Međutim, za evidenciju Studentske ambulante relevantne osobine su ime i prezime, broj indeksa, visina, težina i sl., dok broj položenih ispita nije.

- **Objekat** predstavlja model entiteta, a **klasa** (objekata) predstavlja model klase entiteta.

- Ne manje važan aspekt objektne metodologije jeste **realizacija** klase u nekom objektno orijentisanom programskom jeziku. Određena klasa entiteta se može modelovati na više različitih načina, a isto tako određeni model klase entiteta (klasa objekata) se može realizovati na više različitih načina.

DEFINICIJA KLASE U C++

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

```
class MyClass {
      // <PODACI-ČLANOVI>
      // <OBJEKTI-ČLANOVI>
      // <FUNKCIJE-ČLANICE ili METODE>
};
```

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

```
MyClass
KompleksniBroj
JednakostranicniTrougao
XYVektor
```

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

```
int a;
Ttip t;
```

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

```
MyClass m;
```

- 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 podatka člana tipa `double`:

```
class XYPoint {
      private:
/*PODACI CLANOVI*/
            double x;
            double y;
      public:
/*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
};
```

- Prava pristupa mogu biti:
  - `private`
  - `public`

- Članu klase koji je u **private**-segmentu može se pristupati samo iz unutrašnjosti klase (iz metoda).
- Članu klase koji je u **public**-segmentu može se pristupati kako iz unutrašnjosti, tako i iz spoljašnjosti klase (iz metoda, iz programa koji koriste tu klasu itd.).

- Preporučuje se da podaci-članovi i objekti-članovi budu private, a da metode budu public. Kasnije ćemo na primeru klase Trougao objasniti zašto je to preporučljivo.

- **Konstruktor** je metoda koja kreira objekat. Osobine konstruktora su sledeće:
  - Konstruktor ima isto ime kao i klasa
  - Konstruktor može i ne mora da ima argumente
  - Konstruktor nikad ne vraća vrednost (nikada nema tip i na kraju tela nema naredbu `return`)

- Set-metoda služi da postavi novu vrednost podatka-člana i ima argument tipa koji odgovara tipu podatka-člana kojeg menja, obično je tipa void i nema oznaku **`const`** jer menja stanje objekta.
- Get-metode služe da očitaju vrednost nekog podatka-člana i uvek su tipa koji odgovara tipu tog podatka-člana, obično nemaju argumente i imaju oznaku **`const`** jer ne menjaju stanje objekta.

- PAŽNJA!!! Veoma često studenti zaborave da se definicija klase završava sa znakom **;** (tačka-zarez) i tada se javlja sintaksna greška koju je teško otkriti.

Zadatak 3.1

Napisati klasu XYPoint koja modeluje tačku u xy-ravni. Napisati kratak 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
```

Klasa `XYPoint` se nalazi između direktiva `#ifndef` i `#endif` da bi se sprečilo višestruko prevođenje. Metoda `distance()` nije realizovana u **`.hpp`** fajlu, pa će biti realizovana u **`.cpp`** fajlu. Da bi u **`.cpp`** fajlu se znalo da je metoda `distance()` iz klase `XYPoint` potrebno je koristi operator za razrešenje dosega **`::`** (dva puta dvotačka). Da bi sve deklarativne naredbe iz **`.hpp`** fajla bile dostupne u **`.cpp`** fajlu potrebno je direktivom `#include` uključiti **`.hpp`** fajl u odgovarajući **`.cpp`** fajl.

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

double XYPoint::distance() const {
       return sqrt(x*x + y*y);
}


// Datoteka: xypoint.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 polje x private
       p.setY(5);
// ne mozemo pisati p.y=5; jer je polje y private
       cout<<"Tacka ima koordinate: "<<p.getX()<<" i "<<p.getY()<<endl;
       return 0;
}
```

Zadatak 3.2

Napisati klasu `Trougao`. Napisati kratak test program.

```
// Datoteka: trougao.hpp
#ifndef TROUGAO_DEF
#define TROUGAO_DEF

#include <math.h>
#include <iostream>
using namespace std;

class Trougao {
      private:
/*   P O D A C I - C L A N O V I   */
           double a;
           double b;
           double c;
      public:
/*   M E T O D E   */
           /* 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: test.cpp
#include "trougao.hpp"

int main() {
       Trougao t;
       cout<<"Stranica a: "<<t.getA()<<endl;
       cout<<"Stranica b: "<<t.getB()<<endl;
       cout<<"Stranica c: "<<t.getC()<<endl;
       cout<<"Obim: "<<t.getO()<<endl;
```

```
	cout<<"Povrsina: "<<t.getP()<<endl;
	return 0;
}
```

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

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

Međutim, trougao sa dužinama stranica 5, 1 i 1 ne može postojati, jer ne važi $1^2+1^2>5^2$. Zato je bolje da se postavljanje novih vrednosti polja ne radi direktno, već preko metoda. Ukoliko se izmena vrednosti podataka-članova vrši preko metoda onda se te izmene mogu kontrolisati ranim mehanizmima, na primer odgovarajućim if-blokom naredbi:

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

- Ukoliko bi objekat klase **`Trougao`** bio u takvom stanju da podaci-članovi koji predstavljaju dužine stranica imaju vrednosti 5, 1 i 1 tada se objekat nalazi u nedozvoljenom stanju, jer u tom trenutku predstavlja entitet koji ne može da postoji (trougao sa dužinama stranica 5, 1 i 1 ne postoji). Programer je dužan da napiše takvu klasu da svaki njen objekat u svakom trenutku bude u dozvoljenom stanju (ne sme da bude u nedozvoljenom stanju).

- Međutim, na vežbama ćemo podrazumevati da neće biti situacija u kojim bi objekat prešao u nedozvoljeno stanje, tako da dodatne provere nećemo raditi unutar metoda. Dakle, prethodno razmatranje je poslužilo samo da objasnimo zašto je preporučljivo da podaci-članovi budu **`private`**, a metode **`public`**.

Zadatak 3.3

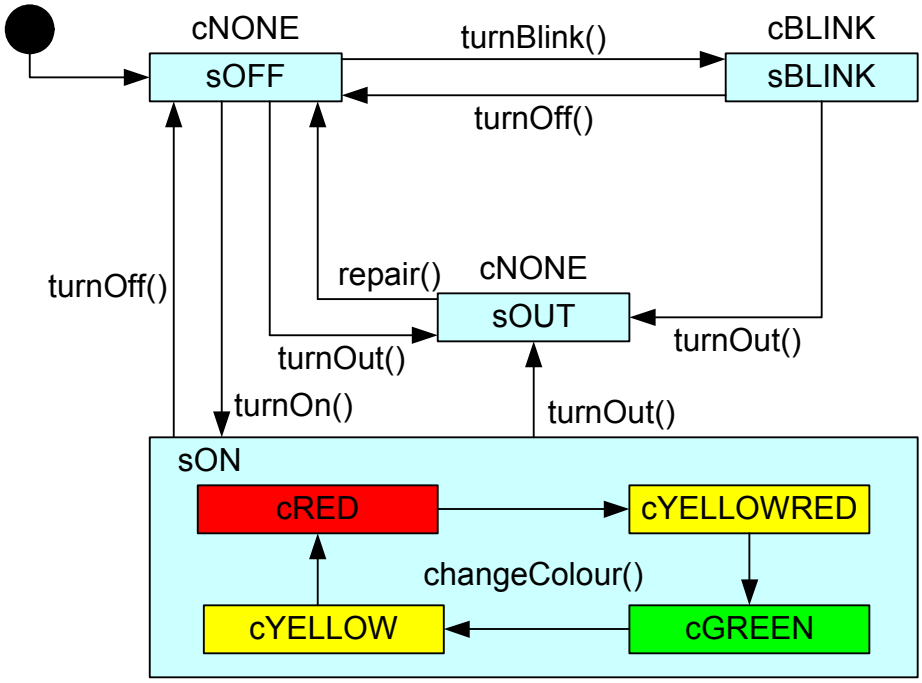
Napisati klasu `Kvadrat`. Napisati kratak test program.

Zadatak 3.4

Napisati klasu `Pravougaonik`. Napisati kratak test program.

KLASA `Semaphore`

- Dijagram rada semafora je ilustrovan na sledećoj slici:



- Semafor može biti u stanjima: **`sOFF`**, **`sON`**, **`sBLINK`** i **`sOUT`**.
- Svetlo semafora može biti: **`cNONE`**, **`cBLINK`**, **`cRED`**, **`cYELLOWRED`**, **`cGREEN`** i **`cYELLOW`**.
- Inicijalno stanje semafora je **`sOFF`**, a svetlo **`cNONE`**.
- Aktiviranjem odgovarajućih metoda semafor menja stanje i svetlo.
- U zadatku ćemo koristiti nabrojane konstante. Nabrojane konstante su celobrojne (tip `int`) konstante koje se definišu nabrajanjem u naredbama oblika:

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

- Imena konstanti u nabrajanju su identifikatori simboličkih konstanti kojima se dodeljuju vrednosti konstantnih celobrojnih izraza označenih sa vrednost konstante. 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.
- U zadatku koristićemo dva nabrajanja:

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

enum Colours {cNONE, cBLINK, cRED, cYELLOWRED, cGREEN, cYELLOW};

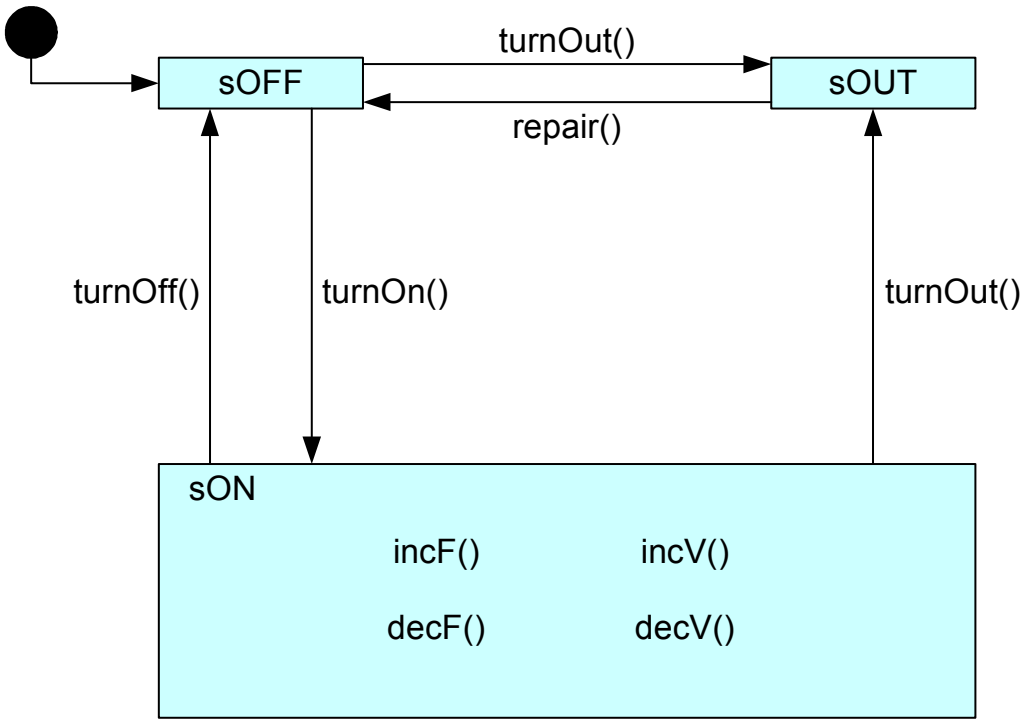
- Klasa Semaphore:

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

enum States {sOFF, sON, 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();
        void 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;
}

/* TREBA REALIZOVATI OSTALE METODE */
```

Vežbe 4

KLASA `FMRadio`

- Napisati klasu `FMRadio`. Klasa treba da 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 0.5. Jačina zvuka se nalazi u opsegu od 0 do 20. Promena jačine zvuka se uvek vrši za korak 1.
- U stanju `sOFF` i `sOUT` frekvencija i jačina imaju vrednost 0.
- Kada objekat pređe u stanje `sON` frekvencija se postavlja na 87.5MHz. Frekvencija i jačina zvuka se mogu menjati samo u stanju `sON`.
- Inicijalno stanje je `sOFF`.
- Na sledećoj slici 4.1 je prikazan dijagram rada.



Slika 4.1

KOMPOZICIJA KLASE

- Već smo rekli da entiteti poseduju različite osobine i međusobno stoje u nekim odnosima. Metodom apstrakcije se odnos između pojedinačnih entiteta može postaviti na viši nivo, tj. nivo klasa entiteta.
- Dalje izlaganje će biti usmereno konkretno na razmatranje odnosa «poseduje». Na primer, posmatra se entitet «automobil», koji poseduje entitet «motor». Metodom apstrakcije dolazi se do zaključka da svaki entitet iz klase entiteta «automobil» poseduje odgovarajući entitet iz klase entiteta «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). Sada se može uočiti da entitet «automobil» predstavlja entitet-celinu koja sadrži entitet-deo «motor» i može se reći.
- U fazi implementacije 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 svaka instanca klase A poseduje bar jednu instancu klase B, pri čemu stvaranje i uništavanje date instance klase B zavisi od stvaranja i uništavanja instance klase A, onda se kaže da između klase A i klase B postoji veza kompozicije.

Zadatak 4.2

Napisati klase `Krug` i `Pravougaonik`. Napisati klasu `Valjak` koja ima dva objekta-člana: `B` (objekat klase `Krug`) i `M` (objekat klase `Pravougaonik`). Napisati kratak test program.

```
// 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: test.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;
}
```

- U klasi `Krug` i `Pravougaonik` koristimo **konstruktor sa parametrima koji ima podrazumevane vrednosti**. Ovako napisan konstruktor zamenjuje konstruktor bez i sa parametrima. Na primer,

```
Krug(double rr=1) { r=rr; }
```

kreira objekat u memoriji i po želji korisnika postavlja ga u inicijalno stanje. Ukoliko korisnik pri pozivu konstruktora navede stvarni argument, kao na primer:

```
Krug k(5);
```

, tada se inicijalizacija obavlja po želji korisnika, ukoliko pak korisnik ne navede stvarni argument, kao na primer:

```
Krug k;
```

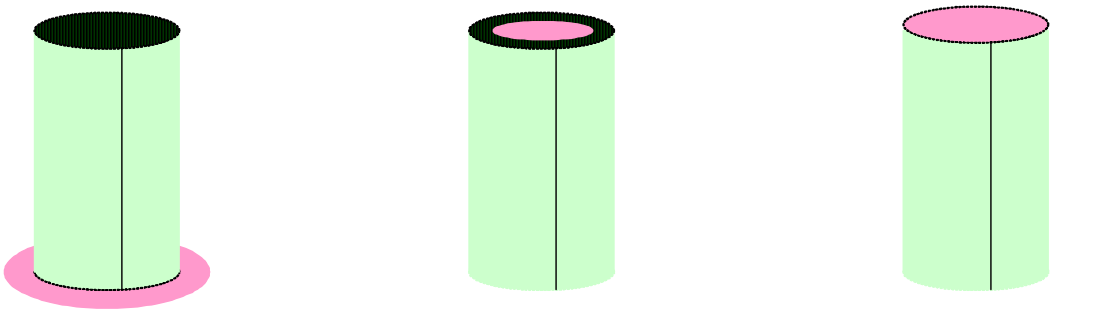
, tada se za postavljanje objekta u inicijalno stanje koristi podrazumevana vrednost.

- Gore opisan **konstruktor sa parametrima koji ima podrazumevane vrednosti** ujedno "obavlja" posao **konstruktora bez parametara** i **konstruktora sa parametrima**. Dakle, isto bi dobili da smo u klasi napisali:

```
Krug() { r=1; }  // konstruktor bez parametara
Krug(double rr) { r=rr; }  // konstruktor sa parametrima
```

- Za računanje površine kruga koristili smo konstantu **`M_PI`**. Da se podsetimo konstanta **`M_PI`** je broj π i ona 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 i zadatak njegovog konstruktora je da postavi objekat u takvo inicijalno stanje da osnova odgovara "savijenom" omotaču valjka, jer samo u tom slučaju možemo reći da je objekat u dozvoljenom stanju. Na slici 4.2 ilustrovane su situacije kada osnova neodgovara i kada odgovara "savijenom" omotaču valjka.



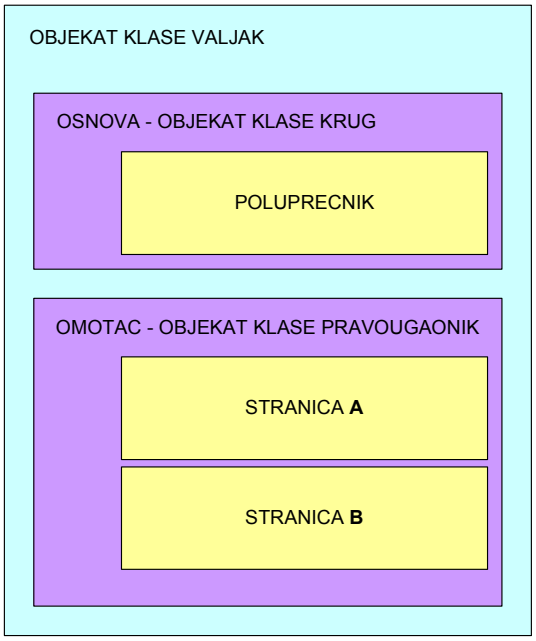
Slika 4.2

- U klasi `Valjak` koristili smo **konstruktor inicijalizator** 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 sa poluprečnikom (`rr`) i visinom (`hh`). Da bi objekat klase `Valjak` došao u dozvoljeno inicijalno stanje, potrebno je da krug koji predstavlja bazu inicijalizujemo poluprečnikom (`rr`), a tada će jedna stranica omotača biti obim kruga (`2*rr*M_PI`), a druga će biti visina (`hh`).

- Na slici 4.3 je ilustrovan izgled objekt klase Valjak.



Slika 4.3

Zadatak 4.3

Napisati klase `JSTrougao` (koja modeluje jednakostranični trougao) i `Pravougaonik`. Napisati klasu `PP3Prizma` koja modeluje pravu pravilnu trostranu prizmu i ima dva objekta-člana: `B` (objekat klase `JSTrougao`) i `M` (objekat klase `Pravougaonik`). Napisati kratak test program.

Vežbe 5

UGRAĐENI KONSTRUKTOR

Do sada smo se upoznali sa konstruktorom bez prametara i konstruktorom sa parametrima. Uloga konstruktora je da kreira objekat i da ga postavi u inicijalno stanje. Postavlja se pitanje – a šta ako ne napišemo konstruktor?

Ugrađeni konstruktor je metoda koja vrši kreiranje objekta, ali ne i njegovu inicijalizaciju.

U svakoj klasi postoji tzv. ugrađeni (default) konstruktor. Ukoliko u klasi ne napišemo konstruktor tada kreiranje objekta vrši ugrađeni konstruktor koji kreira objekat, ali ga ne postavlja u inicijalno stanje. Urađeni konstruktor zauzima potrebnu memoriju za objekat i daje ime objektu, ali ne inicijalizuje njegova polja, tako da ne znamo koje je inicijalno stanje objekta. Na primer, posmatrajmo klasu:

```
class MyClass {
     private:
          int x;
     public:
          void setX(int xx) { x=xx; }
          int getX() const { return x; }
};
```

i posmatrajmo kratak test program:

```
int main() {
     MyClass mc;
     cout<<mc.getX()<<endl;
     return 0;
}
```

, tada je očigledno neizvesno šta će biti ispisano na ekranu, jer u klasi nije napisan konstruktor, pa će objekat kreirati ugrađeni konstruktor, koji će zauzeti potrebnu memoriju i dati ime objektu **`mc`**, ali polje **`x`** u objektu **`mc`** neće biti inicijalizovano, tako da ne znamo koji sadržaj se nalazi unutra.

KONSTRUKTOR KOPIJE

Sada ćemo se upoznati sa konstruktorom kopije.

Konstruktor kopije je metoda koja vrši kreiranje objekta i njegovu inicijalizaciju kopiranjem sadržaja drugog objekta iste klase.

Konstruktor kopije ima formalni argument koji je referenca na objekat iste klase. Na primer:

```
MyClass(const MyClass &m);
```

Na primer, posmatrajmo klasu koja sadrži konstruktor bez parametara, konstruktor sa parametrima i konstruktor kopije:

```
class MyClass {
     private:
          int x;
     public:
          MyClass() { x=5; }
          MyClass(int xx) { x=xx; }
          MyClass(const MyClass &m) { x=m.x; }
          void setX(int xx) { x=xx; }
          int getX() const { return x; }
};
```

i posmatrajmo kratak test program:

```
int main() {
     MyClass mc1;
     MyClass mc2(3);
     MyClass mc3(mc1);
     MyClass mc4(mc2);
     cout<<mc1.getX();
     cout<<mc2.getX();
     cout<<mc3.getX();
     cout<<mc4.getX()<<endl;
     return 0;
}
```

, tada će konstruktor bez parametara kreirati objekat **mc1**, konstruktor sa parametrima objekat **mc2**, a konstruktor kopije objekte **mc3** i **mc4**. Dakle, na ekranu će biti ispisani brojevi **5353**.

Da se podsetimo, kada se u nekoj metodi koristi prenos po referenci, gde referenca predstavlja alternativno ime originala, to znači da se u toj metodi sve vreme radi sa originalom (samo se koristi alternativno ime). Zbog toga je važno da se ispred deklaracije argumenta 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 u gore navedenoj klasi napisali `const MyClass &m`.

ZADATAK ZA VEŽBU

Napisati klasu `Pravougaonik`. Napisati klasu `Kvadar` koja sadrži dva objekta-člana `B` (objekat klase `Pravougaonik`) i `M` (objekat klase `Pravougaonik`). Napisati kratak test program.

PREKLAPANJE METODA

U vežbi broj 2 (pogledajte podnaslov PREKLAPANJE IMENA FUNKCIJA) videli smo da u programskom jeziku C++ postoji mogućnost da postoji više funkcija koje imaju isto ime. Sada ćemo razmotriti preklapanje metoda i najzad preklapanje operatora.

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. Na primer, posmatrajmo klasu:

```
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;
          }
};
```

i posmatrajmo kratak test program:

```
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;
}
```

, tada će prvo biti pozvana prva verzija metode `f(int, int)`, a zatim druga verzija metode `f(double, double)`.

PREKLAPANJE OPERATORA

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

Preklapanje operatora je mogućnost da se za većinu standardnih operatora definiše njihovo ponašanje za slučaj da su operandi klasnih tipova.

Međutim, 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 način:

- metodom
- slobodnom funkcijom (prijateljskom funkcijom)

Sada će preklapanje operatora biti objašnjeno na primeru klase `Complex` koja modeluje kompleksni broj. Klasa `Complex` sadrži dva podatka-člana **`re`** (tipa `double`) i **`im`** (tipa `double`). Definicija klase `Complex` izgleda ovako:

```
#ifndef COMPLEX_DEF
#define COMPLEX_DEF

#include <iostream>
using namespace std;

class Complex {
    private:
        double re;
        double im;
    public:
        Complex();
        Complex(double, double);
        Complex(const Complex&);
        double getRe() const;
        double getIm() const;
        void setRe(double);
        void setIm(double);
        Complex& operator=(const Complex&);
        Complex& operator+=(const Complex&);
        Complex& operator-=(const Complex&);
        Complex& operator*=(const Complex&);
        Complex& operator/=(const Complex&);
        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&);
};

#endif
```

OPERATOR =

Sada ćemo razmotriti preklapanje pojedinih operatora. Operator dodele se preklapa na sledeći način:

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

Operator dodele `=` se preklapa operatorskom metodom. Operatorska metoda kao argument prihvata objekat klase `Complex` po referenci. Smisao preklapanja operatora dodele je da kad napišemo `z1=w1`, gde su `z1` i `z2` objekti klase `Complex`, zapravo se izvršava metoda `operator=` objekata `z1` za stvarni argument `z2`, tj. kao da je napisano `z1.operator=(z2)`.

Vidimo da operatorska metoda ima karakterističnо ime po tome što 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 i koji sadrži adresu datog objekta. Drugim rečima, podatak-član **this** nekog objekta jeste pokazivač koji 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 moglo napisati, na primer, `z1=z2=z3=z4` itd. gde su `z1,z2,z3,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`, tj. izvršiće se metoda `z3.operator=(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`, tj. biće izvršena metoda `z2.operator=(z3)` itd. dok najzad ne bude izvršena dodela `z1=z2`.

OPERATOR +=

Operator += se preklapa na sledeći način:

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

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

OPERATORI -=, *= I /=

Slično se preklapaju i operatori -=, *= i /=. Primera radi, operator *= se preklapa na ovaj način:

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

OPERATOR +

Operator + se preklapa slobodnom funkcijom, tj. prijateljskom funkcijom. Prvo ćemo objasniti šta znači da je neka slobodna funkcija prijatelj klase.

Prijateljska funkcija jeste slobodna funkcija (nije metoda!!!) koja je unutar klase proglašena za prijatelja date klase i time ona ima privilegiju da se unutar nje može pristupati svim članovima te klase (čak i onim koji se nalaze u **private** segmentu).

Neka slobodna funkcija se prograšava prijateljem klase tako što se u klasi navede njen prototip i ispred prototipa navede reč **friend**. U klasi `Complex` je naveden prototip slobodne operatorske funkcije `operator+` i ispred prototipa stoji reč **friend**, odnosno

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

, a njena realizacija izgleda ovako:

```
Complex operator+(const Complex &z1, const Complex &z2) {
    Complex w(z1.re+z2.re, z1.im+z2.im);
    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.re+z2.re` i `z1.im+z2.im`. Posao kreiranja objekta `w` obavlja konstruktor sa parametrima klase `Complex`.

Kao povratnu vrednost, funkcija `operator+` vraća po vrednosti (ne po referenci) objekat `w`. 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. Da se podsetimo, na kraju svake funkcije sve lokalne promenljive i svi lokalni objekti bivaju uništeni. Dakle, nije dobro ako se napiše sledeće:

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

OPERATORI -, * I /

Slično se preklapaju i operatori `-`, `*` i `/`. Primera radi, operator `*` se preklapa na ovaj način:

```
Complex operator*(const Complex &z1, const Complex &z2) {
    double r=z1.re*z2.re - z1.im*z2.im;
    double i=z1.re*z2.im + z1.im*z2.re;
    Complex w(r, i);
    return w;
}
```

OPERATORI ==, I !=

Operatori `==` i `!=` se preklapaju prijateljskim funkcijama. Primera radi, operator `==` se preklapa na ovaj način:

```
bool operator==(const Complex &z1, const Complex &z2) {
    if(z1.re==z2.re && z1.im==z2.im)
        return true;
    else
        return false;
}
```

OPERATOR <<

Operator << se preklapa na ovaj način:

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

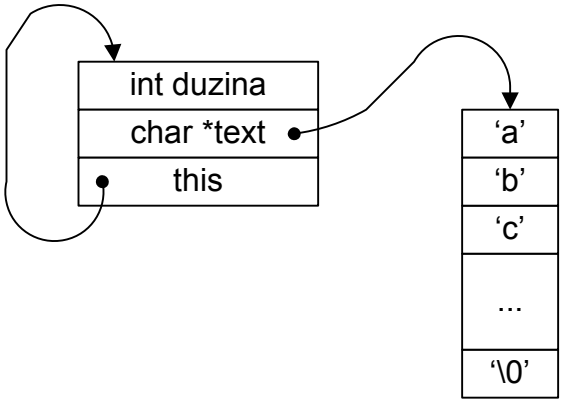
Vežbe 6

KLASA `DinString`

- Definicija klase `DinString` izgleda ovako:

```
#ifndef DSTRING_HPP
#define DSTRING_HPP
#include <iostream>
using namespace std;

class DinString {
private:
            int duzina;
            char *text;
public:
            DinString(){ duzina=0; text=NULL; }
            DinString(const char[]);
            DinString(const DinString&);
            ~DinString() { delete[] text; }
            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 &out, const DinString &s);
            int length()const { return duzina; }
};
#endif
```

- Objekat klase `DinString` sadrži podatak-član `text` koji je pokazivač na tip `char`, podatak-član `duzina` tipa `int` i podatak-član `this` koji pokazuje na sam objekat kojem pripada. Na slici je prikazan objekat klase `DinString`. Pokazivač `text` sadrži adresu memorijskog segmenta u kojem se nalaze elementi tipa `char`. String je vezan za pokazivač `text`, a na kraju stringa se nalazi karakter `NULL`, tj. `'\0'`. Podatak-član `duzina` sadrži broj karaktera u stringu.



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

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

- Konstruktor inicijalizovan stringom `ulazniStr[]` se realizuje na ovaj način:

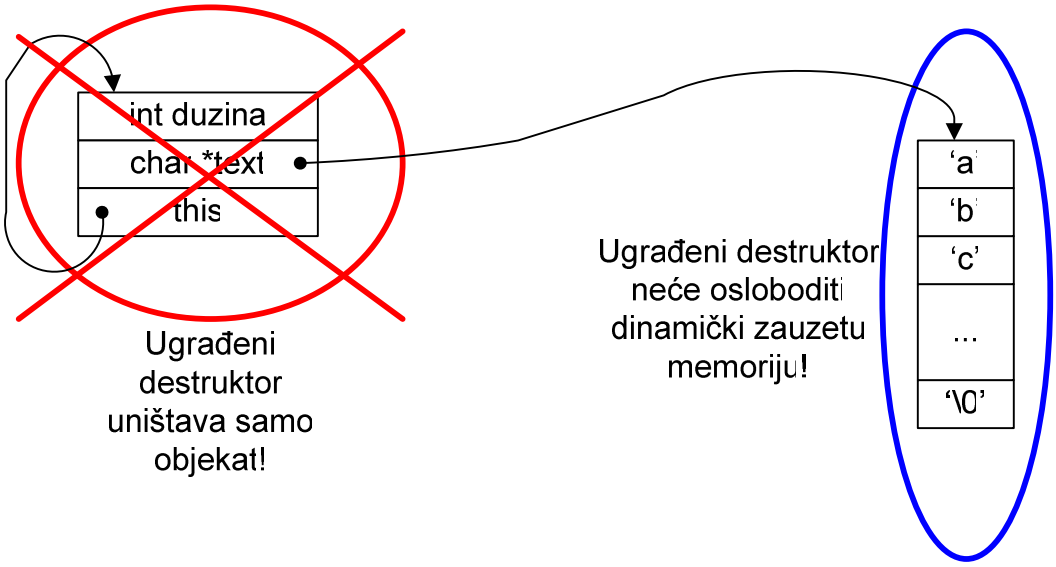
```
DinString::DinString(const char  ulazniStr[]) {
	duzina=0;
	while(ulazniStr[duzina]!='\0'){
		duzina++;
	}
	text=new char[duzina+1];
	for (int i=0;i<duzina;i++)
		text[i]=ulazniStr[i];
	text[duzina]='\0';
}
```

- Operator **`new`** se koristi za dinamičko zauzimanje memorije i vraća adresu od dobijenog memorijskog segmenta.
- Operator **`delete`** služi za oslobađanje dinamički alociranog memorijskog segmenta.
- Konstruktor kopije se realizuje na ovaj način:

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

- U svakoj klasi postoji ugrađeni (*default*) konstruktor i ugrađeni (*default*) destruktor. Ukoliko destruktor nije napisan tada će posao uništavanja objekta obaviti ugrađeni destruktor. Na primer, klasa `Trougao` ili klasa `Semaphore` (koje smo naučili na prethodnim vežbama) nisu imale napisan destruktor, pa je posao uništavanja objekta obavljao ugrađeni konstruktor. Međutim, za klasu `DinString` je obavezno da se napiše destruktor, da bi se sprečilo "curenje memorije". Na slici je ilustrovan efekat uništavanja objekta klase `DinString` pomoću ugrađenog konstruktora, 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 podatak-član `text` (koji je pokazivač na tip char i koji se nalazi u objektu).



- Ako u klasi `DinString` 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 `text`, pa bi na taj način imali program koji "jede" memoriju. Takva pojava je poznata pod imenom "curenje memorije".
- Dakle, možemo usvoji pravilo da **ukoliko u klasi postoji dinamička alokacija memorije, tada je obavezno da se napiše destruktor koji bi oslobodio zauzetu memoriju**.

- Destruktor klase `DinString` se realizuje ovako:

```
~DinString() { delete[] text; }
```

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

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

- 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 prepisana dužina od objekta koji se dodeljuje, zauzeta nova memorija i najzad, prepisani novi elementi. Na kraju se stavlja karakter `NULL`, tj. `'\0'` koji označava kraj stringa. Realizacija operatora dodele izleda ovako:

```
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;
}
```

- 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. Realizacija operatora za spajanje stringova izgleda ovako:

```
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 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. Realizacija operatora za proveru jednakosti izgleda ovako:

```
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;
}
```

- Operator za ispis na ekran se preklapa na ovaj način:

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

- Za ispis na ekran se može pisati `out<<s.text;` jer na kraju se nalazi karakter `NULL`, tj. `'\0'`.

Vežbe 7

NASLEĐIVANJE

- Entitet poseduje određena svojstva i entiteti stoje u nekim odnosima, tj. postoje određene veze između entiteta. Na ovim vežbama proučavamo vezu „izvodi se od“.

Nasleđivanje je veza između klasa koja podrazumeva preuzimanje sadržaja nekih klasa, odnosno klasa-predaka i na taj način, uz mogućnost modifikacije preuzetog sadržaja i dodavanje novog dobija se klasa-potomak.

- Klasa (kojih može biti i više) od koje se preuzima sadržaj naziva se *klasa-predak*, *osnovna klasa*, *klasa davalac* ili *natklasa*.
- Klasa koja prima sadržaj naziva se *klasa-potomak*, *izvedena klasa*, *klasa-primalac* ili *potklasa*.
- Objekti potklase poseduju sve sadržaje, sa ili bez modifikacija, koje poseduju i objekti njihove natklase, a pored toga poseduju i 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**.
- Neka je data klasa A:

```
class A {
        private:
                        …
        protected:
                        …
        public:
                        …
};
```

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

```
                public
class B    :    protected  A {
                public

        private
                        …
        protected:
                        …
        public:
                        …
};
```

- Ponašanje članova natklase u zavisnosti od načina izvođenja (koje 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 *private* članovima natklase može se direktno pristupati u potklasi samo pomoću metoda koje su ili *protected* ili *public* u natklasi.

- Razlika u pristupanju članu klase koji je *private* ili koji je *protected* 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.

- Šta se ne nasleđuje:
  - Konstruktori i destruktori se ne nasleđuju.
  - Prijateljstvo se ne nasleđuje.

- Sada ćemo razmotriti izvođenje klase `JKTrougao` iz klase `Trougao`, a zatim izvođenje klase `JSTrougao` iz klase `JKTrougao`:

```
// 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));
        }
};
```

```
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)
{}
};

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: test.cpp
#include "trougao.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;
}
```

ZADACI ZA VEŽBANJE:

1. Za klasu `Trougao` (u gore navedenom zadatku) isprobati šta se dešava ako su podaci-članovi `private` ili `public`. Zatim, isprobati šta se dešava ako je način izvođenja `private` ili `protected`.
2. Napisati klasu `Pravougaonik`. Iz klase `Pravougaonik` izvesti klasu `Kvadrat`.
3. Napisati klasu `Kvadar`. Iz klase `Kvadar` izvesti klasu `Kocka`.
4. Napisati klasu `Pravougaonik`. Napisati klasu `Kvadar` koja sadrži dva objekta-člana: **`B`** (objekat klase `Pravougaonik`) i **`M`** (objekat klase `Pravougaonik`). Iz klase `Kvadar` izvesti klasu `Kocka`.

Vežbe 8

NASLEĐIVANJE - ZADACI

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

```
// datoteka: pravougaonik.hpp
#ifndef PRAVOUGAONIK_DEF
#define PRAVOUGAONIK_DEF

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: test.cpp
#include "kvadrat.hpp"
#include <iostream>
using namespace std;

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 (podatke-članove i metode) od roditeljske klase `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 na 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 je to klasa `Kvadrat` nasledila, a 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` (svejedno je) i rezultat odgovara površini kvadrata.

6. Napisati klasu `Pravougaonik`. Napisati klasu `Kvadar` koja sadrži dva objekta-člana: **`B`** (objekat klase `Pravougaonik`) i **`M`** (objekat klase `Pravougaonik`). Iz klase `Kvadar` izvesti klasu `Kocka`.

```
// datoteka: pravougaonik.hpp
#ifndef PRAVOUGAONIK_DEF
#define PRAVOUGAONIK_DEF

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: telo.hpp
#ifndef TELO_DEF
#define TELO_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();}
};

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

#endif
```

```
// datoteka: test.cpp
#include "telo.hpp"
#include <iostream>
using namespace std;

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 konstruktoru 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.

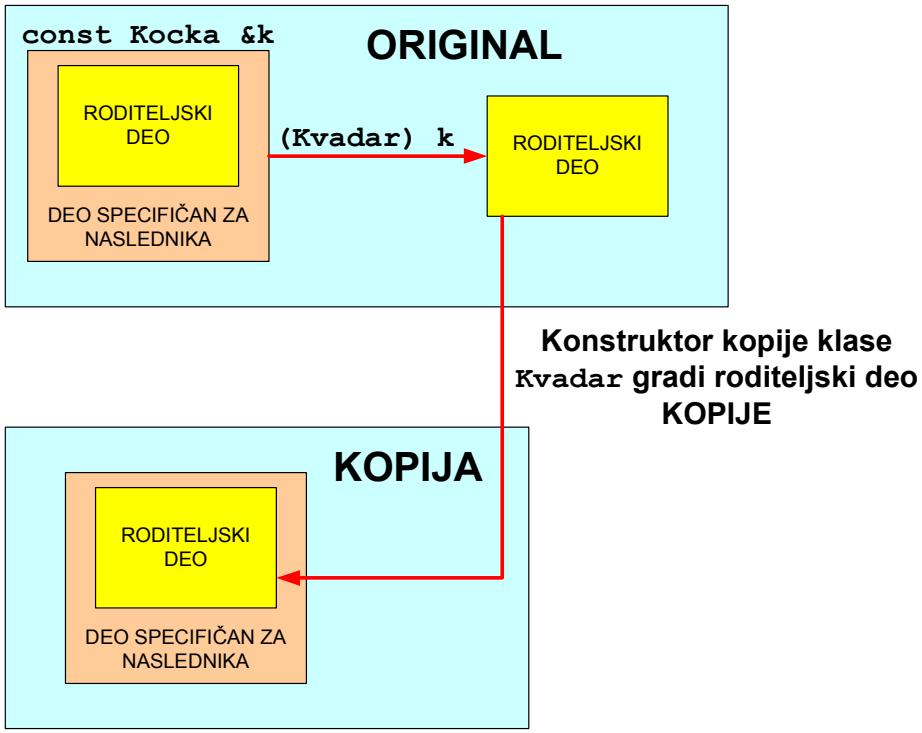
Klasa `Kocka` nasleđuje sve sadržaje od roditeljske klase `Kvadar`. Konstuktor klase `Kocka` "radi po principu" – kocka je kvadar čije su sve ivice jednake. Zbog toga, konstruktor klase `Kocka` izgleda ovako:

```
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 (koji je objekat klase `Kocka`) u klasu `Kvadar` (u konstruktoru je to napisano kao `(Kvadar)k`) kortisti 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 sledećoj slici:



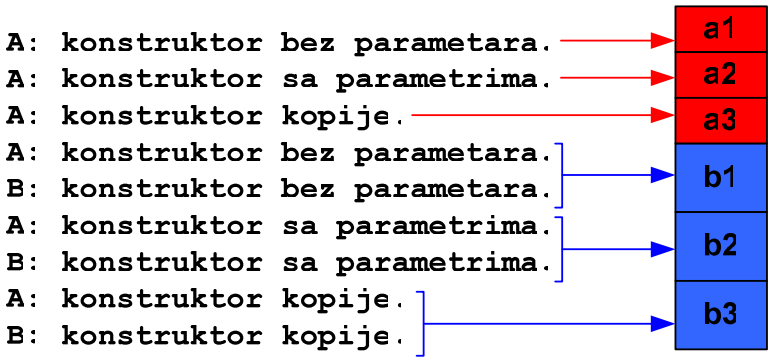
METODE IZVEDENE KLASE

Izvedena klasa (potklasa) nasleđuje sve sadržaje od osnovne klase (natklasa). Sada ćemo razmotriti redosled poziva konstruktora kod izgradnje objekta izvedene klase i redosled poziva destruktora kod uništavanja objekta izvedene klase, kao i slučaj kada se istovremeno u osnovnoj i izvedenoj klasi nalazi ista metoda (metoda koja ima isto ime i istu listu parametara). Razmotrimo sledeći primer:

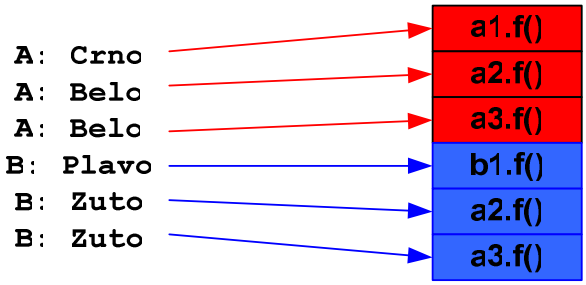
```
// datoteka: klase.hpp
#ifndef KLASE_DEF
#define KLASE_DEF

#include <iostream>
using namespace std;

class A {
    protected:
        int a;
    public:
        A() {
            a=1;
            cout<<"A: konstruktor bez parametara."<<endl;
        }
        A(int aa) {
            a=aa;
            cout<<"A: konstruktor sa parametrima."<<endl;
        }
        A(const A &x) {
            a=x.a;
            cout<<"A: konstruktor kopije."<<endl;
        }
        ~A() {  cout<<"A: destruktor."<<endl; }
        void setA(int aa) { a=aa; }
        int getA() const { return a; }
        void f() {
            if(a>=3)
                cout<<"A: Belo"<<endl;
            else
                cout<<"A: Crno"<<endl;
        }
};



class B : public A {
    public:
        B() : A(1) { cout<<"B: konstruktor bez parametara."<<endl; }
        B(int aa) : A(aa) { cout<<"B: konstruktor sa parametrima."<<endl; }
        B(const B &x) : A((A)x) { cout<<"B: konstruktor kopije."<<endl; }
        void f() {
            if(a>=3)
                cout<<"B: Zuto"<<endl;
            else
                cout<<"B: Plavo"<<endl;
        }
};


#endif
```

```
// datoteka: test.cpp
#include "klase.hpp"

int main() {
    A a1, a2(7), a3(a2);
    B b1, b2(7), b3(b2);
    a1.f();
    a2.f();
    a3.f();
    b1.f();
    b2.f();
    b3.f();
    return 0;
}
```

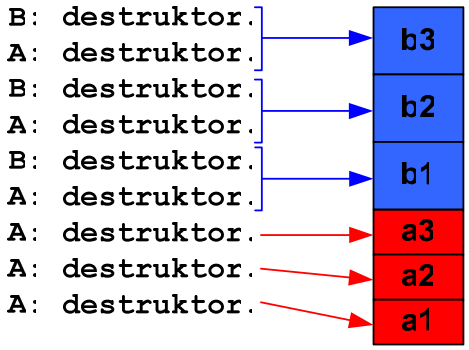
Konstruktori i destruktor u klasama `A` i `B` ispisuju odgovarajuću poruku na ekran. Za izgradnju objekta `a1` biće pozvan konstruktor bez parametara klase `A`, za izgradnju objekta `a2` biće pozvan konstruktor sa parametrima klase `A`, a za izgradnju objekta `a3` biće pozvan konstruktor kopije klase `A`. Za izgradnju objekta klase `b1` biće pozvan konstruktor bez parametara klase `B`, koji podrazumeva da će roditeljski deo objekta izgraditi konstruktor bez parametara klase `A`. Za izgradnju objekta klase `b2` biće pozvan konstruktor sa parametrima klase `B`, koji podrazumeva da će roditeljski deo objekta izgraditi konstruktor sa parametrima klase `A`. Za izgradnju objekta klase `b3` biće pozvan konstruktor kopije klase `B`, koji podrazumeva da će roditeljski deo objekta izgraditi konstruktor kopije klase `A`. Zbog toga ispis konstruktora će biti ovakav:



Zatim, za svaki objekat se poziva metoda `f()`. Metoda `f()`, sa istim imenom i istom listom parametara, postoji i u klasi `A` i u klasi `B`. Objekti klase `B` (objekti `b1`, `b2` i `b3`) sadrži roditeljsku i svoju metodu `f()`, ali će biti pozvana njihova metoda `f()`, pa će ispis na ekran biti ovakav:



Na kraju funkcije `main()` potrebno je uništiti sve postojeće objekte i to je trenutak kada će biti pozvani destruktori. Redosled destruktora je obrnut od redosleda konstruktora, pa će ispis na ekran biti ovakav:



KLASE `Person`, `Student` i `PhDStudent`

Napisati klasu `Person` koja sadrži objekte-članove `ime` i `prezime` (objekti klase `DinString`). Iz klase `Person` izvesti klasu `Student` koja sadrži još i podatak-član `brojIndeksa` (tipa `int`). Iz klase `Student` izvesti klasu `PhDStudent` koja sadrži još i podatak-član `prosečnaOcena` (tipa `double`). Napisati test program i analizirati ispis na ekran.

```
//datoteka: stringd.hpp
#ifndef DSTRING_HPP
#define DSTRING_HPP
#include <iostream>
using namespace std;

class DinString {
private:
       int duzina;
       char *text;
public:
       //konstruktori
       DinString(){
         duzina=0;text=NULL;
         cout<<"DinString: Konstruktor 1."<<endl;
       }
       DinString(const char[]);
       DinString(const DinString&);

       ~DinString() {
         delete[] text;
         cout<<"DinString: Destruktor."<<endl;
       }

       char& operator [](int);
       char operator [](int) const;
       DinString& operator =(const DinString&);
       DinString& operator +=(const DinString&);
```

```
        //slobodne operatorske funkcije
        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 &out, const DinString &s);

        //dodatne metode
        int length()const{
          return duzina;
        }
};

#endif
```

//datoteka: stringd.cpp

```
#include "stringd.hpp"

DinString::DinString(const char  ulazniStr[]) {
        duzina=0;
        //neophodno je utvrditi duzinu obicnog niza karaktera
        while(ulazniStr[duzina]!='\0'){
                duzina++;
        }
        text=new char[duzina+1]; //dinamicka alokacija memorije za smestanje
karaktera
        for (int i=0;i<duzina;i++)
                text[i]=ulazniStr[i];
        text[duzina]='\0';
        cout<<"DinString: Konstruktor 2."<<endl; //ovo samo da bi se prikazalo i
kada se implicitno pozove - videti test
}

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';
        cout<<"DinString: Konstruktor 3."<<endl;
}

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

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

DinString& DinString:: operator =(const DinString& ds) {
        if (this!=&ds){  //obavezna provera da se izbegne unistavanje sadrzaja
ako je neko u kodu napisao a=a
                delete[] text; //faza brisanja trenutnog sadrzaja
                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;
}
```

```
DinString& DinString::operator +=(const DinString& ds) {
	int i;
	char *tempText=new char[duzina+ds.duzina+1]; //napraviti pomocni
placeholder za spajanje nizova
	for (i=0;i<duzina;i++)
		tempText[i]=text[i];  //iskopirati svoj trenutni sadrzaj kao
"donji deo" novog pomocnog niza

	for (i=0;i<ds.duzina;i++) //u nastavku dodati sadrzaj desnog operanda
		tempText[duzina+i]=ds.text[i];

	tempText[duzina+ds.duzina]='\0';  //sve zaciniti null terminatorom
	duzina+=ds.duzina; //azurirati ukupnu duzinu
	delete []text; //obrisati stari sadrzaj
	text=tempText; //prevezati na novu memorijsku lokaciju (ovo nije greska
jer je i tempText dinamicki alociran

	return *this;
}

//slobodne operatorske funkcije
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; //ovo omogucava da se
iskoci kada se prvi put detektuje razlicit karakter na istim pozicijama u dva
niza

	return true; //ako je stiglo ovde onda su isti
}

bool operator !=(const DinString& ds1,const DinString& ds2) {
	if (ds1.duzina!=ds2.duzina)
		return true; //ako su duzine razlicite sigurno nisu isti!

	for (int i=0;i<ds1.duzina;i++)
		if (ds1.text[i]!=ds2.text[i]) return true; //ovo omogucava da se
iskoci kada se prvi put detektuje razlicit karakter na istim pozicijama u dva
niza

	return false; //ako je stiglo ovde onda su isti tj. sigurno nisu
razliciti
}

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 &s) {
	if(s.duzina>0){
```

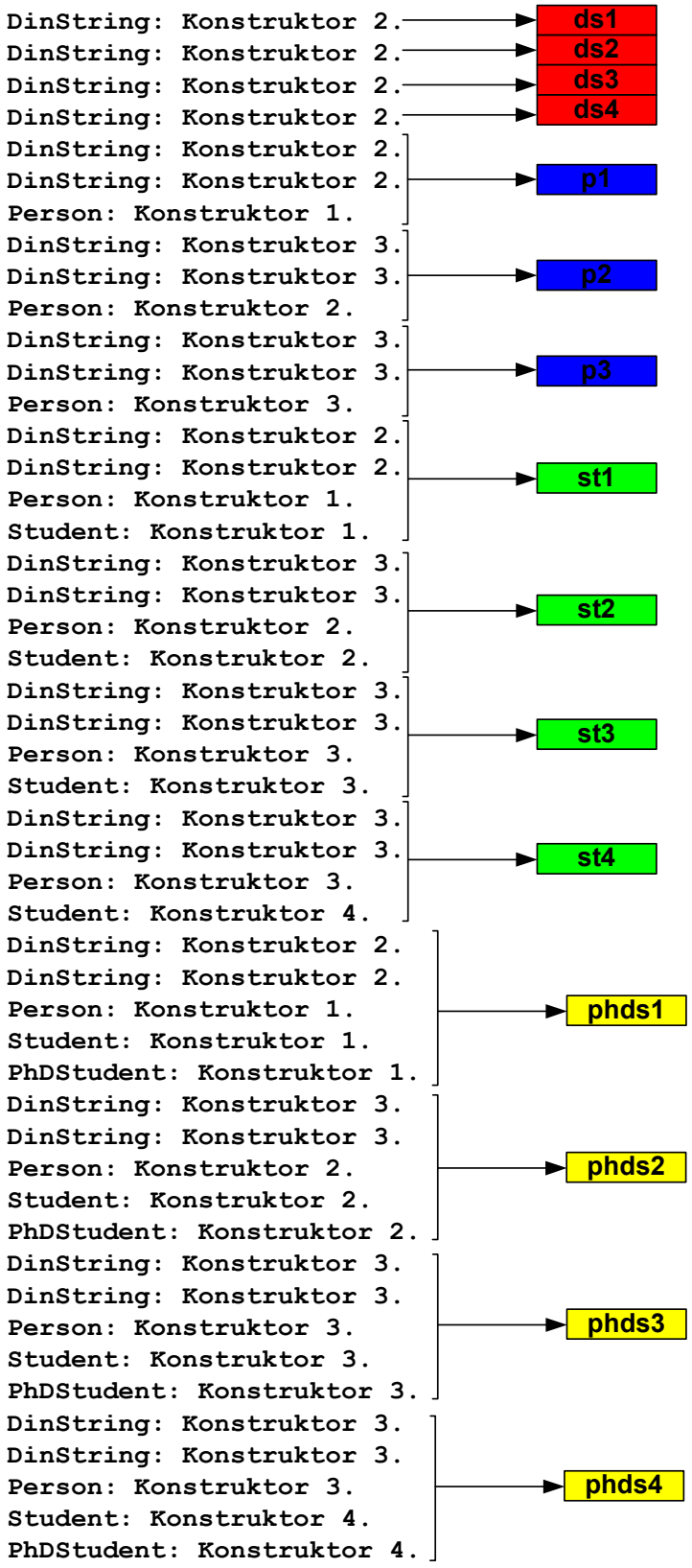
```
            out<<s.text; //ovo moze samo zato sto je null terminiran niz
karaktera inace bi morao ovaj kod u nastavku
            /*
            for(int i=0; i<s.duzina; i++)
            out<<s.text[i];
            */
      }
      return out;
}


//datoteka: person.hpp
#ifndef PERSON_DEF
#define PERSON_DEF

#include "stringd.hpp"
#include <iostream>
using namespace std;

class Person {
    protected:
        DinString ime, prezime;
    public:
        Person(const char *s1="", const char *s2="") : ime(s1), prezime(s2) {
            cout<<"Person: Konstruktor 1."<<endl;
        }
        Person(const DinString &ds1, const DinString &ds2) : ime(ds1),
prezime(ds2) {
            cout<<"Person: Konstruktor 2."<<endl;
        }
        Person(const Person &p) : ime(p.ime), prezime(p.prezime) {
            cout<<"Person: Konstruktor 3."<<endl;
        }
        ~Person() {
            cout<<"Person: Destruktor."<<endl;
        }
        void predstaviSe() const {
            cout<<"Ja sam "<<ime<<" "<<prezime<<"."<<endl;
        }
};

#endif


//datoteka: student.hpp
#ifndef STUDENT_DEF
#define STUDENT_DEF

#include "person.hpp"

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

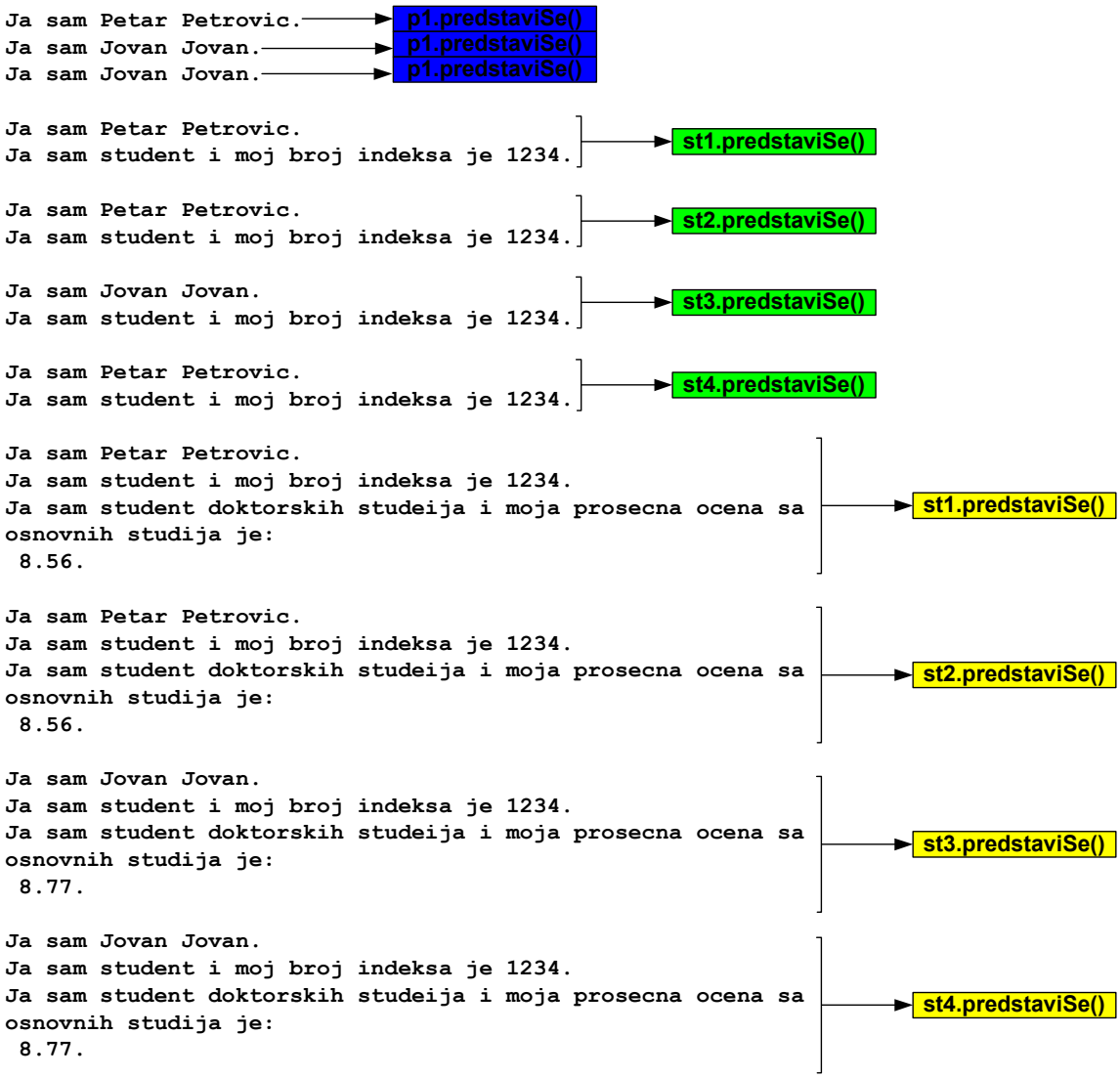
```
        Student(const Person &p, int i) : Person(p), brojIndeksa(i) {
            cout<<"Student: Konstruktor 3."<<endl;
        }
        Student(const Student &s) : Person((Person)s),
brojIndeksa(s.brojIndeksa) {
            cout<<"Student: Konstruktor 4."<<endl;
        }
        ~Student() {
            cout<<"Student: Destruktor."<<endl;
        }
        void predstaviSe() const {
            Person::predstaviSe();
            cout<<"Ja sam student i moj broj 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 Person &p, int i, double po) : Student(p,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<<"Ja sam student doktorskih studeija i moja prosecna ocena sa
osnovnih studija je: "<<prosecnaOcena<<"."<<endl;
        }
};

#endif
```

```
//datoteka: test.cpp
#include "phdstudent.hpp"

int main() {

    const char *s1 = "Petar";
    const char *s2 = "Petrovic";
    const char *s3 = "Jovan";
    const char *s4 = "Jovanovic";


    DinString ds1(s1), ds2(s2), ds3(s3), ds4(s4);


    Person p1(s1,s2), p2(ds3,ds3), p3(p2);


    Student st1(s1,s2,1234), st2(ds1,ds2,1234), st3(p2,1234), st4(st2);


    PhDStudent phds1(s1,s2,1234,8.56), phds2(ds1,ds2,1234,8.56),
phds3(p3,1234,8.77), phds4(st3,8.77);


    p1.predstaviSe();

    p2.predstaviSe();

    p3.predstaviSe();


    st1.predstaviSe();

    st2.predstaviSe();

    st3.predstaviSe();

    st4.predstaviSe();


    phds1.predstaviSe();

    phds2.predstaviSe();

    phds3.predstaviSe();

    phds4.predstaviSe();



    return 0;
}
```

Izvršavanjem programa dobija se sledeći ispis:

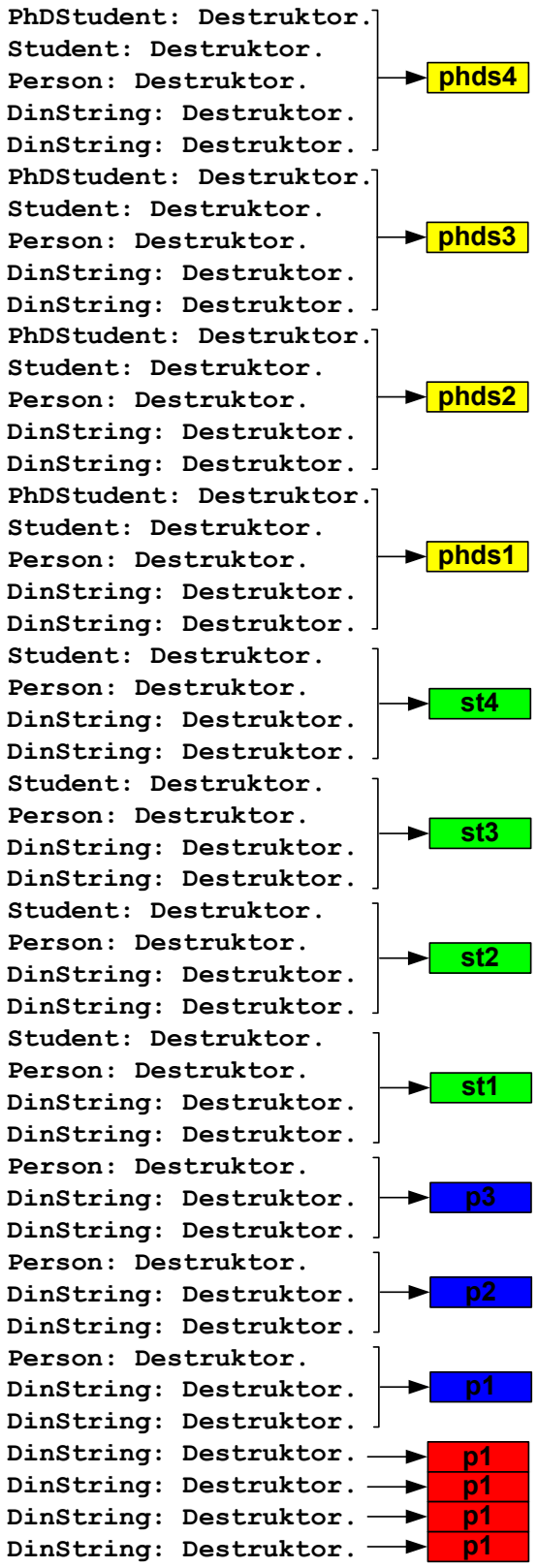




Metoda `predstaviSe()` nalazi se u klasama `Person`, `Student` i `PhDStudent`. Metoda `predstaviSe()` u klasi `Student` sadrži poziv metode `predstaviSe()` od roditeljske klase, tj. klase `Person`. Da bi se pozvala baš roditeljska metoda `predstaviSe()`, tj. od klase `Person`, u klasi `Student` potrebno je pisati ovako:

```
Person::predstaviSe();
```

Na kraju funkcije main() će biti pozvani destruktori koji treba da unište objekte, pa će se dobiti sledeći ispis na ekranu:


PhDStudent: Destruktor.
Student: Destruktor.
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
phds4
PhDStudent: Destruktor.
Student: Destruktor.
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
phds3
PhDStudent: Destruktor.
Student: Destruktor.
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
phds2
PhDStudent: Destruktor.
Student: Destruktor.
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
phds1
Student: Destruktor.
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
st4
Student: Destruktor.
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
st3
Student: Destruktor.
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
st2
Student: Destruktor.
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
st1
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
p3
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
p2
Person: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
p1
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
DinString: Destruktor.
p1
p1
p1
p1

ZADATAK ZA VEŽBU

Iz klase `Person` izvesti klasu `Zaposleni` koja sadrži još i podatke-članove `brojRadnihSati` (tipa `int`) i `vrednostSata` (tipa `double`) i metode `izracunajPlatu()`, koja izračunava i vraća vrednost proizvoda:

`brojRadnihSati*vrednostSata`

i metodu `predstaviSe()`.

Iz klase `Zaposleni` izvesti klasu `Direktor` koja sadrži još i podatak-član `dodatak` (tipa `double`) i metode `izracunajPlatu()`, koja izračunava i vraća vrednost proizvoda:

`(1+dodatak/100)*Zaposleni::izracunajPlatu() )`

i metodu `predstaviSe()`.

Napisati kratak test program.

Vežbe 9

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 virtuelne metoda mora imati isti prototip kao originalna.

Zadatak 9.1

Ramotriti 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: test.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;
}
```

Izvršavanjem programa dobija se 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 "roditeljskih" metoda `f()`. Poziv metoda `f()` za objekte `b1` ili `b2` podrazumeva poziv "potomačkih" metoda `f()`. Međutim, posle 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 "potomačke" metode `f()` i biće ispisana poruka: `Klasa B2: f()`. Razlog za to jeste u tome što je metoda `f()` u klasi `A2` proglašena virtuelnom.

KLASE `Person`, `Student` i `PhDStudent`

Sada ćemo razmotriti slučaj da je u klasi `Person` metoda `predstaviSe()` virtuelna metoda, a zatim ćemo napisati slobodnu funkciju `predstavljanje()` koja kao parametar prihvata po referenci objekat klase `Person`.

```
// datoteka: person.hpp
#ifndef PERSON_DEF
#define PERSON_DEF

#include "stringd.hpp"
#include <iostream>
using namespace std;

class Person {
    protected:
        DinString ime, prezime;
    public:
        Person(const char *s1="", const char *s2="") : ime(s1), prezime(s2) {}
        Person(const DinString &ds1, const DinString &ds2) : ime(ds1),
prezime(ds2) {}
        Person(const Person &p) : ime(p.ime), prezime(p.prezime) {}
        ~Person() {}
        virtual void predstaviSe() const {
            cout<<endl;
            cout<<"Ja sam "<<ime<<" "<<prezime<<"."<<endl;
        }
};

#endif
```

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

#include "person.hpp"

class Student : public Person {
    protected:
        int brojIndeksa;
    public:
        Student(const char *s1="", const char *s2="", int i=0) : Person(s1,s2),
brojIndeksa(i) {}
        Student(const DinString &ds1, const DinString &ds2, int i) :
Person(ds1,ds2), brojIndeksa(i) {}
        Student(const Person &p, int i) : Person(p), brojIndeksa(i) {}
        Student(const Student &s) : Person((Person)s),
brojIndeksa(s.brojIndeksa) {}
        ~Student() {}
        void predstaviSe() const {
            cout<<endl;
            cout<<"Ja sam "<<ime<<" "<<prezime<<".";
            cout<<"Ja sam student i moj broj 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) {}
        PhDStudent(const DinString &ds1, const DinString &ds2, int i, double
po) : Student(ds1,ds2,i), prosecnaOcena(po) {}
        PhDStudent(const Person &p, int i, double po) : Student(p,i),
prosecnaOcena(po) {}
        PhDStudent(const Student &s, double po) : Student(s), prosecnaOcena(po)
{}
        PhDStudent(const PhDStudent &phds) : Student((Student)phds),
prosecnaOcena(phds.prosecnaOcena) {}
        ~PhDStudent() {}
        void predstaviSe() const {
            cout<<endl;
            cout<<"Ja sam "<<ime<<" "<<prezime<<".";
            cout<<"Ja sam student doktorskih studeija i moj broj indeksa je
"<<brojIndeksa;
            cout<<"i moja prosecna ocena sa osnovnih studija je:
"<<prosecnaOcena<<"."<<endl;
        }
};

#endif


// datoteka: test.cpp
#include "phdstudent.hpp"

void predstavljanje(const Person &p) {
    p.predstaviSe();
}

int main() {
    const char *s1 = "Petar";
    const char *s2 = "Petrovic";
    const char *s3 = "Jovan";
    const char *s4 = "Jovanovic";
    DinString ds1(s1), ds2(s2), ds3(s3), ds4(s4);
    Person p1(s1,s2), p2(ds3,ds3), p3(p2);
    Student st1(s1,s2,1234), st2(ds1,ds2,1234), st3(p2,1234), st4(st2);
    PhDStudent phds1(s1,s2,1234,8.56), phds2(ds1,ds2,1234,8.56),
phds3(p3,1234,8.77), phds4(st3,8.77);

    predstavljanje(p1);
    predstavljanje(p2);
    predstavljanje(p3);
    predstavljanje(p3);
    predstavljanje(st1);
    predstavljanje(st2);
    predstavljanje(st3);
    predstavljanje(st4);
    predstavljanje(phds1);
    predstavljanje(phds2);
```

```
    predstavljanje(phds3);
    predstavljanje(phds4);

    return 0;
}
```

APSTRAKTNE KLASE

- **Apstraktna metoda** je virtuelna metoda koja nema telo (tj. nema realizaciju).
- Neka virtuelna metoda se proglašava apstraktnom tako što se na kraju njenog prototipa napiše **=0**, odnosno, opšti oblik prototipa 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 9.2

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; }
```

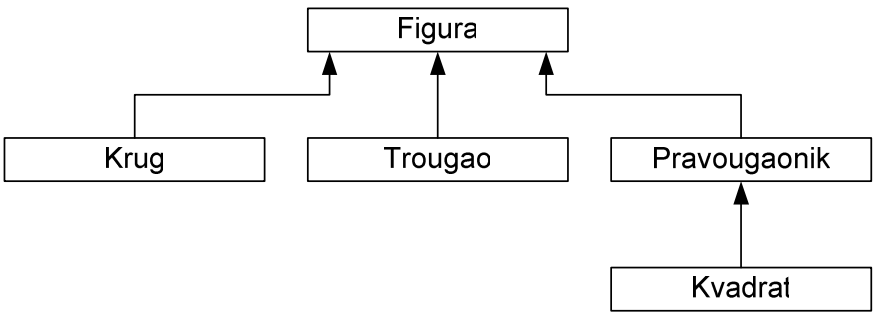
```
        double getO() const { return 2*a + 2*b; }
        double getP() const { return a*b; }
};

#endif



// datoteka: test.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;
}
```

ZAJEDNIČKI ČLANOVI KLASE

- **Zajednički član klase** se dobija tako što se ispred njegove deklaracije navede rezervisana reč **`static`**.
- Zajednički podatak-član je zajednički za sve objekte date klase. Drugim rečima, to znači da u svakom trenutku vrednost tog podatka-člana je jednaka za sve objekte date klase.

Zadatak 9.3

Napisati apstraktnu klasu `Figura` i realizovati sledeću hijerarhiju klasa:



Svaka od klasa treba da sadrži dva **`static`** podatka-člana: `count` i `id`. Podatak-član `count` treba da „broji“ aktuelne objekte u programu, tj. svaki poziv konstruktora će uvećati za jedan podatak-član `count`, a svaki poziv destruktora će smanjiti za jedan podatak-član `count`. Podatak-član `id` jeste identifikator na osnovu kojeg se može zaključiti kojoj klasi dati objekat pripada. Ukoliko je podatak-član `id` jednak `0` radi se o klasi `Figura`, ukoliko je podatak-član `id` jednak `1` radi se o klasi `Krug`, ukoliko je podatak-član `id` jednak `2` radi se o klasi `Trougao`, ukoliko je podatak-član `id` jednak `3` radi se o klasi `Pravougaonik` i najzad, ukoliko je podatak-član `id` jednak `4` radi se o klasi `Kvadrat`.

```
// 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--; }
        int getCount() const { return count; }
        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; }
        Pravougaonik(const Pravougaonik &p) { a=p.a; b=p.b; }
        ~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--; }
```

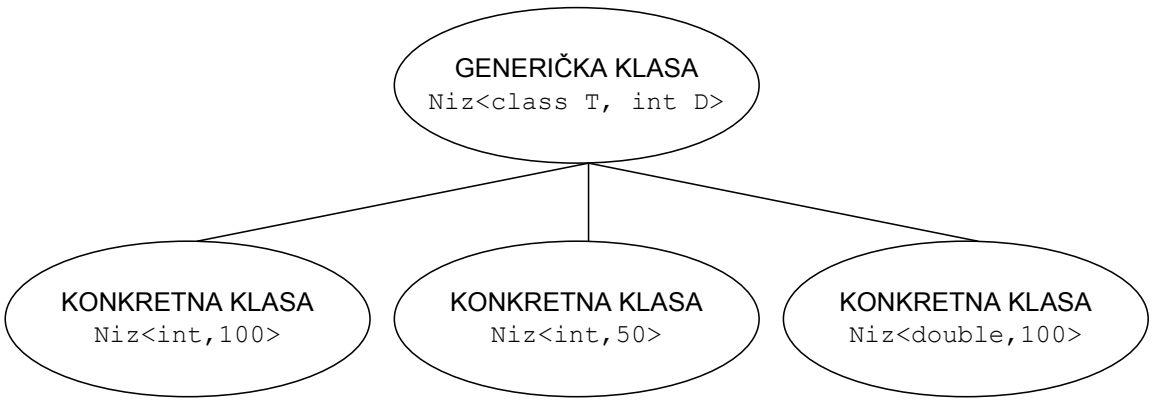
```
        int getCount() const { return count; }
        int getId() const { return id; }
};
#endif

// 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: test.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;
}
```

Vežbe 10

GENERIČKE KLASE

- Razmotrimo jedan primer – posmatrajmo dva niza od 100 elemenata. Elementi prvog niza su tipa `int`, a elementi drugog niza su tipa `double`. Operacije koje definišemo za prvi niz mogli bismo iskoristiti i za drugi niz, s tim da uvek uzimamo u obzir to da 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. Možemo zaključiti da bi bilo jako korisno kada bismo imali mehanizam pomoću kojeg bismo napisali klasu niz za neki tip `T`, a zatim 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, što je ilustrovano na slici:



- Na gornjoj slici 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>`.
- Sada ćemo razmotriti generičku klasu `Pair`.

```
// datoteka: pair.hpp
#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&);
        bool operator==(const Pair<T1,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>
bool Pair<T1,T2>::operator==(const Pair<T1,T2> &p) {
    if((first==p.first) && (second==p.second))
        return true;
    else
        return false;
}

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;
}
// datoteka: test.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(5.4,7.8);
    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 1.

U test programu napraviti konkretizaciju klase `Pair<DinString, DinString>`.

Zadatak 2.

Napsiati šablon klase `Niz<class T, int D>`.

```
// datoteka: niz.hpp
#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>&);
        bool operator==(const Niz<T,D>&);
        bool 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>
bool Niz<T,D>::operator==(const Niz<T,D> &rn) {
    if(brEl!=rn.brEl)
        return false;
    for(int i=0; i<brEl; i++)
        if(el[i]!=rn.el[i])
            return false;
    return true;
}
template <class T, int D>
bool Niz<T,D>::operator!=(const Niz<T,D> &rn) {
    if(brEl!=rn.brEl)
        return true;
    for(int i=0; i<brEl; i++)
        if(el[i]!=rn.el[i])
            return true;
    return false;
}
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;
}


// datoteka: test.cpp
#include "niz.hpp"

int main() {
    Niz<int,10> iNiz;
    iNiz.insertNiz(1);
    iNiz.insertNiz(2);
    iNiz.insertNiz(3);
    iNiz.insertNiz(4);
    iNiz.insertNiz(5);
    iNiz.insertNiz(6);
    iNiz.printNiz();
    return 0;
}
```

Zadatak 3.

U test programu napraviti konkretizaciju klase `Niz<DinString, 10>`.

Vežbe 11

GENERIČKE KLASE - ZADACI

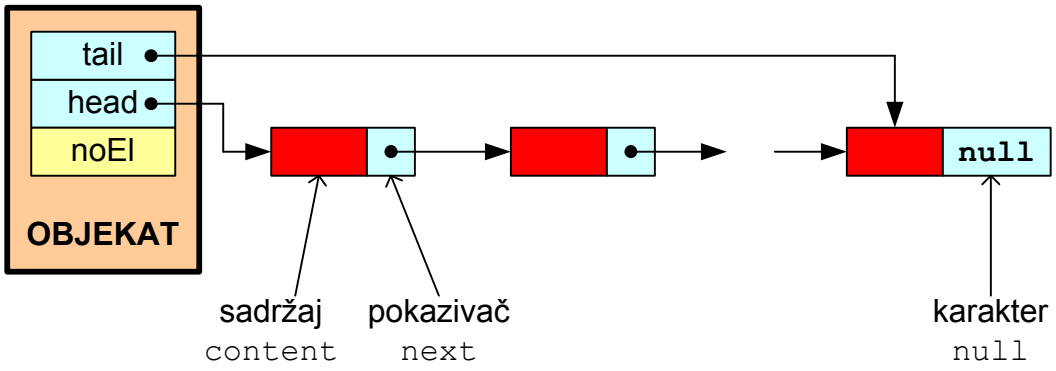
Zadatak 1.

Napisati generičku klasu `List` (jednostrukospregnuta lista). Napisati kratak test program koji demonstrira rad sa jednostrukospregnutom listom objekata klase `Complex`.

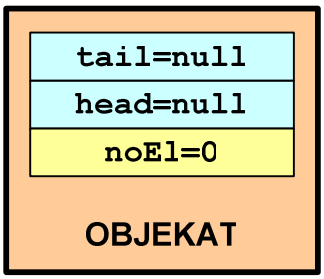
```
// datoteka: list.hpp
#ifndef LIST_DEF
#define LIST_DEF

#include <stdlib.h>
#include <iostream>
using namespace std;

template <class T>
class List{
	private:
		struct listEl{
			T content; //sadrzaj
			struct listEl *next;//pokazivac na sledeci elemenat
		};
		listEl *head;//prvi element liste(glava liste)
		listEl *tail;//nije lose za imati, moze povecati upotrebljivost
strukture
		int noEl;
	public:
		List(){ // konstruktor bez parametara
			head=tail=NULL;
			noEl=0;
		}
		List(const List<T> &);//konstruktor kopije
		List<T>& operator=(const List<T>&);//prekloplen operator =
		virtual ~List();

		int size() const {return noEl;}//broj elemenata u liste
		bool empty() const {return head==NULL?1:0;}

		bool add(int, const T&);
		bool remove(int);//brisanje elementa iz liste
		bool read(int, T&)const; //ocitani element se smesti u T -
funkcija vraca uspelo-neuspelo

		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 sledeće podatke-članove: 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. Sledeća slika prikazuje izgled jednostrukospregnute liste:



Konstruktor bez parametara je realizovan u klasi. Njegov posao je da postavi objekat klase `List` u inicijalno stanje, a to je `head=tail=null` i `noEl=0`. Objekat kojeg je kreirao konstruktor bez parametara izgleda ovako:



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 članu koji je `private`. Za čitanje sadržaja čvora na *i*-toj poziciji koristi se metoda `read()` (koja je `public`). U metodu `read()` prosleđuju se sledeći stvarni parametri: 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 izvršavanje metode `read()`. Operator za ispis na ekran **<<** realizujemo ovako:

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

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

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

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

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

template <class T>
bool List<T>::add(int n, const T& newContent){
	bool isempty=empty();
	if(n<1 || (!isempty && 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){ //novi ide na kraj liste
				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) //znaci da je novi element zakacen na
kraj liste
                          tail=newEl;

                    return true;
             }
      }
}

template <class T>
bool List<T>::remove(int n){
      if(empty() || n<1 || n>noEl)
             return false;
      else{
             if(n==1){
                    listEl *temp=head;
                    head=head->next;
                    delete temp;
                    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) //znaci brise se poslednji
                          tail=temp; //setovati tail na trenutni pretposlednji
element
                    delete del;
                    noEl--;
             }
             return true;
      }
}

template <class T>
bool List<T>::read(int n,T& retVal)const{
      if(empty() || 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: complex.hpp
#ifndef COMPLEX_HPP_INCLUDED
#define COMPLEX_HPP_INCLUDED

#include <iostream>
using namespace std;

class Complex{
private:
    double real, imag;

public:
    Complex( double r=0, double i=0){
        real = r;
        imag = i;
    }

    Complex( const Complex &c){
        real = c.real;
        imag = c.imag;
    }

    double getReal() const { return real;}
    double getImag() const { return imag;}
    void setReal( double r) { real=r;}
    void setImag( double i) { imag=i;}

    Complex &operator+=( const Complex &);
    Complex &operator-=( const Complex &);
    Complex &operator*=( const Complex &);
    Complex &operator/=( const Complex &);

    Complex &operator=( const Complex &);

    friend bool 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 Complex operator/( const Complex &, const Complex &);

    friend ostream &operator<<( ostream &, const Complex &);
};

#endif // COMPLEX_HPP_INCLUDED
```

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

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

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

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

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

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

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);
    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;
}

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;
}
```

```
// datoteka: main.cpp
#include <iostream>
#include "list.hpp"
#include "complex.hpp"
using namespace std;

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 2.

Iz klase `List` izvesti klasu `LinkedQueue` (red ili FIFO memorija). Napisati kratak test program.

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

#include "list.hpp"

template <class T>
class LinkedQueue; //prototip genericke klase LinkedQueue

template <class T>
void printOut(const LinkedQueue<T> &); //prototip slobodne friend-funkcije

template <class T>
class LinkedQueue: protected List <T> {
public:
       LinkedQueue(){};
       bool readFromQueue(T&)const;
```

```
        void removeFromQueue() {List<T>::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> &l){
        cout<<endl;
        cout<<"\tVelicina reda: "<<l.size()<<endl;
        cout<<"\tSadrzaj reda je: ";
        T retVal;
        for(int i=1;i<=l.size();i++){
                if(i>1) cout<<", ";
                l.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: ilt.cpp
#include "queue_lnk.hpp"
#include <iostream>
using namespace std;

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<<endl;
	else
		cout<<"Citanje nije obavljeno - Prazan red!!!"<<endl<<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<<endl;
	else
		cout<<"Citanje nije obavljeno - Prazan red!!!"<<endl<<endl;


	cout<<"--------------------"<<endl;
	cout<<"Dodavanje u red broja 100"<<endl;
	a.addToQueue(100);
	cout<<"Queue: ";
	printOut(a);

	return 0;

}
```

Zadatak 3.

Realizovati klasu `SpisakStudenata` koja modeluje spisak studenata za oglasnu tablu koji sadrži naziv ispita, datum polaganja, brojeve indeksa, imena i prezimena i ocene. Koristiti klasu `DinString` i generičku klasu `List`.

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

#include "stringd.hpp"

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

    public:
        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<<"   "<<s.ime<<"   "<<s.prezime<<"   "<<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 naziv;
        DinString datumPolaganja;
        List<Student> lista;
    public:
        SpisakStudenata() {}
        void setNaziv(const DinString &ds) { naziv=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;
    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<<"*************************************************************"<<endl;
    cout<<st<<endl;

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: "<<naziv<<endl;
    cout<<" ---------------------- REZULTATI ISPITA ----------------------
"<<endl;
    Student st;
    for(int i=1; i<=lista.size(); i++) {
        lista.read(i,st);
        cout<<st<<endl;
    }
    cout<<" -----------------------------------------------------
KATEDRA"<<endl;
}
```

```
// datoteka: test.cpp

#include "spisak.hpp"

int main() {
    char odg='d';
    char *s;
    SpisakStudenata spisak;

cout<<"*************************************************************"<<endl;
    cout<<"Dobrodosli u program za pravljenje spiska studenata..."<<endl;

cout<<"*************************************************************\n\n\n\n\n\n
"<<endl;
    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 zelite da spisak bude sortiran po ocenama? [D/N] ...  ";
    cin>>odg;
    if((odg=='d')||(odg=='D'))
        spisak.sortPoOceni();
    else
    {
        cout<<"Da li zelite 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: 29.05.2009.
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: 29.05.2009.
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
 ------------------------------------------------------ KATEDRA
```

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

```
Datum: 29.05.2009.
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
 ------------------------------------------------------ KATEDRA
```