OOP 2
PREDAVANJE 2

Prof. dr Borivoje Milošević

2018.

Strukture

- Struktura je skup različitih podataka u jednoj jedinstvenoj jedinici (delu) memorije.
- Strukture se deklarišu pomoću ključne reči **struct, i predsdtavljaju MASKU (template) za svoje buduće objekte - instance**:

  - **Deklaracija strukture:**

```
struct Konto
{
char Kontotip [20];
int KontoBroj;
int TajniBroj;
float SaldoKonta;
                    };
```

Strukture

- Promenljive unutar strukture se nazivaju elementi strukture. Svaki se element strukture mora deklarisati kao jedna promenljiva u nekom bloku naredbi. Elementi strukture u gornjem primeru su:

  - **char Kontotip [20];** //polje čiji je tip podataka **char**
  - **int KontoBroj;** //dve promenljive tipa **int**
  - **int TajniBroj;**
  - **float SaldoKonta; //** jedna promenljiva tipa **float**.

- Posle deklaracije strukture kreiraju se njene instance strukture. One se dodaju kao promenljive između zatvorene zagrade i tačke zareza na kraju strukture.

Strukture

- **Strukture omogućavaju jednostavno modeliranje praktično svih tipova podataka koji se susreću u realnom životu. Međutim, one definišu samo *podatke* kojima se opisuje neka pojava (npr. neki student), ali ne i *postupke* koji se mogu primeniti nad tom pojavom.**

- **Na taj način, strukture su na izvestan način *preslobodne*, u smislu da se nad njihovim poljima mogu nezavisno izvoditi sve operacije koje su dozvoljene sa tipom podataka koji polje predstavlja, bez obzira da li ta operacija ima smisla nad strukturom *kao celinom*. Na primer, neka je definisana struktura nazvana "Datum", koja se sastoji od tri polja (atributa) nazvana "dan", "mesec" i "godina":**

```
struct Datum {
int dan, mesec, godina;
};
```

- **Polja "dan", "mesec" i "godina" očigledno su namenjena da čuvaju dan, mesec i godinu koji čine neki stvarni datum.**

Strukture

- **Međutim, kako su ova polja praktično obične celobrojne promenljive, ništa nas ne sprečava da napišemo nešto kao:**

  - **Datum d;**
  - **d.dan = 35; //pridruživanje objektu - instanci d strukture Datum**
  - **d.mesec = 14; //stvarnih vrednosti operatorom “. “**
  - **d.godina = 2004;**

  bez obzira što je datum 35. 14. 2004. očigledno lišen svakog smisla.

- **Međutim, kako se “d.dan”, “d.mesec” i “d.godina” ponašaju kao obične celobrojne promenljive, ne postoji nikakav mehanizam koji bi nas sprečio da ovakvu dodelu učinimo.**

- **Sa aspekta izvršavanja operacija, ne vidi se da “dan”, “mesec” i “godina” predstavljaju deo jedne nerazdvojive celine, koji se ne mogu postavljati na proizvoljan način, i nezavisno jedan od drugog.**

Strukture

- **Razmotrimo još jedan primer koji ilustruje “opasnost” rada sa strukturama, zbog preslobodne mogućnosti manipulacije sa njenim poljima. Neka je, na primer, data sledeća struktura:**

```
struct Student {
char ime[30], prezime[30];
int indeks;
int ocene[50];
double prosek;
};
```

- **Očigledno je da bi polje “prosek” trebalo da bude “vezano” za polje “ocene”. Međutim, ništa nas ne sprečava da sve ocene nekog studenta postavimo npr. na 6, a prosek na 10. Ova polja se ponašaju kao da su potpuno odvojena jedna od drugog, a ne tesno povezane komponente jedne celine. Jedna od mogućnosti kojima možemo *delimično* rešiti ovaj problem je da definišemo izvesne funkcije koje će pristupati poljima strukture na strogo kontrolisan način.**

Strukture

- **Na primer, mogli bi postaviti funkciju “PostaviDatum” koja bi postavljala polja “dan”, “mesec” i “godina” unutar strukture koja se prenosi kao prvi parametar u funkciju na vrednosti zadane drugim, trećim i četvrtim parametrom. Pri tome bi funkcija mogla proveriti smislenost parametara i preduzeti neku akciju u slučaju da oni nisu odgovarajući (npr. ukloniti izuzetak).**

- **Na primer, takva funkcija bi mogla izgledati ovako (provera da li je godina prestupna izvedena je u skladu sa gregorijanskim kalendarom):**

```
void PostaviDatum (Datum &d, int dan, int mesec, int godina)
{
int broj dana[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
if(godina % 4 == 0 && godina % 100 != 0 || godina % 400 == 0)
broj dana[1]++;
if(godina < 1 || mesec < 1 || mesec > 12 || dan < 1 || dan >
broj dana[mesec – 1])
throw "Neispravan datum!\n";
d.dan = dan; d.mesec = mesec; d.godina = godina;
}
```

Структуре

Појам структуре

- Изведени тип података
- Може бити веома комплексан тип
- Колекција променљивих, међусобно истог или различитог типа, које су логички повезане у програму
- Колекција променљивих, повезаних због једноставнијег приступа
- Тип искоришћен за развој апстрактног типа - класе

Структуре

Елементи структуре

- Чланови структуре
- Заједничко име - структуре којој припадају
- Сваки члан има своје јединствено име
- Типови – могу бити међусобно различити (разлика у односу на елементе низова)
- Могу бити:
  - променљиве основног типа
  - променљиве показивачи
  - низови променљивих
  - друге структуре...

Структуре

Чланови структуре

- Пример 1. Структура типа **Adresa**:
  - ime_ulice
  - broj_ulaza
  - broj_stana…
- Пример 2. Структура типа **Stan**:
  - Adresa
  - kvadratura
  - broj_soba
  - cena…

Чланови структура - произвољни С типови

Структуре

Приступ члановима структуре

- Различите технике (помоћу показивача и без њих)
- Неопходни:
  - Дефинисани тип структуре
  - Структура у меморији дефинисаног типа
- Приступ без показивача на структуру, помоћу:
  - имена структуре и
  - имена члана структуре
- Приступ са показивачем на структуру, помоћу:
  - имена показивача и
  - имена члана структуре

Структуре

Општи облик дефиниције типа структуре

```
struct ime_tipa {
    tip1 ime_promenljive1;
    tip2 ime_promenljive2;
    ...
    };
```

Дефиниција структуре
не резервише простор у меморији

Структуре

Општи облик дефиниције типа структуре

резервисана"C" реч struct имплицира да ће се користити структура

```
struct ime_tipa {
    tip1 ime_promenljive1;
    tip2 ime_promenljive2;
    ...
    };
```

Структуре

Општи облик дефиниције типа структуре

"C" идентификатор ознака структуре (tag)

```
struct ime_tipa {
    tip1 ime_promenljive1;
    tip2 ime_promenljive2;
    ...
    };
```

Структуре

Општи облик дефиниције типа структуре

```
struct ime_tipa {
    tip1 ime_promenljive1;
    tip2 ime_promenljive2;
    ...
    };
```

листа декларације чланова структуре уоквирена заградама { }

Структуре

Општи облик дефиниције типа структуре

```
struct ime_tipa {
    tip1 ime1, ime2;
    tip2 ime3, ime4, ime5;
    ...
};
```

дефиниција типа структуре обавезно се завршава знаком ;

Структуре

Декларација структурних променљивих дефинисаног типа (1. начин)

```
struct ime_tipa {
    tip1 ime2;
    tip2 ime2;
    }ime_strukturne_promenljive;
```

Декларација структурне променљиве унутар дефиниције типа структуре

Пример

```
struct tačka {
int x,y;
} gornjaDesna, donjaDesna;
```

U gornjem primeru se deklariše struktura **tačka** i stvaraju dve instance strukture:

gornjaDesna, donjaDesna.

Пример

```
struct Structure1 {
char c;
int i;
float f;
double d;
};
int main() {
struct Structure1 s1, s2;
s1.c = 'a'; // Selektovanje ;lana korišćenjem operatora '.'
s1.i = 1;
s1.f = 3.14;
s1.d = 0.00093;
s2.c = 'a';
s2.i = 1;
s2.f = 3.14;
s2.d = 0.00093;
}
```

Структуре

Декларација структурних променљивих дефинисаног типа (2. начин)

```
struct ime_tipa {
    tip1 ime2;
    tip2 ime2;
    };
. . .
struct ime_tipa ime_strukturne_promenljive;
```

Декларација структурне променљиве ван дефиниције типа структуре

Структуре

```
struct Konto
{
char Kontotip [20];
int KontoBroj;
int TajniBroj;
float SaldoKonta;
};
```

Struktura se može generisati i na drugi način. Dovoljno je posle naziva strukture navesti u glavnom programu naziv njene instance:

```
main(){……..
```

– **Konto prvi;**

```
……..}
```

- Ta naredba rezerviše memoriju za strukturu Konto i dodeljuje joj promenljivu – instancu ili objekat sa imenom prvi.
- Posle stvaranja instance strukture mogu se elementima srukture (varijablama) pridruživati vrednosti:

```
strcpy(prvi.Kontotip, “Žiro“);          // Za pridruživanje vrednosti
                                        // koristi se funkcija strcpy!
prvi.KontoBroj = 43212345;
prvi.TajniBroj= 1212;
prvi.SaldoKonta = 34.67;
```

ПРИМЕР:

```
#include <iostream.h>
struct Zaposleni // Deklaracija strukture
{
char ime[20];       //Deklaracija polja char sa dvadeset znakova
char prezime[20];
char adresa[50];
char grad[20];
int BrojPoste;
};
void main()
{
struct Zaposleni slog1; // Instanca strukture Zaposleni, a njen naziv je
                        // slog1
strcpy(slog1.ime, "Branкo");      //Dodeljivanje odgovarajućih vrednosti
strcpy(slog1.prezime, "Mitić");   //članovima instance slog1 strukture
strcpy(slog1.adresa, “Niška 112");  //Zaposleni
strcpy(slog1.grad, “Beograd");
slog1.BrojPoste = 11000;
```

Структуре

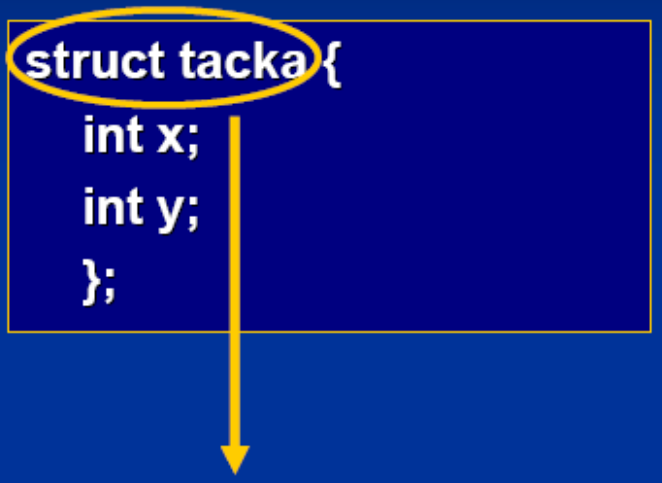
Избор начина декларације структура

- Декларација унутар дефиниције – само када се структурна променљива (или више њих истог типа) уводи на истом месту у програму
- Декларација ван дефиниције чешћа - због тога што је:
  - на једном месту у програму (најчешће глобално) дефинисан тип структуре
  - на произвољном броју места у програму уводе се и користе структурне променљиве истог типа
- Термин за структурну променљиву: **структура**

Структуре

Пример дефиниције једног типа структуре

```
struct tacka {
    int x;
    int y;
};
```



Дефиниција уводи тип структуре tacka који има два члана: x и y, сваки типа int

Структуре

Декларација једне структуре дефинисаног типа (1. начин)

Декларација унутар дефиниције типа структуре:

```
struct tacka {
    int x;
    int y;
    }t1;
```

t1 је структура типа tacka,
која садржи два члана x и y, сваки типа int

Структуре

Декларација више структура дефинисаног типа (1. начин)

Декларација унутар дефиниције типа структуре:

```
struct tacka {
    int x;
    int y;
    }t1, t2;
```

t1 и t2 су структуре типа tacka,
свака од њих садржи чланове x и y, сваки типа int

Структуре

Декларација више структура дефинисаног типа (2. начин)

Декларација ван дефиниције типа структуре:

```
struct tacka {
    int x;
    int y;
    };
. . .
struct tacka t1, t2;
```

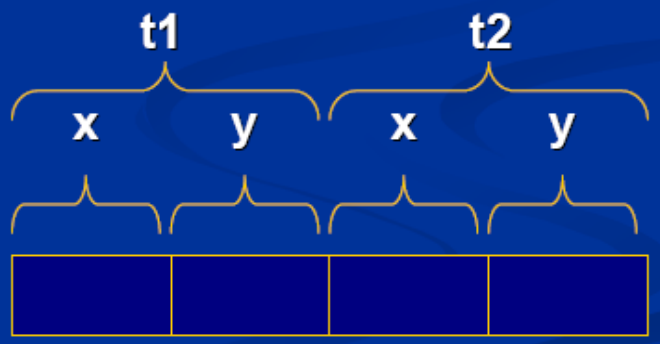
t1 и t2 су структуре типа tacka, свака од њих садржи чланове x и y, сваки типа int

Структуре

Декларација више структура дефинисаног типа (2. начин)

Декларација структура t1 и t2 типа tacka:

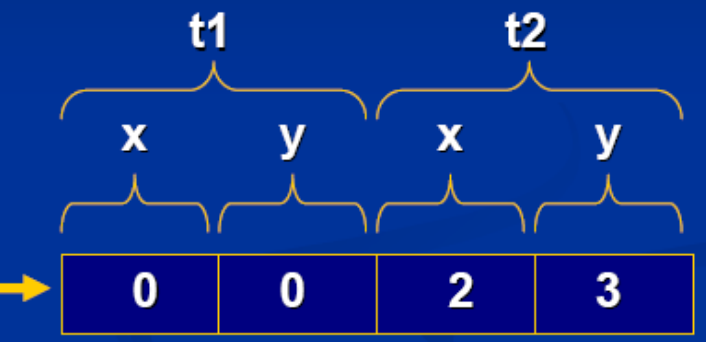
```
struct tacka {
    int x;
    int y;
    };
.   .    .
struct tacka t1, t2;
```



Структуре

Иницијализација декларисаних структура унутар дефиниције типа (1. начин)

```
struct tacka {
    int x;
    int y;
    }t1 = {0, 0}, t2 = {2, 3};
```



Чланови структура t1 и t2 добијају вредности

Структуре

Иницијализација декларисаних структура ван дефиниције типа (2. начин)

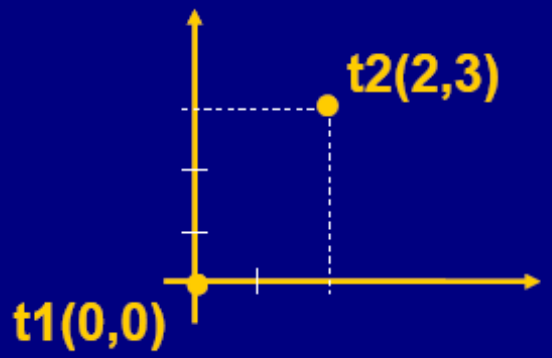
```
struct tacka {
    int x;
    int y;
    };
struct tacka t1={0, 0}, t2 = {2, 3};
```

Резултат у меморији је исти, као када се иницијализација реализује на 1. начин

Структуре

Пример примене структура дефинисаног типа tacka

```
struct tacka {
    int x, y;
    };

struct tacka t1 = {0, 0};
struct tacka t2 = {2, 3};
```



Структуре

Приступ члановима структуре помоћу оператора "тачка" (.)

---

ime_strukturne_promenljive.ime_clana

За оператор приступа члану структуре користи се као симбол знак тачка (.)

~~**ime_clana**~~

Било ком члану структуре не може се приступити само помоћу имена члана (јер је скривен у структури)

Структуре

Приступ члановима структуре помоћу оператора тачка (.)

```
struct tacka {
        int x;
        int y;      };

struct tacka t1;
```

```
t1.x = 0;
t1.y = 0;
```

```
t1.x = t1.y = 0;
```

Структуре

Додела садржаја једне структуре другој структури истог типа

```
struct tacka {
        int x;
        int y;      };
struct tacka t1, t2;

t1.x = 0;
t1.y = 0;
t2.x = t1.x;
t2.y = t1.y;
```

```
t2 = t1;
```

Структуре

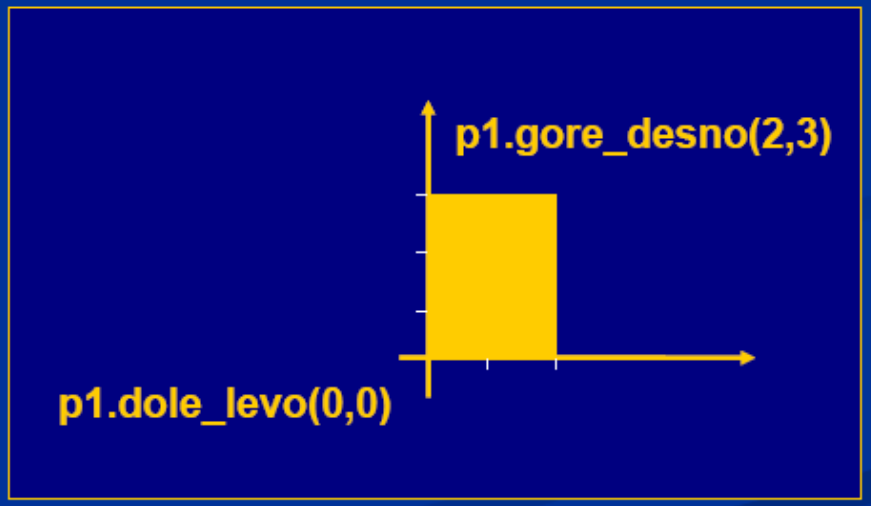
Комплексне (угњеждене) структуре

```
struct tacka {
        int x;
        int y;      };

struct pravougaonik {
        struct tacka dole_levo;
        struct tacka gore_desno;   };

   struct pravougaonik p1;
```

Структуре

Пример примене структура дефинисаног типа tacka

```
p1.dole_levo.x = 0;
p1.dole_levo.y = 0;
p1.gore_desno.x = 2;
p1.gore_desno.y = 3;
```

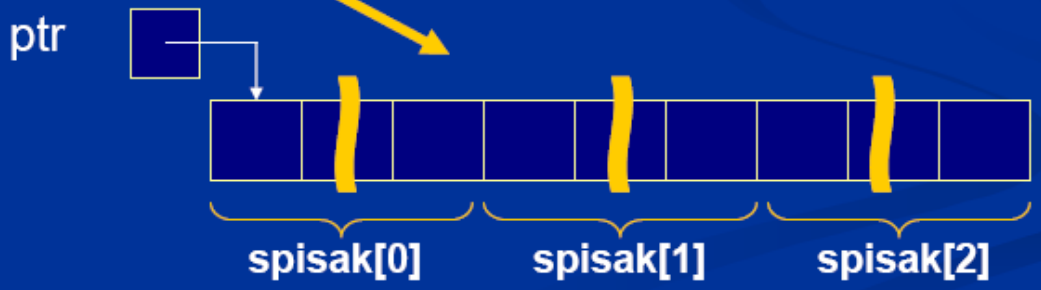


Структуре

Показивач – на низ структура

Struktura spisak[ ] tipa adresa

```
struct  adresa{
          char ime_ulice[31];
          int broj_ulaza;
          } spisak[ ]  =  {“Ime1”, 1,
                           “Ime2”, 2,
                           “Ime3”, 3        };
struct adresa *ptr;
ptr = spisak;                        /* isto sto i:      ptr = &spisak[0] */
```



Strukture

- Članovi strukture mogu imati bilo koji tip podataka osim tipa viod, nekompletnog tipa ili funkcije.
- U C++ mogu se graditi u ugnježdene strukture kao na primeru:

```
struct a {    int x;
struct b    {   int y;  } var2; } var1;
```

ugnježdena struktura

Strukture

- **Primer:**
- Ovaj primer ilustruje deklaraciju strukture:
- struct *zaposleni* /* Definiše strukturu nazvanu temp tipa zaposleni */

```
{   char ime[20];
        int id;
            long klasa; } temp;
```

- Struktura *zaposleni* poseduje tri člana: ime, id i klasa. Identifikator *zaposleni* je identifikator strukture:

  struct *zaposleni* student, fakultet, odsek;

- Predhodni primer definiše tri strukturne promenljive:

  student, fakultet, odsek, a svaka od njih poseduje listu od po tri člana definisanu u primeru na početku.

Strukture

- **Primer 1**

```
struct PERSON // Deklariše PERSON tip strukture
{ int age; // Deklariše tip člana strukture
      long ss;
      float weight;
      char name[25]; }
family_member; // Deklariše objekat tipa PERSON

struct PERSON sister; // C stil deklaracije strukture
PERSON brother; // C++ stil deklaracije strukture
sister.age = 13; // dodeljivanje vrednosti članovima
brother.age = 7; //strukture
```

Strukture

- **Primer 3:**
- struct sample /* Definiše strukturu imena x */

```
{   char c;
        float *pf;
        struct sample *next; } x;
```

Prva dva člana stukture su tipa character i tipa ukazivača na vrednost float. Treći član next je delkarisan kao ukazivač na tip strukture definisane u (sample)

Strukture

struct A

```
{ int i, j, k; };
int f( );
void g( );
int f( ) { return i + j + k; }
void g( ) { i = j = k = 0; }
```

class B

```
{    private:
int i, j, k;
public:
int f( );
void g( );    };
int B::f( ) { return i + j + k; }
void B::g( ) { i = j = k = 0; }
```

COMPLEX i tip

- struct complex8 { float real, imag; };
- struct complex16 { double real, imag; };

- Fortran **LOGICAL*2** se pamti kao 1-byte čija je indikatorska vrednost (1=true, 0=false) praćena sa jednim neiskorišćenim byte_om. Fortran **LOGICAL*4** se pamti kao 1-byte indikatorske vrednosti praćena sa tri neiskorišćena byte_a..

Selekcija članova strukture

- Selekcija članova strukture ukazuje na članove strukture a svaki izraz selekcije ima vrednost i tip selektovanog člana.
- **Sintaksa** prikazana u obliku dva načina pristupa:

postfix-expression . identifier
postfix-expression –> identifier

-Prva forma reprezentuje vrednost tipa strukture dok identifier imenuje člana specificirane strukture

-Druga forma reprezentuje ukazivač na strukturu dok identifier imenuje člana specificirane strukture

- *U ovom slučaju moramo znati da je izraz sa operatorom* ***–>*** *:*

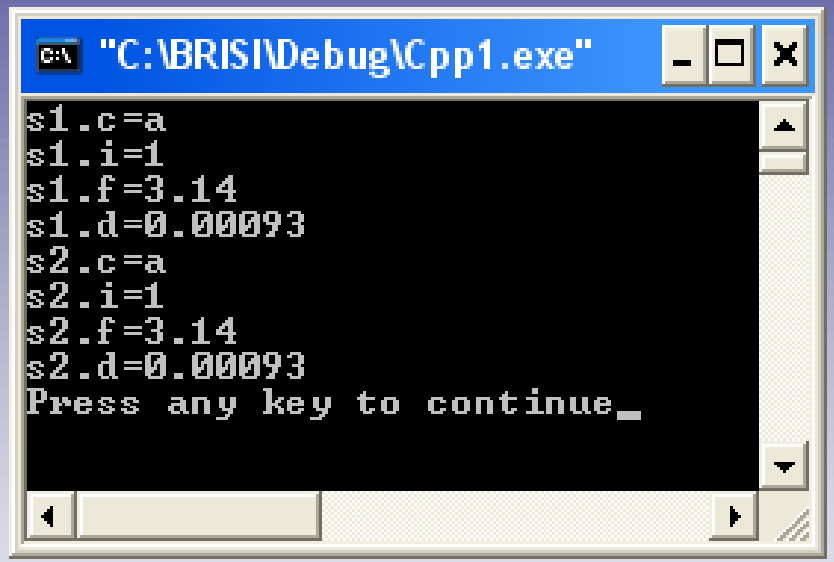
  expression –> identifier
- *Ekvivalentan izrazu sa operatorom* ***.***

  (*expression) . identifier

Strukture

- Struct je način da skupimo grupu različitih promenljivih i napravimo njihovu STRUKTURU. Jednom kada se ona kreira, možemo napraviti mnoge instance ovog novog tipa promenljivih koje smo uveli. Na primer:

```
#include<iostream.h>
struct Structure1 {
char c;
int i;
float f;
double d;
};
int main() {
struct Structure1 s1, s2;    // instance
s1.c = 'a'; cout<<"s1.c="<<s1.c<<endl;
s1.i = 1; cout<<"s1.i="<<s1.i<<endl;
s1.f = 3.14; cout<<"s1.f="<<s1.f<<endl;
s1.d = 0.00093; cout<<"s1.d="<<s1.d<<endl;
s2.c = 'a'; cout<<"s2.c="<<s2.c<<endl;
s2.i = 1; cout<<"s2.i="<<s2.i<<endl;
s2.f = 3.14; cout<<"s2.f="<<s2.f<<endl;
s2.d = 0.00093; cout<<"s2.d="<<s2.d<<endl;
return 0;}
```

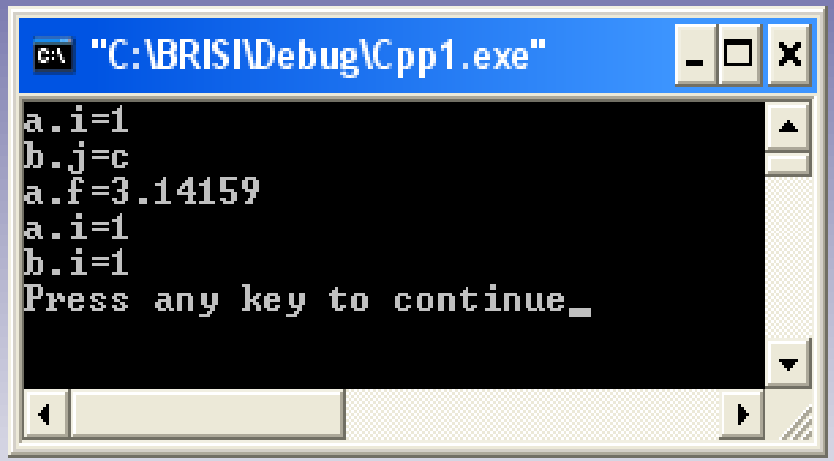


U main( ), svaka od instanci strukture ima svoje separatne verzije c, i, f i d. Tako s1 i s2 reprezentuju grupe potpuno nezavisnih promenljivih. Da bi selektovali neki od elemenata u okviru s1 ili s2, koristimo '.'

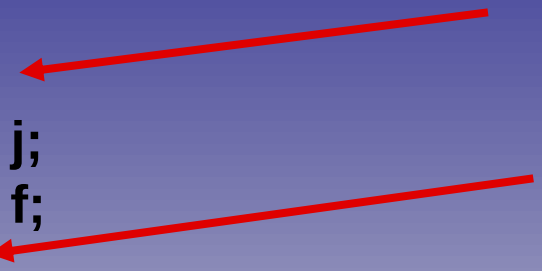
Strukture

```
#include <iostream.h>
struct A
        {int i;
        char j;
        float f;
        void func( );};
void A::func( ) { }
struct B
        {public:
        int i;
        char j;
        float f;
        void func();};
        void B::func() {  }
int main()
{A a; B b;  // instance struktura A  i  B
a.i = b.i = 1;cout<<"a.i="<<a.i<<endl;
a.j = b.j = 'c';cout<<"b.j="<<b.j<<endl;
a.f = b.f = 3.14159;cout<<"a.f="<<a.f<<endl;
a.func();cout<<"a.i="<<a.i<<endl;
b.func();cout<<"b.i="<<b.i<<endl;
return 0;}
```

Tri nove ključne reči za setovanje ograničenja strukture: **public, private i protected**



Setovanje ograničenja strukture



```
struct B
{ private:
        char j;
        float f;
public:
        int i;
        void func( ); };
void B::func( ) {i = 0; j = '0'; f = 0.0;};
int main()
{ B b;
b.i = 1; // OK, public
//! b.j = '1'; // Ilegal, private
//! b.f = 1.0; // Ilegal, private
return 0; }
```

Prijateljske funkcije strukture

- Ako želimo da eksplicitno damo pristup nekoj funkciji koja nije članica tekuće strukture, to činimo deklarisanjem prijateljske funkcije i to unutar same deklaracije strukture. Važno je da se deklaracija ove funkcije nalazi unutar deklaracije strukture pošto moramo omogućiti da kompajler pročita kompletnu deklaraciju i na osnovu toga utvrdi sva pravila oko veličine i ponašanja ove grupe podataka. A najvažnije pravilo u svakom odnosu relacija je definisanje ''ko može pristupiti mojim privatnim podacima?''

– *There's no magic way to "break in" from the outside if you aren't a **friend**;*

Prijateljske funkcije strukture

```
/*program demonstrira upotrebu
funkcija u radu sa strukturama
*/
#include <iostream>
using namespace std;

struct Racun{
  double stanje;
  double kamata;
  int period;
};
Racun udvostruciKamatu(Racun);
int main (){
  Racun moj_racun, novi_racun;
  cout<<"Unesite stanje racuna: ";
  cin>>moj_racun.stanje;
  cout<<"Unesite odobrenu kamatu racuna: ";
  cin>>moj_racun.kamata;
  cout<<"Unesite period orocenja racuna: ";
  cin>>moj_racun.period;
  novi_racun=udvostruciKamatu(moj_racun);
  cout<<"Podaci vezani za \"moj_racun\" su: \n";
  cout<<moj_racun.stanje<<endl;
  cout<<moj_racun.kamata<<endl;
  cout<<moj_racun.period<<endl;
  cout<<"Podaci vezani za \"novi_racun\" su: \n";
  cout<<novi_racun.stanje<<endl;
  cout<<novi_racun.kamata<<endl;
  cout<<novi_racun.period<<endl;
  system("Pause");
  return 0;
}
Racun udvostruciKamatu(Racun stari_racun){
    Racun temp=stari_racun;
    temp.kamata=2*stari_racun.kamata;
    return temp;
}
```

```
/*program demonstrira  upotrebu
struktura  unutar struktura
*/
struct Datum{
  int mjesec;
  int dan;
  int godina;
};
struct Osoba{
  double visina;
  int tezina;
  Datum rodjendan;
};
int main (){
  Osoba osoba1;
  cout<<"Unesite vasu visinu: ";
  cin>>osoba1.visina;
  cout<<"Unesite vasu tezinu: ";
  cin>>osoba1.tezina;
  cout<<"Unesite godinu rodjenja: ";
  cin>>osoba1.rodjendan.godina;
  cout<<"Unesite mjesec rodjenja: ";
  cin>>osoba1.rodjendan.mjesec;
  cout<<"Unesite datum u mjesecu: ";
  cin>>osoba1.rodjendan.dan;

  cout<<"Unijeli ste ove podatke: \n";
  cout<<"Visina: "<<osoba1.visina<<endl;
  cout<<"Tezina: "<<osoba1.tezina<<endl;
  cout<<"Datum rodjenja: "<<osoba1.rodjendan.dan;
  cout<<"."<<osoba1.rodjendan.mjesec;
  cout<<"."<<osoba1.rodjendan.godina<<"."<<endl;


  system("Pause");
  return 0;
}
```

Prijateljske funkcije strukture

```
// primjer sa strukturama
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
struct movies t
{
   string title;
   int year;
} mine, yours;
void printmovie (movies t movie);
int main ()
{
   string mystr;
   mine.title = "2001 A Space Odyssey";
   mine.year = 1968;
   cout << "Naziv: ";
   getline (cin,yours.title);
   cout << "Godina: ";
   getline (cin,mystr);
   stringstream(mystr) >> yours.year;
   cout << "Moj omiljeni film:\n ";
   printmovie (mine);
   cout << "Tvoj omiljeni film:\n ";
   printmovie (yours);
   return 0;
}
void printmovie (movies t movie)
{
   cout << movie.title;
   cout << " (" << movie.year << ")\n";
}
```

```
Naziv: Alien
Godina: 1979
Moj omiljeni film:
 2001 A Space Odyssey (1968)
Tvoj omiljeni film:
 Alien (1979)
```

Prijateljske funkcije strukture

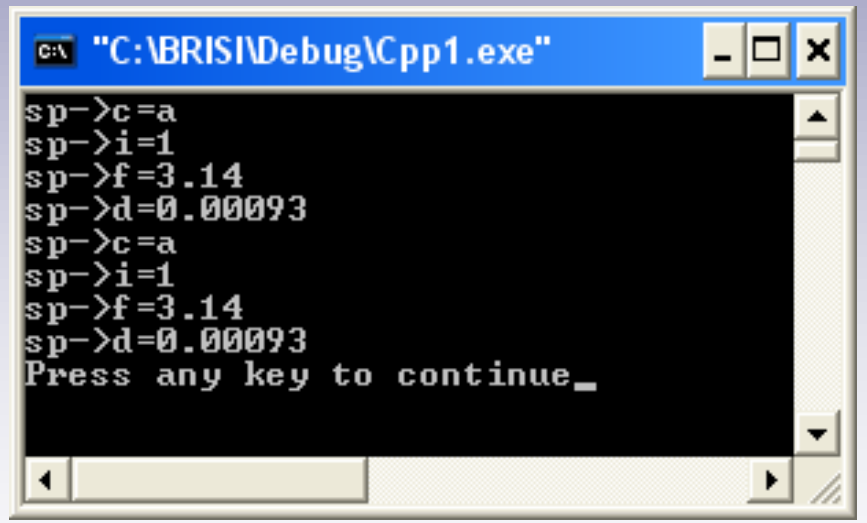
```
// niz struktura
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
#define N MOVIES 3
struct movies_t
{
   string title;
   int year;
} films [N_MOVIES];
void printmovie (movies t movie);
int main ()
{
   string mystr;
   int n;
   for (n=0; n<N MOVIES; n++)
   {
      cout << "Naziv: ";
      getline (cin,films[n].title);
      cout << "Godina: ";
      getline (cin,mystr);
      stringstream(mystr) >> films[n].year;
   }
   cout << "\nUpisali ste:\n";
   for (n=0; n<N MOVIES; n++)
   printmovie (films[n]);
   return 0;
}
void printmovie (movies_t movie)
{
   cout << movie.title;
   cout << " (" << movie.year << ")\n";
}
```

```
Naziv: Blade Runner
Godina: 1982
Naziv: Matrix
Godina: 1999
Naziv: Taxi Driver
Godina: 1976

Upisali ste:
Blade Runner (1982)
Matrix (1999)
Taxi Driver (1976)
```

Korišćenje pointera u strukturi

```
Using pointers to structs
#include<iostream.h>
typedef struct Structure3
{
char c;
int i;
float f;
double d;
} Structure3;

int main() {
Structure3 s1, s2;
Structure3* sp = &s1;
sp->c = 'a';cout<<"sp->c="<<sp->c<<endl;
sp->i = 1;cout<<"sp->i="<<sp->i<<endl;
sp->f = 3.14;cout<<"sp->f="<<sp->f<<endl;
sp->d = 0.00093;cout<<"sp->d="<<sp->d<<endl;
sp = &s2; // Pointer na drugi objekat strukture
sp->c = 'a';cout<<"sp->c="<<sp->c<<endl;
sp->i = 1;cout<<"sp->i="<<sp->i<<endl;
sp->f = 3.14;cout<<"sp->f="<<sp->f<<endl;
sp->d = 0.00093;cout<<"sp->d="<<sp->d<<endl;
return 0;}
```

U **main( )**, **struct** pointer **sp** inicijalno ukazuje na objekat **s1**, a članovi **s1** su inicijalizovani njihovom selekcijom preko '**->**' (jer koristimo isti operator u redu da pročitamo ove članove). Ali onda kada **sp** ukazuje na objekat **s2**, te promenljive se inicijalizuju na isti način..



Korišćenje pointera u strukturi

izraz	Objašnjenje	Ekvivalent
a.b	Pristup podatku članu b objekta a	
a->b	Pristup podatku članu b objekta na koji pokazuje a	(*a).b
*a.b	Pristup vrijednosti na koju pokazuje podatak član b objekta a	*(a.b)

Pokazivači na strukture

Kao i za bilo koji drugi tip podatka, postoje i pokazivači na strukture:

```
struct movies_t {
   string title;
   int year;
};
movies_t amovie;
movies_t * pmovie;
```

Ovdje je *amovie* objekat strukture *movies_t*, do je *pmovie* pokazivač na objekat strukture *movies_t*. Dakle, sljedeći kod će biti validan:

```
pmovie = &amovie;
```

Sada je vrijednost pokazivača *pmovie* jednaka adresi objekta *amovie*.

Slijedi primjer sa pokazivačima, koji služi da se uvede novi operator: operator strelice (->):

Slijedi primjer sa pokazivačima, koji služi da se uvede novi operator: operator strelice (->):

```
// pokazivaci na strukture
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
struct movies t {
   string title;
   int year;
};
int main ()
{
   string mystr;
   movies t amovie;
   movies_t * pmovie;
   pmovie = &amovie;
   cout << "Naziv: ";
   getline (cin, pmovie->title);
   cout << "Godina: ";
   getline (cin, mystr);
   (stringstream) mystr >> pmovie->year;
   cout << "\nUpisali ste:\n";
   cout << pmovie->title;
   cout << " (" << pmovie->year << ")\n";
   return 0;
}
```

```
Naziv: Invasion of the body
snatchers
Godina: 1978

Upisali ste:
Invasion of the body snatchers
(1978)
```

Prethodni kod uključuje uvodjenje operatora strelica (->). Ovo je operator dereference koji se koristi sa pokazivačima na objekte sa članovima. Operator služi da pristupi članovima objekta preko pokazivača na objekat. U primjeru smo koristili:

```
pmovie->title
```

što je ekvivalentno sa:

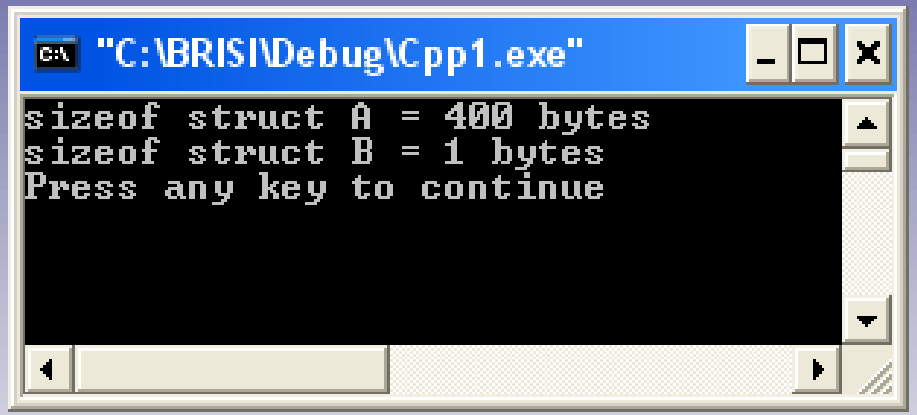
```
(*pmovie).title
```

Oba izraza *pmovie->title* i (**pmovie*).*title* su važeći i oba znače da smo pristupili članu *title* strukture podataka na koju pokazuje *pmovie*. Prethodno se mora jasno razlikovati od:

```
*pmovie.title
```

Strukture i operator ::

- #include <iostream.h>
- struct A
- { int i[100]; };
- struct B
- { void f(); };
- void B::f() { }
- int main( )
- {
- cout << "sizeof struct A = " << sizeof(A)<< " bytes" << endl;
- cout << "sizeof struct B = " << sizeof(B)<< " bytes" << endl;
- return 0;
- }



PRIMER 1

```
#include <iostream>
#include <cstring>

using namespace std;

struct Knjige
{
   char  Naslov[50];
   char  Autor[50];
   char  Izdavac[100];
   int   Knjiga_id;
};

int main( )
{
   struct Knjige Knjiga1;
   struct Knjige Knjiga2;

   // Specifikacija Knjiga1
   strcpy( Knjiga1.Naslov, "Learn C++ Programming");
   strcpy( Knjiga1.Autor, "Chand Miyan");
   strcpy( Knjiga1.Izdavac, "C++ Programming");
   Knjiga1.Knjiga_id = 6495407;

// Specifikacija Knjiga2
   strcpy( Knjiga2.Naslov, "Telecom Billing");
   strcpy( Knjiga2.Autor, "Yakit Singha");
   strcpy( Knjiga2. Izdavac, "Telecom");
   Knjiga2.Knjiga_id = 6495700;

   // Print Knjiga1 info
   cout << "Knjiga 1 Naslov : " << Knjiga1.Naslov <<endl;
   cout << "Knjiga 1 Autor : " << Knjiga1.Autor <<endl;
   cout << "BKnjiga 1 Izdavac : " << Knjiga1.Osoba <<endl;
   cout << "Knjiga 1 id : " << Knjiga1.Knjiga_id <<endl;

   // Print Knjiga2 info
   cout << "Knjiga 2 Naslov : " << Knjiga2.Naslov <<endl;
   cout << "Knjiga 2 Autor : " << Knjiga2.Autor <<endl;
   cout << "BKnjiga 2 Izdavac : " << Knjiga2.Osoba <<endl;
   cout << "Knjiga 2 id : " << Knjiga2.Knjiga_id <<endl;

   return 0;
}
```

Strukture i Funkcijski argumenti

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books book ); //funkcija
struct Knjige
{
   char  Naslov[50];
   char  Autor[50];
   char  Izdavac[100];
   int   Knjiga_id;
};

int main( )
{
   struct Knjige Knjiga1;
   struct Knjige Knjiga2;

   // Specifikacija Knjiga1
   strcpy( Knjiga1.Naslov, "Learn C++
Programming");
   strcpy( Knjiga1.Autor, "Chand Miyan");
   strcpy( Knjiga1.Izdavac, "C++ Programming");
   Knjiga1.Knjiga_id = 6495407;

// Specifikacija Knjiga2
   strcpy( Knjiga2.Naslov, "Telecom Billing");
   strcpy( Knjiga2.Autor, "Yakit Singha");
   strcpy( Knjiga2. Izdavac, "Telecom");
   Knjiga2.Knjiga_id = 6495700;

// Print Knjiga1 info
Stampaj(Knjiga1 );
// Print Knjiga2 info
Stampaj(Knjiga2 );
return 0; }

//FUNKCIJA
void Stampaj( struct Knjige knjiga ) { cout <<
“Naslov knjige : " << knjiga.Naslov <<endl; cout
<< “Autor : " << knjiga.Autor <<endl; cout <<
“Izdavač : " << knjiga.Izdavac <<endl; cout <<
“Knjiga id : " << knjiga.Knjiga_id <<endl; }
return 0;
}
```

Pointeri u Strukturi

```
#include <iostream>
#include <cstring>
using namespace std;
void printBook( struct Books *book );
struct Books { char title[50];
char author[50];
char subject[100];
int book_id; };
int main( ) {
struct Books Book1;
struct Books Book2;
Book 1 specification
strcpy( Book1.title, "Learn C++ Programming");
strcpy( Book1.author, "Chand Miyan");
strcpy( Book1.subject, "C++ Programming");
Book1.book_id = 6495407;
// Book 2 specification
strcpy( Book2.title, "Telecom Billing"); strcpy( Book2.author, "Yakit Singha"); strcpy( Book2.subject, "Telecom"); Book2.book_id = 6495700;
// Print Book1 info, passing address of structure
printBook( &Book1 );
// Print Book1 info, passing address of structure
printBook( &Book2 );
return 0; } // This function accept pointer to structure as parameter.
void printBook( struct Books *book )
{ cout << "Book title : " << book->title <<endl;
cout << "Book author : " << book->author <<endl;
cout << "Book subject : " << book->subject <<endl;
cout << "Book id : " << book->book_id <<endl; }
```

REZIME

Videli smo da strukture omogućavaju jednostavno modeliranje praktično svih tipova podataka koji se susreću u realnom životu. Međutim, one *definišu samo podatke* kojima se definiše neka pojava (neki student, fakultet...), ali *ne i postupke* koji se mogu primeniti nad tom pojavom. Na taj način, strukture su ‘’ preslobodne ‘’, u smislu da se nad njihovim poljima mogu nezavisno izvoditi sve operacije koje su dozvoljenje sa tipom podataka koje polje predstavlja, bez obzira da li ta operacija ima smisla nad strukturom kao celinom.

Pregled osnovnih koncepata OOP na jeziku C++

Klase, atributi i objekti

Konstruktori i destruktori

Nasleđivanje

Polimorfizam

Klase i objekti

- Klasa je ključni pojam C++ jezika koja namenski opisuje vrstu tipa podataka:

✓ prošireni tip strukture

- Klasa je grupa:

✓članova – logički povezanih podataka

✓članica –funkcija nad tim podacima

- Objekata - instanci klase za pristup:

✓članovima te klase

✓članicama te klase

Klase i objekti

- Klasa je skup apstraktnih podataka i funkcija međusobno povezanih, kako bi izvršili neki programski zadatak. Klasa je tako nalik strukturi.
- Koncept klasa u C++ ima obeležja koja su zajednička svim objektno orijentisanim jezicima. To su sledeća bitna obeležja:

  - atribut
  - metoda
  - kapsuliranje podataka i ograničenje pristupa samo atributima klase
  - konstruktor
  - destruktor

  - Klasa je opšti plan, obrazac za konkretni objekat (instancu klase). Svaki je pojavni oblik klase (instanca) je poseban objekat i on ima svoje elemente.
  - Objekti su međusobno nezavisni i svi su istog tipa (jer pripadaju istoj klasi), ali u memoriji zauzimaju posebne delove.

Klase i objekti

- *Klasa* je realizacija apstrakcije koja ima svoju internu predstavu (svoje atribute) i operacije koje se mogu vršiti nad njenim instancama-objektima. Klasa definiše apstraktni tip podataka.
- Jedan primerak takvog tipa (*instanca klase*) naziva se *objektom* te klase (engl. *class object).*

- *Podaci koji su deo klase nazivaju se podaci članovi klase (engl. data members).*

- *Funkcije koje su deo klase nazivaju se funkcije članice klase (engl. member functions).* Funkcije članice nazivaju se još i *metodima* klase.

Klase i objekti

- **Klasa** je nacrt objekta, odnosno način po kojem je određeni objekt stvoren. Npr.
- Određeni tip automobila definisan je jednim nacrtom (odnosno skupom nacrta).
  - Na temelju jednog nacrta moguće je proizvesti više primeraka istog tipa automobila (tj. objekata), koji će se međusobno razlikovati po stanju i ponašanju.
  - Podskup stanja i ponašanja (varijabli i metoda) može biti zajednički svim objektima određene klase. Nazivamo ih promenljivima i metodama klase.

Objekti mogu biti:

- **Fizički objekti: automobili u simulaciji saobraćaja, komponente u električnim kolima, zemlje u ekonomskim modelima, avioni u simulaciji vazdušnog saobraćaja, ...**

- **Elementi u kompjuterskom okruženju: prozori, meniji, grafički objekti (linije, pravougaonici, krugovi), miš, tastatura, printer, ...**

- **Konstruktori za čuvanje podataka: polje, stekovi, linkovane liste, binarna stabla, ...**

- **Ljudski entiteti: radnici, studenti, kupci, prodavci, ...**

- **Kolekcije podataka: inventar, lični podaci, rečnik, tabela geografske širine i dužine za gradove sveta, ...**

- **Tipovi podataka koje definiše korisnik: vreme, uglovi, kompleksni brojevi, tačke u ravni, ...**

- **Komponente u kompjuterskim igrama: automobili u trci, pozicije u raznim igrama (šah, mica, ...), životinje u eko-simulaciji, suparnici i prijatelji u avanturističkim igrama, ...**

Klase i objekti

- Klasa se pre upotrebe mora **deklarisati.** Posle deklaracije se ponaša kao **tip** podataka.
  - Sličan koncept vredi i za promenljive.
  - Pre upotrebe promenljive se deklarišu tako što se navede tip podatka i naziv promenljive. Npr. promenljiva A će čuvati celobrojni tip podatka: int A;.
- Tek posle deklaracije promenljiva se može inicijalizirati i upotrebiti u programu.
- Slično je i sa klasom. Tek posle deklaracije klase ona može poslužiti za kreiranje objekata.

Objekti klase nasleđuju sve atribute i metode klase kojoj pripadaju.

Klase i objekti

- Definicija klase predstavlja navođenje svih članova klase. Na osnovu te definicije mora da se zna veličina potrebanog memorijskog prostora za smeštanje pojednih objekata tipa te klase. Klasa se definiše opisom *class* čiji je opšti oblik:

class identifikator {

- ***public: član član -***
- ***private: član član - -***
- ***protected: član član - - };***

Klase i objekti

- *Član* u definiciji klase može da bude:

- ✓ Definicija atributa u obliku naredbe za definisanje podataka. Odjednom mogu da se definišu više atributa zajedničkog osnovnog tipa (neki od njih mogu da budu pokazivači, upućivači ili nizovi).
- ✓ Definicija metode koja se po formi poklapa sa definicijiom običnih (globalnih) funkcija: Za metode koje se definišu u definiciji klase, podrazumeva se modifikator *inline*. Oni se, dakle, ugrađuju neposredno u kod.
  - ✓ Metode koje se u definiciji klase samo deklarišu, moraju da budu definisane na nekom drugom mestu, izvan definicije klase. Te metode, takođe, mogu da se neposredno ugrađuju u kod, ali to treba uraditi primenom modifikatora *inline* -prilikom definisanja.

- ✓ Naredba *typedef* i *enum* kojom se uvodi identifikator tipa ili identifikatori simboličkih konstanti, ali koji ne stvaraju članove klase (ne utiču na veličinu objekata niti na funkcionalnost klase).

Klase i objekti – ograničenja

private

Članovi (podaci ili funkcije) klase iza ključne reči *private* : zaštićeni su od pristupa spolja (enkapsulirani su). Ovim članovima mogu pristupati samo funkcije članice klase. Ovi članovi nazivaju se *privatnim članovima klase* (engl. *private class members).*

public

Članovi iza ključne reči *public* : dostupni su spolja i nazivaju se *javnim članovima klase* (engl. *public class members).*

protected

Članovi iza ključne reči *protected* : dostupni su funkcijama članicama date klase, kao i klasa izvedenih iz te klase, ali ne i korisnicima spolja, i nazivaju se *zaštićenim članovima klase* (engl. *protected class members)*

Preporučuje se da se klase projektuju bez javnih podataka članova. Podaci članovi treba da budu privatni, osim ukoliko postoje jaki razlozi sa suprotnu odluku. Javne treba da budu samo funkcije članice koje predstavljaju operacije date apstrakcije koje su na raspolaganju korisnicima klase. Zaštićene su obično jednostavne operacije klase.

Klase i objekti

- **Klasa je skup podataka i funkcija međusobno povezanih kako bi izvršili neki programski zadatak.** Klasa je tako slična strukturi.
- Koncept klasa u C++ ima obeležja koja su zajednička svim objektno orijentisanim jezicima.

- To su sledeća bitna obeležja:
  - • atribut
  - • metoda
  - • kapsuliranje podataka i ograničenje pristupa samo atributima klase
  - • konstruktor
  - • destruktor

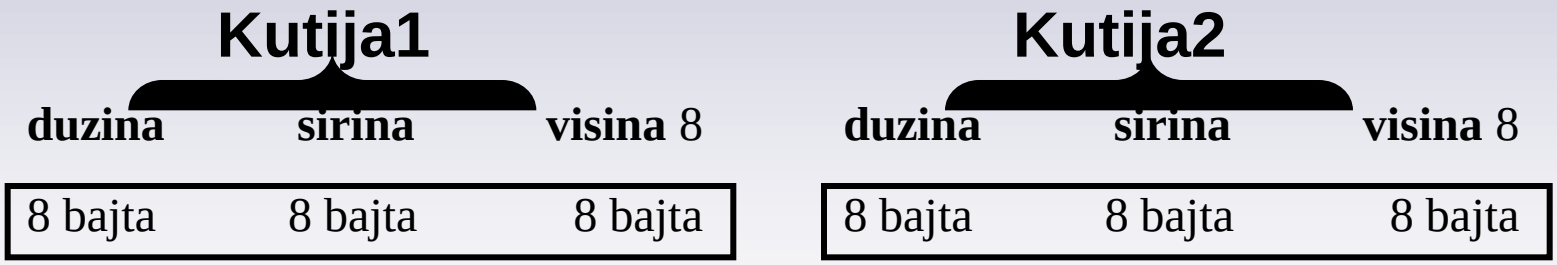
Klase

```
class Kutija
{ public:
double duzina; // duzina kutije u cm.
double sirina; // sirina kutije u cm.
double visina; // visina kutije u cm. };
```

U klasi Kutija definiše se pristup atributima članicama – instancama klase koji će slediti…

Deklarisanje Objekata Klase

- Deklarišemo objekte klase potpuno iste vrste i tipa, onako kako su deklarisani u osnovnoj inicijalizaciji klase.
  - **Kutija Kutija1; // Objekat Kutija1 tipa klase Kutija**
  - **Kutija Kutija2; // Objekat Kutija2 tipa klase Kutija**



Pristup podacima članicama klase

- Podacima članicama objekata klase može se pristupiti preko operatora: (.)

Operator Direct member access – direktni pristup članu (.)

```
Kutija2.visina = 18.0;
/* Dodeljivanje vrednosti članu visina,
objekta Kutija2, klase tipa Kutija*/
```

PRIMER

```
#include <iostream.h>
class Kutija // Globalna definicija opsega klase ( inicijalizacija )
{ public: double duzina;   // duzina Kutije
                double sirina;     // sirina Kutije
                double visina;    // visina Kutije };
int main(void)
{ Kutija Kutija1; // Deklarisanje objekta Kutija1 klase Kutija
  Kutija Kutija2; // Deklarisanje objekta Kutija1 klase Kutija
  double zapremina = 0.0; // postavljanje zapremine na pocetnu vrednost
  Kutija1.visina = 18.0;   // Definisanje vrednosti
  Kutija1.duzina = 78.0;  // clanova
  Kutija1.sirina = 24.0;   // objekta Kutija1
  Kutija2.visina = Kutija1.visina - 10;   // Definisanje clanova objekta Kutija2
  Kutija2.duzina = Kutija1.duzina/2.0;  // u odnosu na
  Kutija2.sirina = 0.25*Kutija1.duzina; // postavljenje u objektu Kutija1
        //Izracunavanje zapremine Kutija1
zapremina = Kutija1.visina * Kutija1.duzina * Kutija1.sirina;
cout << endl << "zapremina of Kutija1 = " << zapremina;
cout << endl << "Kutija2 ima konacnu zapreminu " << Kutija2.visina *
Kutija2.duzina * Kutija2.sirina << " kubnih cm.";
cout << endl // Prikaz velicine Kutija1 u memoriji
cout<< “Objekat Kutija1 zauzima"   << sizeof Kutija1 << " bytes."; cout <<endl;
return 0; }
```

Klase i objekti

- Analizirajmo jednu klasu i neka se naziva **Konto**.
  - Naziv klase: **Konto**.
  - Atributi, članovi klase:
    - **BrojKonta** (broj konta) i
    - **SaldoKonta** (saldo konta).
- Atributi u klasi Konto pripadaju celobrojnom tipu podataka - *integer*.
- Jedna metoda (funkcija) klase Konto može biti **uplata**, a označava povećanje stanja (salda) na kontu.

Klase i objekti

Naziv klase: Konto

Atributi:

int BrojKonta ;

int SaldoKonta;

Metoda:

Uplata;

Klase i objekti

Deklaracije strukture i klase.

```
struct Konto
{
int BrojKonta;
int SaldoKonta;
};
```

```
class Konto
{
int BrojKonta;
int SaldoKonta
};
```

Deklaracija klase ima sledeći izgled:

```
class naziv_klase    // zaglavlje klase.
{
        // telo klase.
}; Deklaracija klase uvek završava znakom ;
```

Klase i objekti

- **Posle deklaracije klase KONTO potrebno je kreirati objekte te klase.** Može se kreirati više objekata koji pripadaju klasi KONTO. Npr. tri takva objekta su

  – ŽiroRacun,
  – GotovinskiRačun i
  – RačunŠtednje.

- U primeru koji sledi, kreiraju se te tri objekta - instance klase KONTO:

  – **KONTO** ŽiroRacun;
  – **KONTO** GotovinskiRačun;
  – **KONTO** RačunŠtednje;

- Objekti se mogu kreirati tek nakon deklaracije klase KONTO i stoje na raspolaganju za dalju obradu.

Klase i objekti - skrivanje informacija

- Središnja paradigma objektne orijentacije temelji se na načelu skrivanja informacija. To znači, da se vrednosti atributa nekog objekta mogu promeniti samo upotrebom definisanih metoda.
- Kako se primenjuju načela skrivanja informacija u C++?

Ponovo je potreban raniji primer.

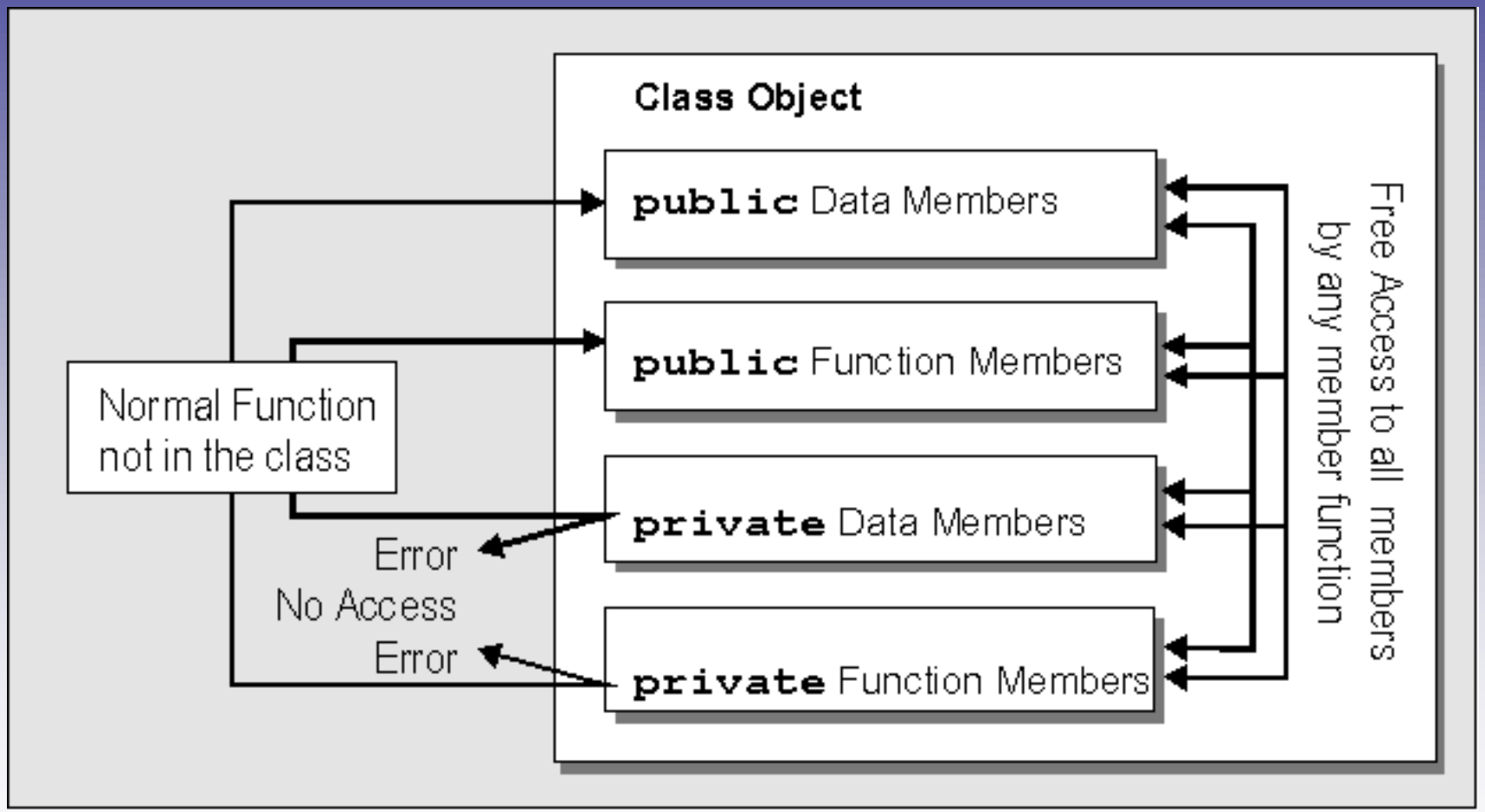
- U deklaraciji klase KONTO nalaze se atributi BrojKonta i SaldoKonta. Ako ništa nije naznačeno, atributi su deklarisani kao *privatni* upotrebom ključne reči **private**. To znači da spolja nije moguće pristupiti atributima i njihovim vrednostima unutar klase KONTO.

Tako se realizuje načelo skrivanja podataka.

Treba paziti na !

- class KONTO //Deklaracija klase Konto
- {
- int BrojKonta; // članovi koji su
- int SaldoKonta; // bez deklaracije po difoltu su privatni
- };
- void main()
- {
- KONTO ŽiroRačun; //Kreiranje objekta ŽiroRačun
- ŽiroRačun.BrojKonta=555666; **// Objekt instanciran, ali //greška u kompajliranju!!!**
- }

Klase i objekti



Ali postoje rešenja !

- Direktni pristup vrednostima atributa klase KONTO, što pokazuje gornji primer, nije moguć.

  Pojavljuje se greška pri kompajliranju:

  „KONTO::BrojKonta is not accessible“ !

- Kako je onda moguće pristupiti podacima jednog objekta, npr. podatku SaldoKonta?

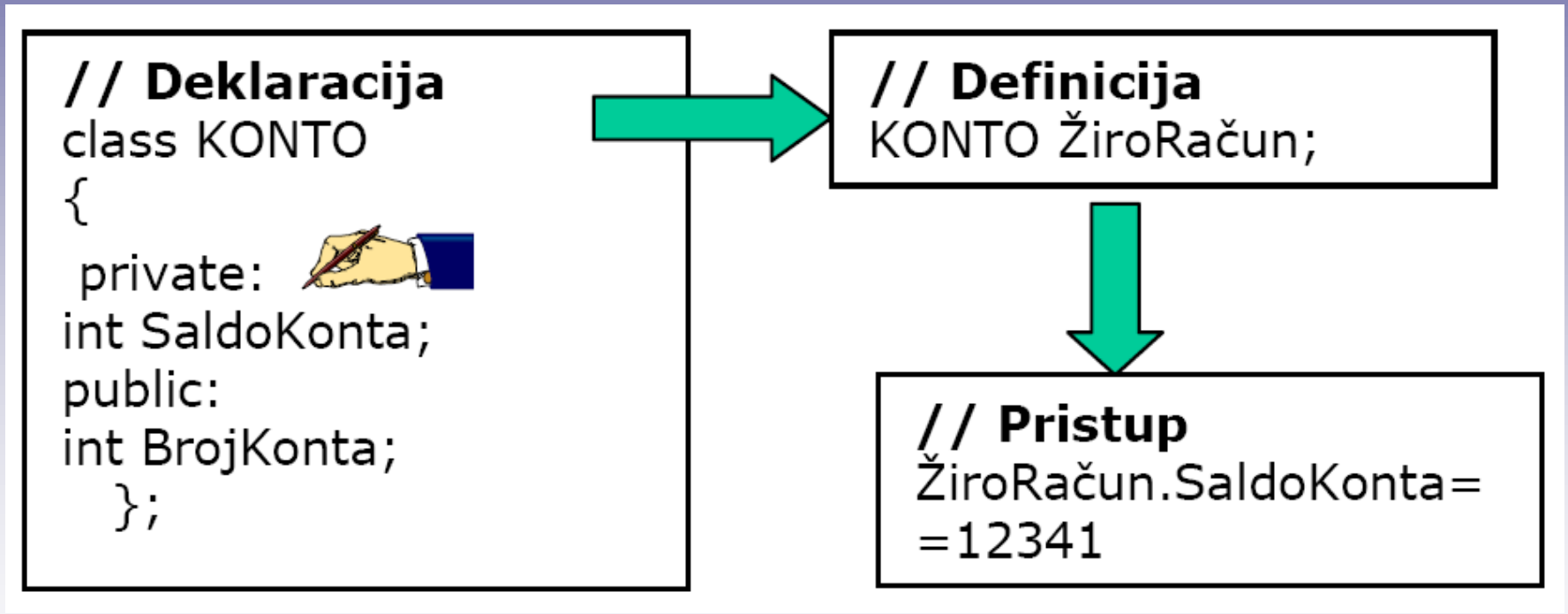
Postoje dve mogućnosti:

- Jedna je putem metode koja može menjati podatke. Ta mogućnost će se prikazati nešto kasnije.
- Druga je upotreba ključne reči **„public”- javni.** Atributi se deklarišu upotrebom ključne reči public što ilustruje sledeći primer.

ISPRAVNO!

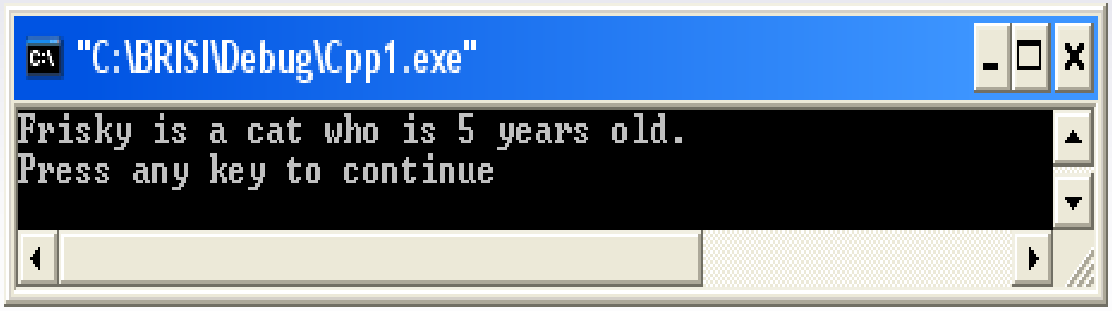
```
#include <iostream.h>
class KONTO //Definicija Klase Konto
{
public:
int BrojKonta;
int SaldoKonta // sada su ovi članovi javni
};
void main()
{
KONTO ŽiroRačun; // formiranje - instanciranje objekta ŽiroRačun
ŽiroRačun.BrojKonta=555666; //Pristup instanci i dodela vrednosti
ŽiroRačun.SaldoKonta=12341;
cout<<“Broj žiro računa je: “<<ŽiroRačun.BrojKonta;
cout<<“Stanje na računu je: “<<ŽiroRačun.SaldoKonta;
}
```

- Međutim, svojstvo **„public“** se mora primenjivati veoma promišljeno, inače će se izgubiti načelo skrivanja podataka. Zato je u deklaraciji atributa neke klase svojstvo **„private“** postavljano kao standardno.



Klase i objekti

```
#include <iostream.h>   // obezbedjenje cout
  class Cat       // deklarisanje klase
  {   public:      // clanovi klase koji slede su public
     int itsAge;
    int itsWeight; };
            int main()
 {    Cat Frisky;  // instanca klase Cat je objekat Frisky
Frisky.itsAge = 5; // dodela clana klase promenljivoj-objektu
    cout << "Frisky is a cat who is ";
    cout << Frisky.itsAge << " years old.\n";
    return 0; }
```



Klase i objekti

- Evo još jednog primera definicije klase:

Nedostaje private: Ali se podrazumeva

```
class Nesto
{
int a, b                // Privatni atributi.
int f (int) ;           // Privatna metoda.
Nesto n;                // GRESKA: ne može atribut  tipa Nesto!
Nesto *pn;              // Pokazivač na Nesto može.
Nesto &un;              // Upućivač na Nesto može.
public:
int c;                  // Javni atribut.
int g (int);            // Javna metoda.
Nesto h (Nesto);        // Metoda tipa Nesto može,
                        // takođe i argument tipa Nesto.
};
```

Klase i objekti

- *Objekti klasnih tipova :*

- Posle definisanja neke klase, automatski stoje na raspolaganju sledeće radnje nad tom klasom i njenim objektima:

✓ definisanje (stvaranje) objekata i nizova objekata ([ ]),
✓ definisanje pokazivača (*) i upućivača (&) na objekte,
✓ dodeljivanje vrednosti (=) jednog objekta drugom,
✓ nalaženje adresa objekata (&) i pristup objektima na osnovu adrese (*) ili indeksiranjem ([ ]),
✓ prisrup članovima objekata neposredno (.) ili posredno (->).

Metode Klase

- **Metode su funkcije** koje pripadaju jednoj klasi. Samo pomoću njih je moguće pristupiti atributima klase koji su deklarisani kao privatni **(private)**.
  - Metode zato osiguravaju i primjenjuju načelo tajnosti podataka.
  - Prostor njihove upotrebe je klasa.
  - One ne postoje izvan klase. Metode se mogu pozvati samo unutar klase ili ih može pozvati instanca klase.
  - Pomoću metoda klase se može pristupiti svim atributima klase bez obzira na njihova ograničenja pristupa.

Metode Klase

- Metoda u C++ mora biti pre upotrebe ***deklarisana***, a zatim i ***definisana***. Metode se kao i atributi mogu deklarisati uporabom ključnih riječi public, protected i private.

  - Metode deklarisane kao *public* omogućuju pristup svim delovima klase.
  - Metode *private* služe za interni rad s klasom. Korisnik ih ne može pozivati. Zato su privatne i skrivene od spoljnjeg sveta.
  - Metode *protected*. Njima mogu pristupiti samo klase koje nasleđuju metode svojih nadklasa.

Metode Klase

- Slično funkciji, metoda ima glavu i telo i deklariše se kao:

<povratni tip> ***naziv_metode***

(<tip>argument_1,

<tip>argument_2,...<tip>argument_n);

Deklaracija metode

```
class KONTO
{ private:
double Saldo; // Deklaracije jednog atributa
  public:
  int Saldo_pročitati(void); // Deklaracija metode public
};                                   //  u okviru klase
```

- U klasi KONTO deklarisana je metoda:

  int Saldo_pročitati(void). Povratni tip podataka metode **int Saldo_pročitati(void)** je integer, naziv metode je **Saldo_pročitati**, a metoda ima samo jedan argument tipa void.

Primer definicije metode unutar klase.

```
class KONTO
  {
private:
int BrojRačuna; //Deklaracija atributa
int SaldoKonta;
public:
int Saldo_pročitati(void) // Definicija metode Saldo_pročitati
{ return(SaldoKonta);// Telo metode Saldo_pročitati  }
  };
```

Posle deklaracije moramo definisati metodu. Definisati metodu znači detaljno prikazati šta ta metoda radi.

Primer definicije metode izvan klase.

- Ako su metode složenije i sadrže veći broj programskih redova, onda je definicija unutar klase suviše nepregledna. Zato je bolje metodu definisati izvan klase:

- U definiciji metode izvan klase upotrebljava se tzv. operator klase (class operator), a njegov simbol je „**::**“. Naziva se i operator razrešenja dosega ili područja vrednosti. Njime definišemo pripadnost date metode.

.....

.....

- **int KONTO::Saldo_pročitati(void) //Glava metode:**
- **{ *// Metoda* Saldo_pročitati *isporučuje vrednost atributa Saldo //klase KONTO***
- **return(SaldoKonta); // Telo metode:**
- **}**

.....

Sada je jasno da metoda Saldo_pročitati propada klasi KONTO

Primer definicije metode izvan klase.

- Implementiranje metode izvan klase zahteva:
  - tip podataka koji će metoda vratiti (u gornjem primeru taj tip je int)
  - naziv klase kojoj pripada metoda
  - operator “::”,
  - naziv metode
  - argumenti unutar zagrada
  - i na kraju telo metode u kome su naredbe. One definišu šta će metoda raditi.

Metode Klase

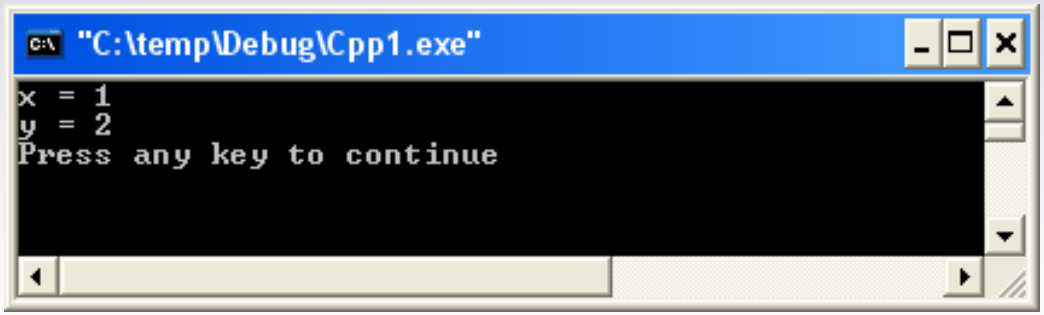
- Metode klase - funkcije članice klase:

- Ako se metoda definiše unutar definicije klase, nema nikakvog problema u korišćenju identifikatora ostalih članova te klase. Svi se oni nalaze unutar istog dosega. Prilikom **odvojenog definisanja metode**, definicija se nalazi izvan dosega kome pripada metoda.

- Zato je neophodno operatorom za razrešenje dosega (::) navesti ime klase kojoj pripada metoda. Drugi operand u posmatranom slučaju, treba da je identifikator metode koja se definiše. Time se doseg navedene klase proširuje na celu definiciju metode, pa unutar tela metode članovi klase mogu da se koriste navođenjem samo njihovih identifikatora.

Metode Klase

```
#include <iostream.h>
//using namespace std;
class MetodaStampe
{public:
 int x;
 int y;
 void print(){
 cout << "x = " << x << endl;
 cout << "y = " << y << endl;
                        }
                                  };

int main()
{ MetodaStampe IZVRSI;
  IZVRSI.x=1;
  IZVRSI.y=2;
  IZVRSI.print();
  return 0; }
```



Metode Klase

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

class Rectangle {
    int x, y;
  public:
    void set_values (int,int); // deklaracija metode za dodeljivanje vrednosti
//članovima klase
    int area (void) {return (x*y);} // metoda za izračunavanje površine unutar klase
                              };

void Rectangle::set_values (int a, int b) // ona se definiše izvan klase
{
  x = a;
  y = b;
            }

int main ()
{
  Rectangle rect; // Organizovanje instance rect klase tipa Rectangle
   rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
            }
```

Metode Klase

- **Primer:**

```
#include <iostream.h>
#include <string.h>
class CStr {
  private:
        char sData[256];
  public:
        char *get(void);
        int getlenght(void);
        void cpy(char *s);
        void cat(char *s);};

char *CStr::get(void) /*vrati pokazivač na  podatke znakovnog niza*/
{return sData;}
int CStr::getlenght(void) /*vrati dužinu niza*/
{return strlen(sData);}
void CStr::cpy(char *s) /*kopiraj sadržaj argumenta */
{strcpy(sData,s);}
void CStr::cat(char *s) /*nadoveži sadržaj argumenta */
{strcat(sData,s);}
```

Metode Klase

- Pozivanje funkcija članica – metoda odvija se po sintaksi:

objekat.funkcija_članica(argumenti)

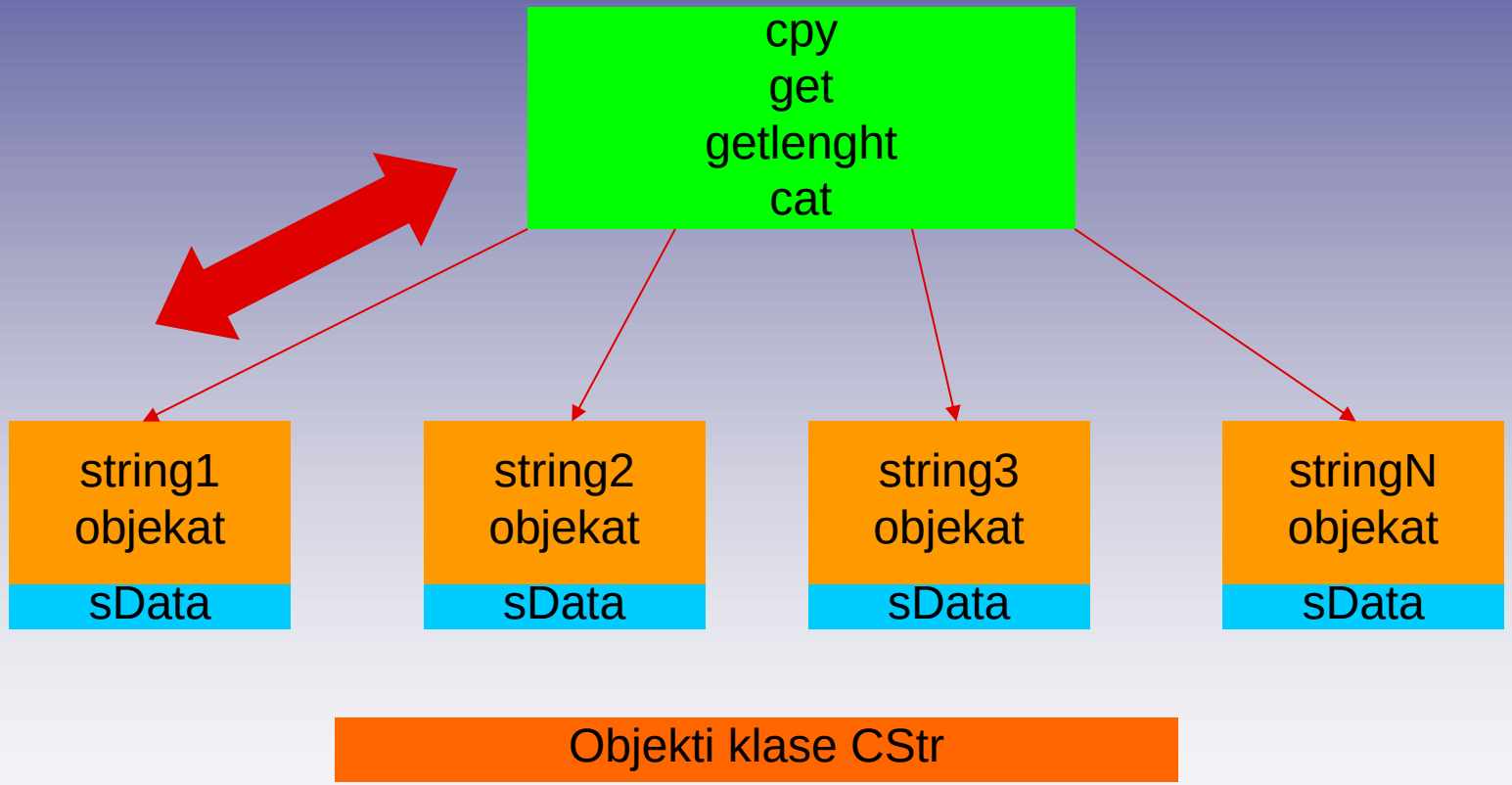
string1.cpy(“Ja se zovem”); /*kopira znakovni niz u objekat string1*/

Ovaj poziv funkcije nalaže objektu string1 da kopira u sebe podatke “Ja se zovem” pa je kopiranje deo ponašanja objekta.

U stvarnosti se dešava sledeće: prevodilac prepoznaje string1.cpy() kao poziv funkcije. On zna da je string1 objekat klase CStr.Zato se poziv funkcije razrešava kao poziv funkcije CStr::cpy

Metode Klase

- Funkcije članice klase CStr:



Metode Klase

- ***class*** *datum*
- *{*
- *int dan; // Varijable koje se koriste za čuvanje datuma*
- *int mesec;*
- *int godina;*
- ***public: //*** *Metode klase*
- *datum(); //konstructor*
- *datum(int,int,int);*
- *void postavidatum (int, int, int); //prototip funkcije*
- *void pokazi(); // prototip funkcije*

Klase, atributi i objekti

```
// kreiranje objekta Cat, postavljanje age, poziv funkcije
// meow, govori nam njenu starost - age, onda f.ja meow
//ponovo.
int main()
{
   Cat Frisky;
   Frisky.SetAge(5);
   Frisky.Meow();
   cout << "Frisky is a cat who is ";
   cout << Frisky.GetAge() << " years old.\n";
   Frisky.Meow();
    return 0;
}
```

```
Meow.
Frisky is a cat who is 5 years old.
Meow.
Press any key to continue
```

Metode Klase

```
#include <iostream>
using namespace std;
class KONTO
{
private:
double SaldoRačuna; //saldo računa
public:
//-------------------------------------------------------------
KONTO (double iznos) //konstruktor
{
SaldoRačuna = iznos;
}
//-------------------------------------------------------------
void depozit(double uplata) //uplata novca
{
SaldoRačuna = SaldoRačuna + uplata;
}
//-------------------------------------------------------------
void podizanje(double iznos) //podizanje novca
{
SaldoRačuna = SaldoRačuna - iznos;
}
//-------------------------------------------------------------
void prikaz() //prikaz salda računa
{
cout << “Saldo računa=” << SaldoRačuna << endl;
}
}; //kraj klase KONTO

int main()
{
KONTO prvi(67000,00); //kreiranje objekta prvi račun
cout << “Pre uplate “;
prvi.prikaz(); //prikaz stanja
prvi.depozit(1215,70); //uplata
prvi.podizanje(3245,00); //podizanje novca
cout << “Posle transakcije, “;
prvi.prikaz(); //prikaz salda računa
}
```

Summary of Operators

rator	Meaning	Grouping	Lvalue	Usage
`::`	scope resolution	L	Y/N	*class_name* `::` *member*
`::`	global name access		Y/N	`::` *name*
`[]`	indexing	L	Y	*expr* `[` *expr* `]`
`()`	function call	L	Y/N	*expr* `(` *list_of_expr* `)`
`()`	object construction		N	*type_name* `(` *list_of_expr* `)`
`.`	member access	L	Y/N	*expr* `.` *name*
`->`	indirect member access	L	Y/N	*expr* `->` *name*
`++`	postfix increment	L	N	*lvalue* `++`
`--`	postfix decrement	L	N	*lvalue* `--`
`++`	prefix increment	R	Y	`++` *lvalue*
`--`	prefix decrement	R	Y	`--` *lvalue*
`sizeof`	size of object	R	N	`sizeof` *expr*
`sizeof`	size of type	R	N	`sizeof(`*type*`)`
`new`	dynamic object creation		N	`new` *type*
`delete`	dynamic object destruction		N	`delete` *expr*
`~`	bitwise complement	R	N	`~` *expr*
`!`	logic negation	R	N	`!` *expr*
`-`	unary minus	R	N	`-` *expr*
`+`	unary plus	R	N	`+` *expr*
`&`	address of	R	N	`&` *lvalue*
`*`	pointer dereferencing	R	Y	`*` *expr*

Summary of Operators

`()`	cast	R	Y/N	`(`*type*`)` *expr*
`.*`	pointer to member dereferencing	L	Y/N	*expr* `.*` *expr*
`->*`	pointer to member dereferencing	L	Y/N	*expr* `->*` *expr*
`*`	multiplication	L	N	*expr* `*` *expr*
`/`	division	L	N	*expr* `/` *expr*
`%`	remainder	L	N	*expr* `%` *expr*
`+`	addition	L	N	*expr* `+` *expr*
`-`	subtraction	L	N	*expr* `-` *expr*
`<<`	shift left	L	N	*expr* `<<` *expr*
`>>`	shift right	L	N	*expr* `>>` *expr*
`<`	less than	L	N	*expr* `<` *expr*
`<=`	less than or equal to	L	N	*expr* `<=` *expr*
`>`	greater than	L	N	*expr* `>` *expr*
`>=`	greater than or equal to	L	N	*expr* `>=` *expr*
`==`	equal	L	N	*expr* `==` *expr*
`!=`	not equal	L	N	*expr* `!=` *expr*
`&`	bitwise And	L	N	*expr* `&` *expr*
`^`	bitwise Exclusive Or	L	N	*expr* `^` *expr*
`|`	bitwise Or	L	N	*expr* `|` *expr*
`&&`	logic And	L	N	*expr* `&&` *expr*
`||`	logic Or	L	N	*expr* `||` *expr*
`? :`	conditional operator	L	Y/N	*expr* `?` *expr* `:` *expr*

Summary of Operators

=	simple assignment	R	Y	*lvalue* = *expr*
*=	multiplication and assignment	R	Y	*lvalue* *= *expr*
/=	division and assignment	R	Y	*lvalue* /= *expr*
%=	remainder and assignment	R	Y	*lvalue* %= *expr*
+=	addition and assignment	R	Y	*lvalue* += *expr*
-=	subtraction and assignment	R	Y	*lvalue* -= *expr*
>>=	sift right and assignment	R	Y	*lvalue* >>= *expr*
<<=	sift left and assignment	R	Y	*lvalue* <<= *expr*
&=	bitwise And and assignment	R	Y	*lvalue* &= *expr*
|=	bitwise Or and assignment	R	Y	*lvalue* |= *expr*
^=	bitwise Xor and assignment	R	Y	*lvalue* ^= *expr*
,	sequence	L	Y/N	*expr* , *expr*

Klase, atributi i objekti

```
//primer klase CRectangle za izracunavanje povrsine pravougaonika

#include<iostream.h>
class CRectangle
{ int x,y;
public:
void set_values (int, int);
int area (void) {return (x*y);}};
void CRectangle::set_values (int a, int b)
{ x=a; y=b;}

void main()
{CRectangle recta, rectb;
recta.set_values (3,4);
rectb.set_values (5,6);
cout<< "recta area: "<<recta.area()<<endl;
cout<< "rectb area: "<<rectb.area()<<endl;}
```

Izlaz iz programa:

recta area: 12

rectb area: 30

Klase, atributi i objekti

```
//primer klase CVector uz koriscenje operatora    ::
#include<iostream.h>
class CVector {
public:
int x,y;
CVector( ) { };   //prazan blok instrukcija
CVector(int,int);  //drugi konstruktor
CVector operator + (CVector );
};
CVector :: CVector (int a, int b)
{ x=a; y=b;}
CVector CVector::operator+(CVector param)
{CVector temp;
temp.x=x+param.x;
temp.y=y+param.y;
return (temp);}

main(){
        CVector  a (3,1);
        CVector  b (1,2);
        CVector  c;
        c=a+b;
cout<< c.x<<","<<c.y<<endl;
return 0;}
```

Izlaz iz programa:

4,3

Pristup članicama klasa

- **Nekada ima smisla da dopustimo korisničkoj klasi, kada kreira neku novu klasu, da direktno pristupa članicama ostalih klasa koje moraju biti public.**

```
class Engine {
public:
void start() const {}
void rev() const {}
void stop() const {}
};
class Wheel {
public:
void inflate(int psi) const {}
};
class Window {
public:
void rollup() const {}
void rolldown() const {}
};
```

```
class Door {
public:
Window window;
void open() const {}
void close() const {}
};
class Car {
public:
Engine engine;
Wheel wheel[4];
Door left, right; // 2-door
};
int main() {
Car car;
car.left.window.rollup();
car.wheel[0].inflate(2);
}
```

Konstruktori i destruktori

- Konstruktori su posebne metode koje se ***pozivaju automatski kad god se kreira novi objekat.*** Konstruktor uvek ima tačno isti naziv kao i klasa.
  - Konstruktor je metoda koja može prihvatiti parametre, ali ne može vratiti ni jednu vrednost pa ni tip **void**.
  - Konstruktor nema povratne vrednosti.
  - Konstruktor kreira i inicijalizuje objekte klase.
- Destruktor briše objekte i oslobađa memoriju. Kao i konstruktor i destruktor uvek sadrži naziv klase, ali se ispred naziva klase nalazi tilda linija tj. znak ~. Destruktor nema argumente i ne vraća nikakvu vrednost.

Konstruktori i destruktori

- **Konstruktori** su metode koje imaju zadatak da učestvuju u kreiranju objekata ili promenljive, i mogu, ali ne moraju da sadrže njihovu inicijalizaciju.

- **Konstruktor** se najčešce koristi za inicijalizaciju članova - promenljivih jednog objekta.

- **Inicijalizacija** je operacija kojom se objekat isključivo dovodi u neko početno stanje.

- **U C++** ovaj postupak uglavnom radi konstruktor, koji zauzima prostor u memoriji a poziva se implicitno naredbama za definisanje podataka.

Konstruktori i destruktori

- **Destruktori** su specijalne metode koje ili direktno uništavaju ranije kreiran objekat ili učestvuju u toj operaciji.

- Sama operacija uništavanja promenljive ili objekta podrazumeva oslobađanje njihovog memorijskog prostora, a koriste se kad imamo dinamičko zauzimanje memorije.

- Terminator prevodi objekat u neko završno ili tačnije ‘’zamrznuto’’ stanje. Razliku između destruktora i terminatora je u tome što posle primene destruktora objekat postaje nedostupan (praktično mu se oduzima identitet), dok ga terminator ostavlja u nekom od stanja iz kojeg se opet može inicijalizovati bez promene identiteta.

Konstruktori i destruktori

- Svaka klasa ima bar dva konstruktora. To su **ugrađeni konstruktori**.
  - Jedan je konstruktor objekta, a
  - drugi konstruktor kopije.

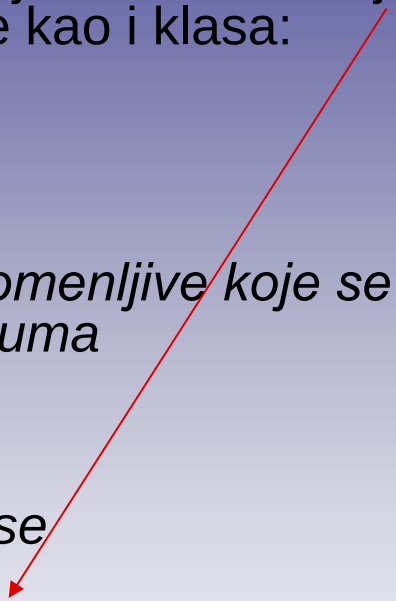
- **Konstruktor objekta** ima ulogu samo u zauzimanju memorije. Podrazumevani konstruktor je konstruktor bez parametara. Ako u klasi ima mnogo konstruktora prilikom implicitnog poziva poziva se podrazumevani konsturuktor. Zato klasu često tako pišemo da napišemo i podrazumevani konstruktor da bi u pomenutom slučaju on bio pozvan. Podrazumevane vrednosti parametara se ne vezuju isključivo za konstruktore već i za slobodne funkcije i metode.

- **Konstruktor kopije** služi za pravljenje kopije objekta. Pojavljuje se u kontekstu prenošenja argumenata po vrednosti. On je zadužen da ceo objekat kopira na stek, on postoji u svakoj klasi (ugrađen). Konstruktor treba da bude otvorena metoda.

Konstruktori i destruktori

- Da bi se omogućila inicijalizacija objekta, u klasi se definiše posebna metoda koja se implicitno (automatski) poziva i izvršava onda kada objekat nastaje. Ova funkcija se naziva *konstruktor* (*constructor*) i nosi isto ime kao i klasa:

- ***class*** *datum*
- *{*
- *int dan; // privatne promenljive koje se koriste za čuvanje datuma*
- *int mesec;*
- *int godina;*
- ***public:*** ***//*** *Metode klase*
- *datum(); //konstructor*
- *datum(int,int,int);*
- *void postavidatum (int, int, int); //prototip funkcije*
- *void pokazi(); //function prototype};*



Konstruktori i destruktori

- **Kao što smo rekli, kada definišemo klasu možemo definisati i njene metode. Definisanje metode ima sledeci format:**

- ***Povratni tip ime klase::ime metode (parametri) { Telo metode }***
- **Sledi definicija metoda za naš primer:**
- ***void datum::postavidatum (int dd, int mm, int yy)***
- ***{***
- ***dan=dd;***
- ***mesec=mm;***
- ***godina=yy;***
- ***}***
- ***void datum::pokazi (void)***
- ***{***
- ***cout<<"Dan:"<<dan<<"\n"<<"Mesec:"<<mesec<<"\n"<<"Godina:"<<godina;***
- ***}***

Konstruktori i destruktori

- Konstruktor se najčešce koristi za inicijalizaciju promenljivih jednog objekta. U pravilu se definiše više konstruktora.
- *datum::datum ()*
- *{*
- *dan=1;*
- *mesec=1;*
- *godina=2010;*
- *}*

U gornjem primeru konstruktor inicijalizuje datum na 1.1.2010. godine.

- ✓ Svaka klasa mora imati najmanje jedan konstruktor.
- ✓ Ime konstruktora mora biti jednako imenu klase.
- ✓ Iako nije prikazano u gornjem primeru, konstruktor može imati i parametre.
- ✓ Konstruktor ne vraća nikakvu vrednost (čak ni void).
- ✓ Objekat se može stvoriti samo jednom kao što se i promenljiva samo jednom može definisati.

Konstruktori i destruktori

- Ako dodamo još jedan konstruktor postojecoj klasi.

- *//Stari konstruktor koji postavlja datum na predefinisanu vrednost*
- *datum(); //ovo je samo prototip konstruktora*
- */*Novi konstruktor koji stvara objekat datum sa definisanim vrednostima za dan, mesec i godinu*/*
- *datum(int dd, int mm, int yy); //ovo je samo prototip konstruktora*
- *datum::datum(int dd, int mm, int yy)*
- *{*
- *dan=dd;*
- *mesec=mm;*
- *godina=yy; //ovo je definicija konstruktora*
- *}*
- Primer kako pozvati oba konstruktora:
- *datum petak; /*ovo će pozvati konstruktor sa predefinisanim vrednostima*/*
- *datum danas(15,11,2007); //ovo će pozvati novi konstruktor*

Konstruktori i destruktori

Primer:

```
class Osoba {
    public:
      Osoba(char *ime, int godine); // konstruktor
      void koSi();                  // funkcija: predstavi se!
    private:
      char *ime;                    // podatak: ime i prezime
      int   god;                    // podatak: koliko ima god
    };

    Osoba::Osoba (char *i, int g) {
      if (proveriIme(i))            // proveri ime
        ime=i;
      else
        ime="necu da ti kazem ko";
      god=((g>=0 && g<=100)?g:0);  // proveri godine
    }
```

Konstruktori i destruktori

- Kreiranje dva objekta sada izgleda ovako:

Osoba Bora(" Borivoje Milošević ",25),
mojOtac(“Milan Milošević",58);

Bora.koSi();
mojOtac.koSi();

- Moguće je definisati i funkciju koja se poziva uvek kada objekat prestaje da živi. Ova funkcija naziva se *destruktor* (*destructor*)

Destruktori

- Znači, kao što se objekat može stvoriti tako se mora naći način i za njegovo uništenje. C++ omogućuje eksplicitno korišćenje destruktora, tj. metode koja se poziva neposredno pre uništenja objekta.

- Sintaksa za destruktor je vrlo jednostavna:
- ~ime klase();
- U našem primeru:
- *~datum();*
- Destruktori se koriste za čišcenje memorije nakon što su objekti uništeni.

Konstruktori i destruktori

```
//primer klase CRectangle sa upotrebom konstruktora
#include<iostream.h>
class CRectangle {
int width,visina;
public:
CRectangle (int,int);  // konstruktor kao funkcija
int area (void) {return (width*visina);}// f-ja clanica klase
};
CRectangle::CRectangle (int a, int b)
{width=a; visina=b;}
main(){
CRectangle recta (3,4);// novi objekat, instanca klase
CRectangle rectb (5,6); );// novi objekat, instanca klase
cout<< "recta area: "<<recta.area()<<endl;// primena f-je nad objektom
cout<< "rectb area: "<<rectb.area()<<endl;// primena f-je nad objektom
}
```

Izlaz iz programa:

recta area: 12

rectb area: 30

Konstruktori i destruktori

ZADATAK

- a) Napraviti metodu koja će postavljati koordinate tačke na zadanu vrednost i koja će ispisati koordinate tačke.
- b) Dodati metodu koja će izračunati udaljenost između dve tačke i ispisati udaljenost na ekranu. Funkciju preopteretiti da računa udaljenost zadane tačke od središta koordinatnog sistema.

Konstruktori i destruktori

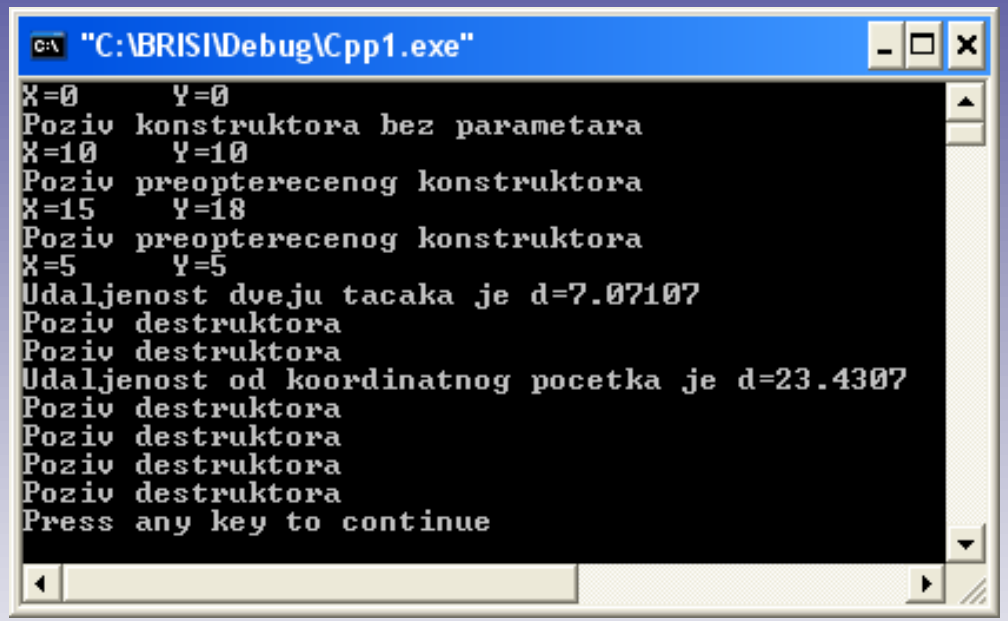
```
#include <iostream.h>
#include <math.h>
class Point {
int x;
int y;
double d;
public:
Point();  //konstruktor
Point(int, int); //konstruktor
~Point(); //destruktor
void SetXY(int,int); //Deklaracija metode SetXY
void Distance(Point,Point); //Deklaracija metode Distance
void Distance(Point);
};
Point::Point(){ //Definisanje konstruktora
SetXY(0,0);
cout<<"Poziv konstruktora bez parametara"<<endl;
}
Point::Point(int InitX,int InitY){ //Preopterećeni konstruktor
SetXY(InitX,InitY); //Poziv metode SetXY
cout<<"Poziv preopterećenog konstruktora "<<endl;
}
```

Konstruktori i destruktori

```
void Point::SetXY(int Xset,int Yset) { //Definisanje metode SetXY
x=Xset;
y=Yset;
cout<<"X="<<x<<"\tY="<<y<<endl;
}
void Point::Distance(Point P1, Point P2) { //Definisanje metode Distance
d=sqrt((P1.x-P2.x)* (P1.x-P2.x) + (P1.y-P2.y)* (P1.y-P2.y));
cout<<"Udaljenost dveju tačaka je d="<<d<<endl;
}
void Point::Distance(Point P1) {
d=sqrt((P1.x-0)* (P1.x-0) + (P1.y-0)* (P1.y-0));
cout<<"Udaljenost od koordinatnog početka je d="<<d<<endl;
}
Point::~Point() { //Definisanje destruktora
cout<<"Poziv destruktora"<<endl;
}
```

Konstruktori i destruktori

- GLAVNI PROGRAM:
- void main() {
- Point P1; /*Stvaranje objekta P1 prvim konstruktorom*/
- Point P2(10,10); /*Stvaranje P2 preopterecenim konstruktorom*/
- Point P3(15,18); /*Stvaranje P3 preopterecenim konstruktorom*/
- P1.SetXY(5,5);
- P1.Distance(P1,P2);
- P3.Distance(P3);
- }



Konstruktori i destruktori

```
//primer klase CRectangle sa upotrebom konstruktora i destruktora
#include<iostream.h>
class CRectangle
{
int *width,*visina;
public:
CRectangle (int,int);  //konstruktor kao funkcija clanica
~CRectangle ();  //destruktor kao funkcija clanica
int area (void) {return (*width**visina);}//f-ja clanica klase};
};
CRectangle::CRectangle(int a,int b)
{width=new int; visina=new int;*width=a;*visina=b;}
CRectangle::~CRectangle (){delete width; delete visina;}
main()
{
CRectangle recta (3,4); //novi objekat, instance klase
CRectangle rectb (5,6); //novi objekat, instance klase
cout<< "recta area: "<<recta.area()<<endl;//primena f-je nad objektom
cout<< "rectb area: "<<rectb.area()<<endl;//primena f-je nad objektom
return 0;
}
```

Izlaz iz programa:

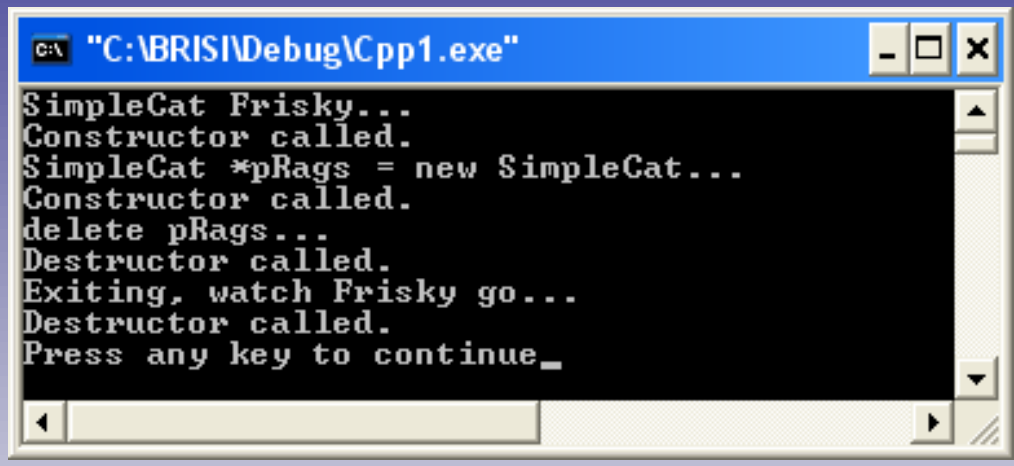
recta area: 12

rectb area: 30

Konstruktori i destruktori

- **Kreiranje objekta u slobodnom skladištu**

```
#include <iostream.h>
class SimpleCat
{    public:
      SimpleCat();
       ~SimpleCat();
private:
       int itsAge; };
  SimpleCat::SimpleCat()
  {   cout << "Constructor called.\n";
        itsAge = 1;       }
  SimpleCat::~SimpleCat()
  {    cout << "Destructor called.\n";    }

 int main()
 {    cout << "SimpleCat Frisky...\n";
          SimpleCat Frisky;
cout << "SimpleCat *pRags = new SimpleCat...\n";
          SimpleCat * pRags = new SimpleCat;
          cout << "delete pRags...\n";
          delete pRags;
          cout << "Exiting, watch Frisky go...\n";
          return 0;
  }
```



Konstruktori i destruktori

```
//primer klase CRectangle sa upotrebom ukazatelja
#include<iostream.h>
class CRectangle {
int width,visina;
public:
void set_values (int, int);
int area (void) {return (width*visina);}
};
void CRectangle::set_values (int a, int b)
{ width=a; visina=b;}

main()
{
CRectangle a, *b, *c;
CRectangle *d=new CRectangle[2];
b=new CRectangle;
c=&a;
a.set_values(1,2);
b->set_values(3,4);
d->set_values(5,6);
d[1].set_values(7,8);
cout<< "a area:     "<<a.area()<<endl;//primena f-je nad objektom
cout<< "*b area:    "<<b->area()<<endl;//primena f-je nad objektom
cout<< "*c area:    "<<c->area()<<endl;//primena f-je nad objektom
cout<< "*d[0] area: "<<d[0].area()<<endl;//primena f-je nad objektom
cout<< "*d[1] area: "<<d[1].area()<<endl;//primena f-je nad objektom
return 0;
}
```

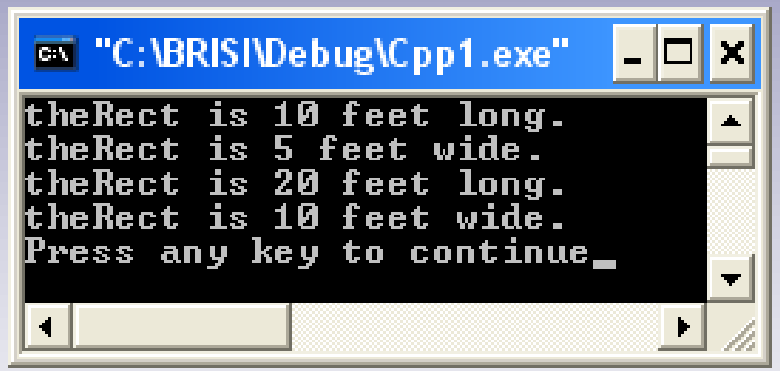
Izlaz iz programa:

```
a area:      2
*b area:    12
*c area:     2
d[0] area:  30
d[1] area:  56
```

Konstruktori i destruktori

- **Korišćenje this pointera**

```
#include <iostream.h>
class Rectangle
{    public:
     Rectangle();
     ~Rectangle();
     void Setduzina(int duzina) { this->itsduzina = duzina; }
     int Getduzina() const { return this->itsduzina; }
     void SetWidth(int width) { itsWidth = width; }
     int GetWidth() const { return itsWidth; }
private:
     int itsduzina;
     int itsWidth; };
Rectangle::Rectangle()
{   itsWidth = 5;
    itsduzina = 10; }
Rectangle::~Rectangle()
{}
int main()
{    Rectangle theRect;
     cout << "theRect is " << theRect.Getduzina() << " feet long.\n";
     cout << "theRect is " << theRect.GetWidth() << " feet wide.\n";
     theRect.Setduzina(20);
     theRect.SetWidth(10);
     cout << "theRect is " << theRect.Getduzina()<< " feet long.\n";
     cout << "theRect is " << theRect.GetWidth()<< " feet wide.\n";
     return 0; }
```



Nasleđivanje

- **Nasleđvanje je mehanizam pomoću koga je na temelju postojećih klasa moguće definisati nove i proširene klase. Ideja nasleđvanja je da se odrede slične klase i da se pri nasleđvanju promene samo neka specifična svojstva, dok se ostala svojstva nasleđuju u nepromenjenom obliku.**

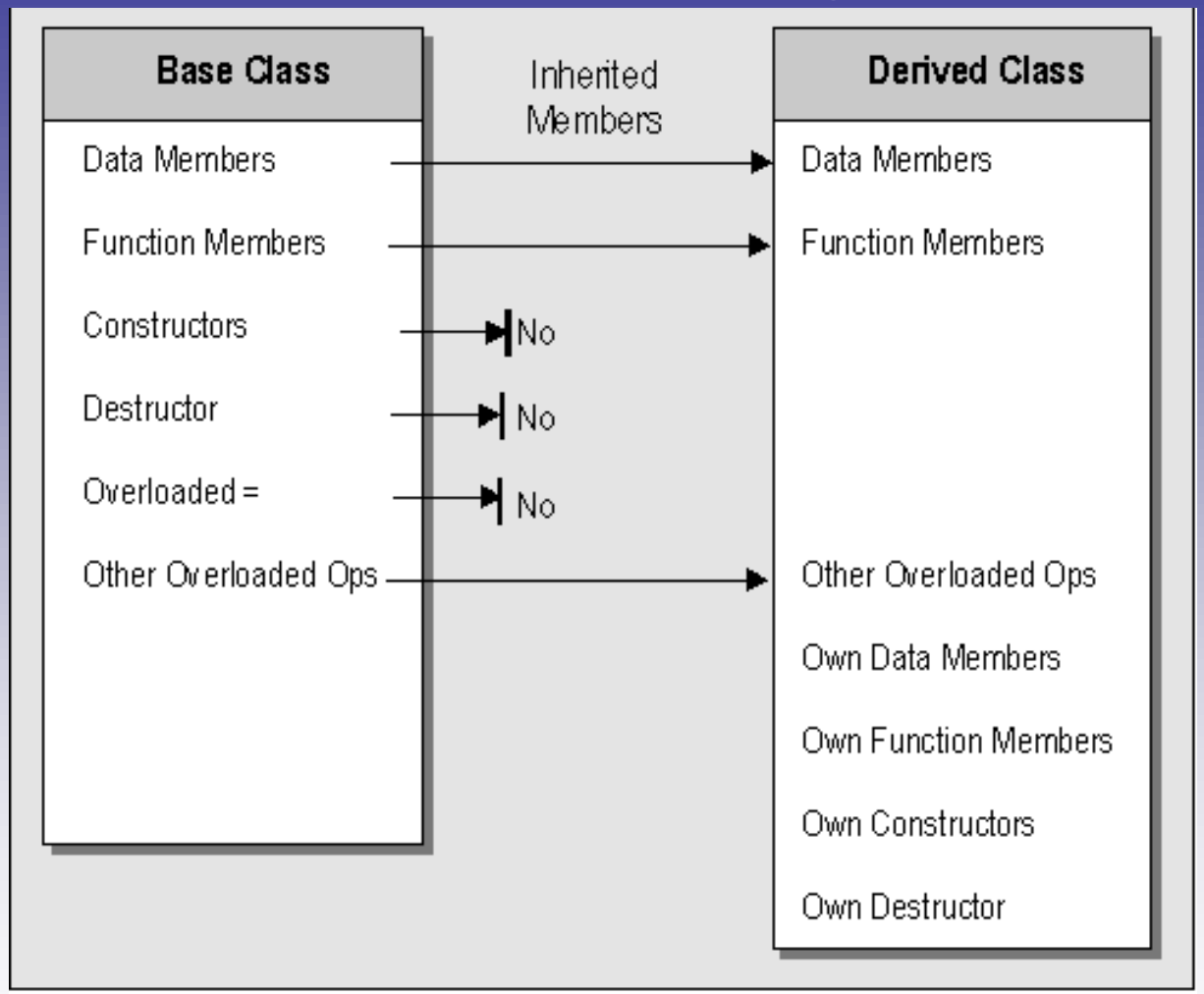
- **Svojstva izvedene klase:**

Ona nasleđuje promenljive i metode svoje nadređene klase tj. klase koju nasleđuje

- ✓ **Ima svoje nove promenljive i metode koje ispunjavaju njene specificne zahteve**
- ✓ **Izvedena klasa nasleđuje sva svojstva i mogućnosti nadređene klase**
- ✓ **Svaki konstruktor izvedene klase trebao bi pozvati konstruktor nadređene klase**
- ✓ **Npr. klasa Tačka može biti nadređena klasi Krug (krug se može opisati pomoću tačke i pripadajućeg radijusa)**

Ideja nasleđivanja je da svaki objekat podređene klase sadrži promenljive i metode nadređene klase. Ako se promenljive u nadređenoj klasi nalaze pod privatnim pristupom (private) onda im se ne može pristupiti iz podređene klase. Da bi im se moglo pristupiti promenljive bi se morale nalaziti pod ključnom reči protected, što bi značilo da im metode iz podređene klase mogu pristupiti. Ali ako želimo zadržati fleksibilnot i integritet prezentacije podataka u osnovnoj klasi moramo ih držati kao protected. Stoga ako je moguće, podređene klase bi trebale koristiti metode sa javnim pristupom (public) u nadređenoj klasi kako bi pristupile zaštićenim podacima.

Nasleđivanje



klasa naslednik:public osnovna klasa

Nasleđivanje

- **Primer:**

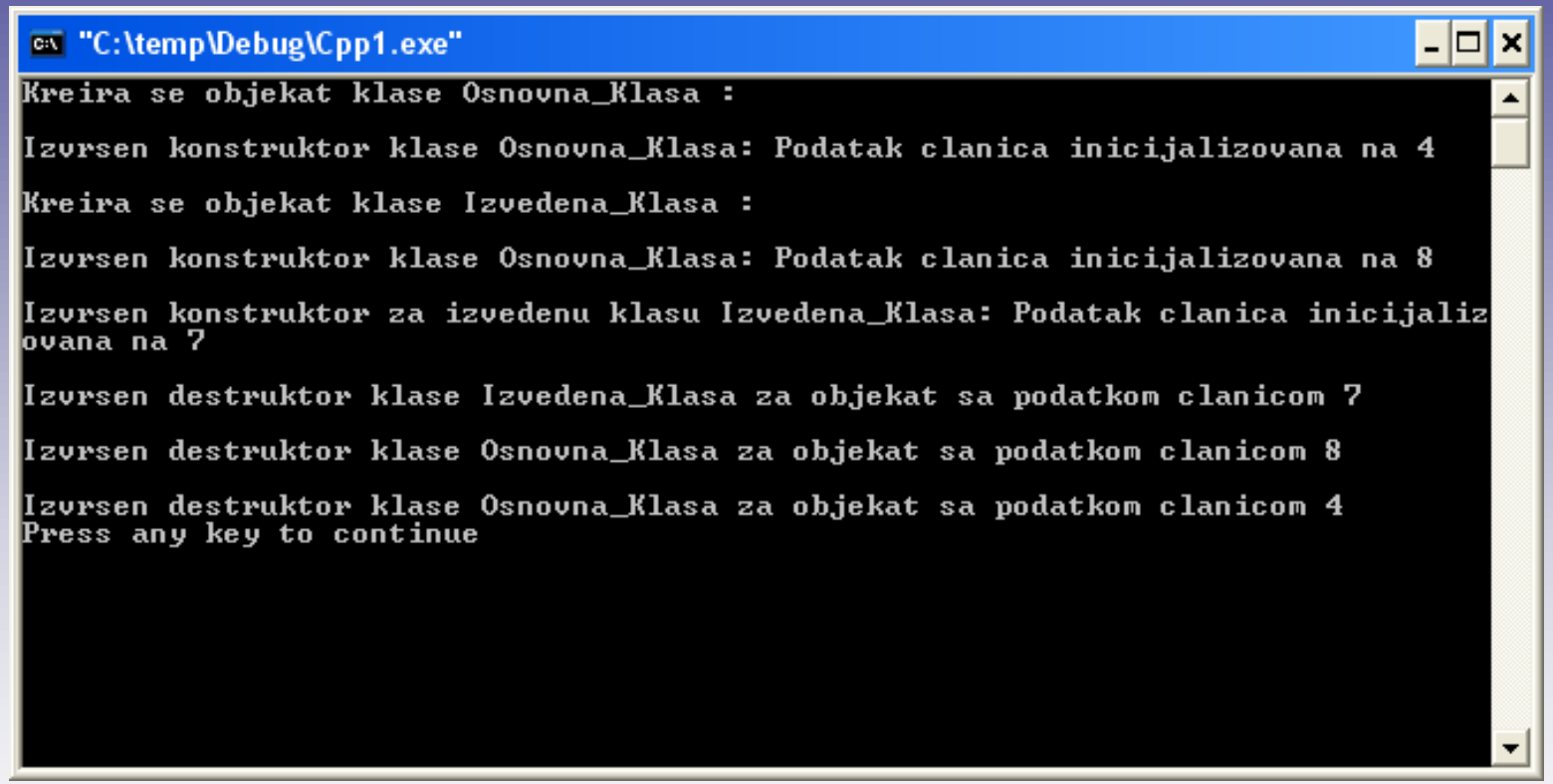
```
#include <iostream.h>
class A {
public:
void print() {
cout<<”Metoda koja pripada nadređenoj klasi”<<endl;}
};
class B: public A {
public:
void print() {
cout <<”Metoda koja pripada podređenoj klasi ”<<endl;}
};
void main() {
A Obj1;
B Obj2;
Obj1.print(); //Ovo poziva metodu iz nadređene klase
Obj2.print(); //Ovo poziva metodu iz podređene klase
Obj2.A::print(); //Ovo poziva metodu iz nadređene klase pomocu operatora ::
}
```

Nasleđivanje

```
// Program pokazuje kako se izvrsavaju konstruktori i destruktori u osnovnoj klasi

#include <iostream.h>
class Osnovna_Klasa
{
protected:
int n;
public:
Osnovna_Klasa(int i = 0); // Konstruktor sa jednim argumentom
~Osnovna_Klasa();
};
Osnovna_Klasa::Osnovna_Klasa(int i) : n(i)// ako pravimo polje intedžera svi se inicijalizuju na 0
{
cout << endl;
cout << "Izvrsen konstruktor klase Osnovna_Klasa: "<< "Podatak clanica inicijalizovana na " << n << endl;
}
Osnovna_Klasa::~Osnovna_Klasa()
{
cout << endl;
cout << "Izvrsen destruktor klase Osnovna_Klasa "<< "za objekat sa podatkom clanicom " << n << endl;
}
class Izvedena_Klasa : public Osnovna_Klasa
{
protected:
int m;
public:
Izvedena_Klasa(int j = 0);
~Izvedena_Klasa();
};
```

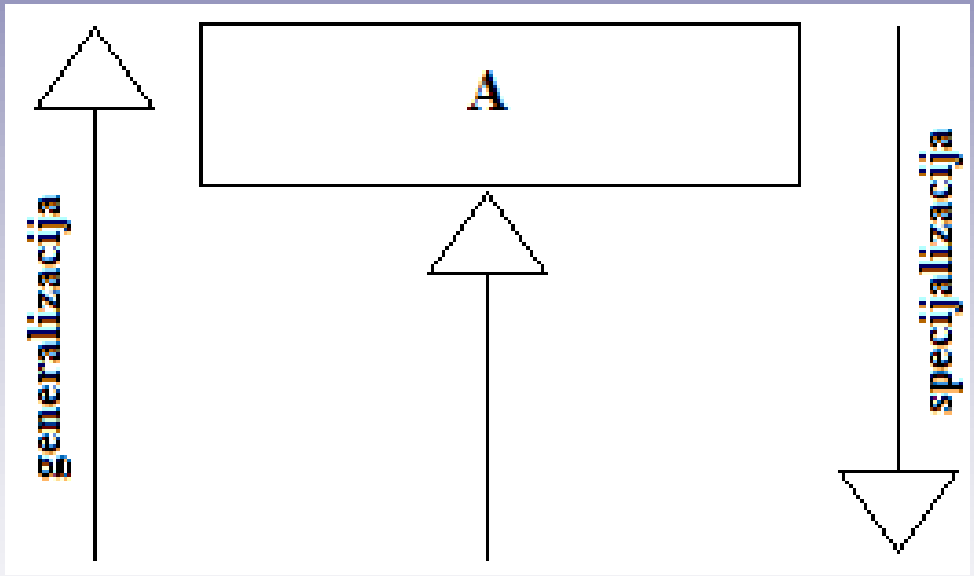
```
Izvedena_Klasa::Izvedena_Klasa(int j) : Osnovna_Klasa(j+1), m(j)
{
cout << endl;
cout << "Izvrsen konstruktor za izvedenu klasu Izvedena_Klasa: "
<< "Podatak clanica inicijalizovana na "
<< m <<endl;}
Izvedena_Klasa::~Izvedena_Klasa()
{
cout << endl;
cout << "Izvrsen destruktor klase Izvedena_Klasa "
<< "za objekat sa podatkom clanicom "
<< m << endl;
}
int main()
{
cout << "Kreira se objekat klase Osnovna_Klasa : " << endl;
Osnovna_Klasa obekt_osnovne_klase(4);
cout << endl;
cout << "Kreira se objekat klase Izvedena_Klasa : " << endl;
Izvedena_Klasa obekt_izvedene_klase(7);
return 0;
}
```

izlaz iz programa:



Nasleđivanje

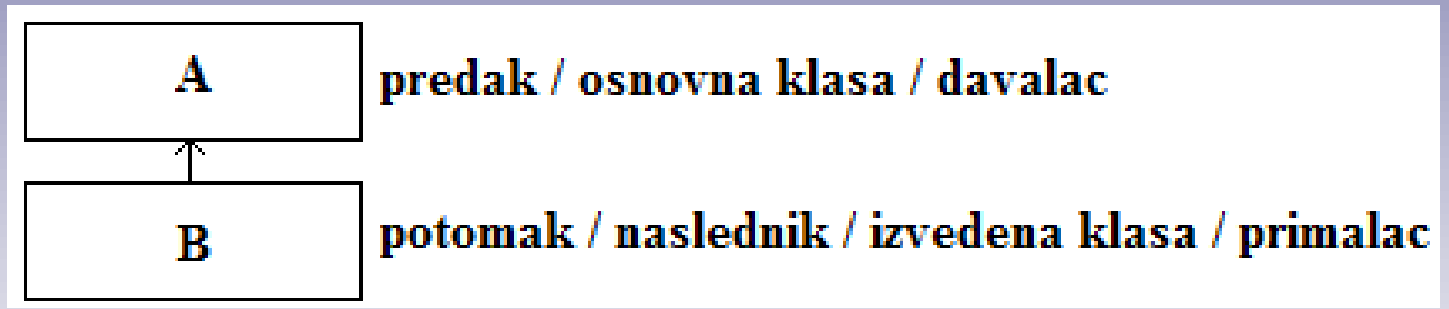
- Nasleđivanje modeluje odnos između dve klase. Sa jedne strane predstavlja vezu generalizacija-specijalizacija ili opšte-pojedinačno (npr. veza budilnik-sat)



Sa druge strane nasleđivanje je bitno kao sredstvo kojim se obezbeđuje višekratna upotreba.

Nasleđivanje

- **Definicija**: Nasleđivanje predstavlja preuzimanje kompletnog sadržaja neke druge klase uz mogućnost dodavanja članova svih vrsta i modifikacije metoda.



Osobine nasleđivanja:

- Izvođenje klase B iz klase A ne zahteva pristup njenom izvornom kodu.
- Klasa B preuzima sve što sadrži klasa A, sa tim da može još sadržaja da se doda.

U klasi B se mogu dodati nova polja ili metode.

- **Mogućnost modifikacije metoda** – isti posao u klasi A i klasi B se ne izvršava istom metodom. Modifikacija metode se zove redefinisanje, jer nema drugog načina da metodu promenimo nego da je ponovo napišemo pod istim imenom.
- **Tranzitivnost** – ako klasa C nasleđuje klasu B, a ona nasleđuje klasu A, onda klasa C nasleđuje klasu A.
- **Višestruko nasleđivanje** – nasleđivanje kod koga klasa nasleđuje dve ili više klasa.

Nasleđivanje

- Pretpostavimo da nam je potreban novi tip, **Maloletnik**. Maloletnik je "jedna vrsta" osobe, koja "poseduje sve što i osoba, samo ima još nešto", tj. ima staratelja. Znači, ovakva relacija između klasa naziva se *nasleđivanje*:

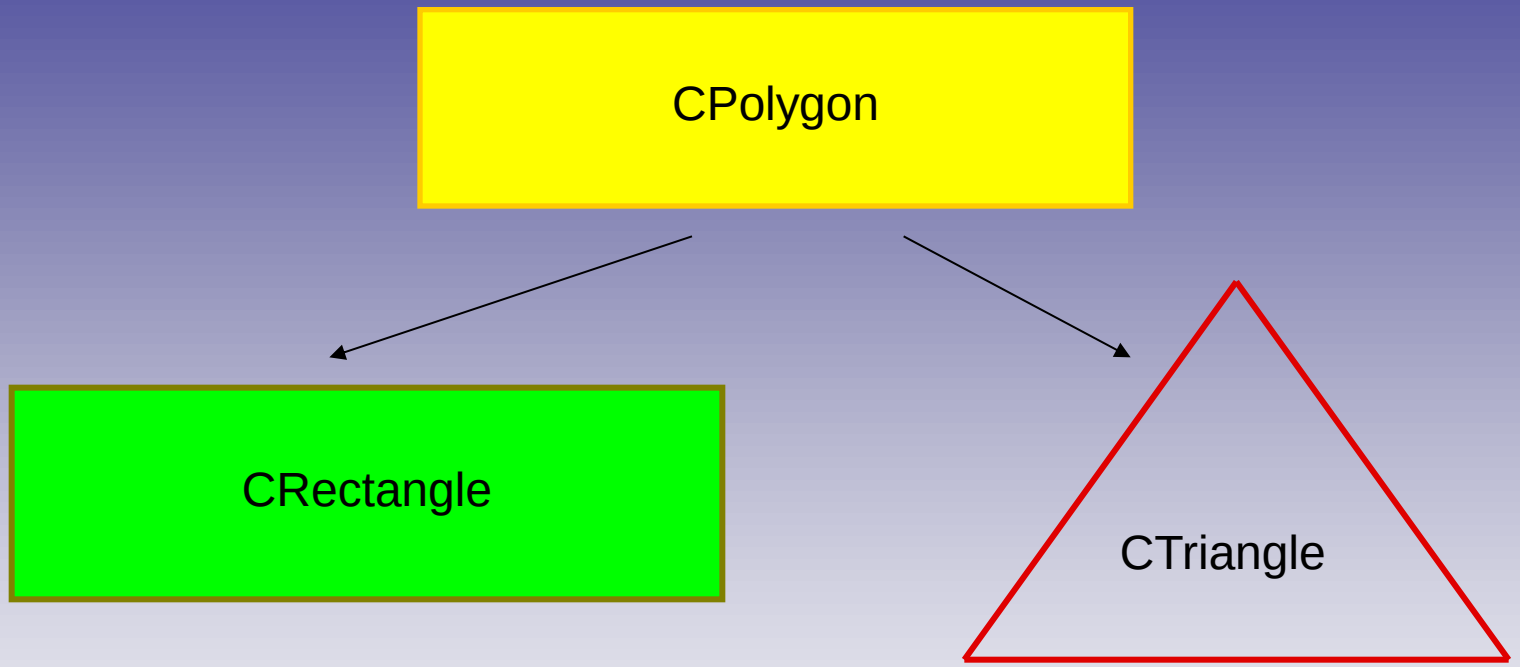
```
class Maloletnik : public Osoba {
public:
  Maloletnik (char* ime, char* staratelj, int godine);
  void koJeOdgovoran();
private:
  char *staratelj;
};

void Maloletnik::koJeOdgovoran (){
  cout<<"Za mene odgovara "<<staratelj<<".\n";
}

Maloletnik::Maloletnik (char *i, char *s, int g) :
Osoba(i,g), staratelj(s) {}
```

Nasleđivanje

- Sada se mogu koristiti i nasleđene osobine objekata klase **Maloletnik**, a na raspolaganju su i njihova posebna svojstva kojih nije bilo u klasi **Osoba**:

```
Osoba otac("Petar Petrovic",40);
Maloletnik dete("Milan Petrovic","Petar
Petrovic",12);

otac.koSi();
dete.koSi();
dete.koJeOdgovoran();
otac.koJeOdgovoran();     // ovo, naravno, ne može!
```

Nasleđivanje



Klasa CPoligon sadrži članove koji su opšti za sve mnogouglove pa u našem slučaju članovi width i visina za klasu CRectangle i Ctriangle biće proizvodi te klase.

Nasleđivanje

```
//primer klase CPolygon i naslednih klasa CRectangle I CTriangle
#include<iostream.h>
class CPolygon {
protected:
int width,visina;
public:
void set_values (int a,int b)
{ width=a;visina=b; }
};
class CRectangle: public CPolygon { //klasa naslednik:public osnovna klasa
public:
int area (void)
{ return (width * visina); }};
class CTriangle: public CPolygon { //klasa naslednik:public osnovna klasa
public:
int area (void)
{ return (width * visina/2); }};
main() {
      CRectangle rect;//instanca, novi objekat
      CTriangle trgl; //instanca, novi objekat
      rect.set_values(4,5);
      trgl.set_values(4,5);
      cout<< "rect area: "<<rect.area()<<endl;
      cout<< "trgl area: "<<trgl.area()<<endl;
return 0;
}
```

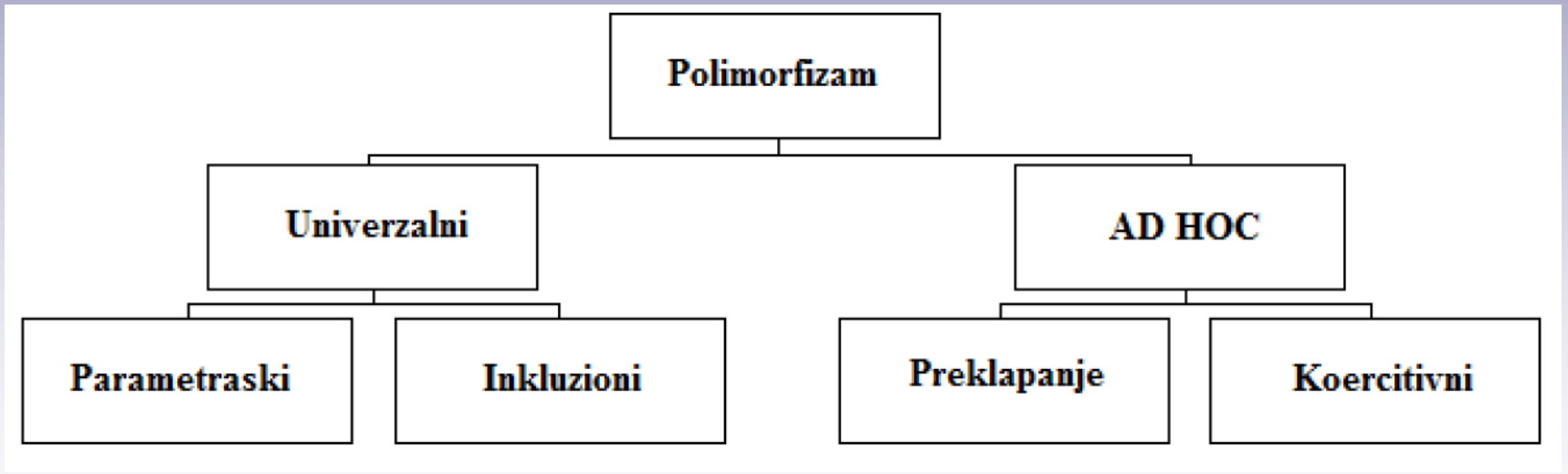
Izlaz iz programa:

rect area: 20

trgl area: 10

Polimorfizam

- Polimorfizam je konteksno, zavisno ponašanje (određene kategorije se u zavisnosti od okolonosti ponašaju drugačije). Tako se ponašaju promenljive, objekti i funkcije.



Polimorfizam

- **Univerzalni polimofizam**, jedna kategorija se ponaša različito u zavisnosti od okolnosti.
- Imamo dve podvrste:
  - ***parametarski*** polimorfizam funkcije je osobina funkcije da podešava svoje ponašanje prema tipu podataka. Klase i tipovi koji iskazuju osobine ovog polimorfizma su generičke klase i generički tipovi.
  - ***inkluzioni*** polimorfizam je osobina promenljive da može istovremeno pripadatai većem broju tipova (npr. *char* koji je i *int*)

Polimorfizam

- **AD HOC polimofizam**, prividni polimorfizam, jedna oznaka za vise srodnih kategorija koje se različito ponašaju. Imamo dve podvrste:
  - ***preklapanje*** je polimorfizam kada u jednoj klasi mogu da se nađu funkcije sa istim imenom i operatori sa istim znakom.
  - ***koercitivni*** (prinudni) polimorfizam, promenljiva biva prinuđena da se ponaša kao neka druga promenljiva ili kao promenljiva drugog tipa, odnosno prinuđena je da promeni svoj tip. Različito ponašanje iste kategorije.

Polimorfizam

- **Nasleđivanjem klasa možemo konstruisati hijerarhijsku strukturu koja će nam poslužiti za lakše oblikovanje programskog koda. Zajednička svojstva grupe klasa mogu biti sadržana u jednoj ili više nadređenih klasa iz koje su izvedene ostale klase. Imamo li više izvedenih klasa, možemo pojednostaviti rukovanje s njima koristeći pokazivac ili referencu na osnovnu klasu. Pokazivac tipa osnove klase može, osim na objekte osnovne klase, pokazivati i na bilo koji objekt izvedenih klasa, kojima nije jednak po tipu. Zavisno od vrste objekta na koji pokazuje, prevodilac može različito interpretirati upotrebu takvog pokazivača (polimorfizam).**

- **OsnovnaK *p;**
- **IzvedenaK1 a;**
- **IzvedenaK2 b;**
- **p=&a;**
- **p=&b;**
- **p=new IzvedenaK3;**

- **Izvedena klasa moći će koristiti *public* i *protected* metode osnovne klase, ali se te metode također mogu i nadopuniti ili čak ponovo definisati u izvedenoj klasi. Ispred deklaracije metode osnovne klase koja će biti redefinisana u nekoj izvedenoj klasi potrebno je staviti ključnu rec *virtual*. Tada će prevodilac pozvati istoimenu metodu izvedene klase, čak i kada je pokazivač, koji pokazuje na objekat izvedene klase, iz osnovne klase.**

Polimorfizam

- **Primer:**
- **#include<iostream.h>**
- **class A {**
- **public:**
- **void funkcija() {**
- **cout<<”Poziva se A::funkcija()”<<endl;**
- **}**
- **};**
- **class B : public A { // klasa B je izvedena klasa od klase A**
- **public:**
- **virtual void funkcija() {**
- **cout<<”Poziva se B::funkcija()”<<endl;**
- **}**
- **};**
- **class C : public B {**
- **public:**
- **virtual void funkcija() {**
- **cout<<”Poziva se C::funkcija()”<<endl;**
- **}**
- **};**

Polimorfizam

- Pretpostavimo da nam je potrebna nova klasa žena, koja je "jedna vrsta" osobe, samo što još ima i devojačko prezime. Klasa **Zena** biće izvedena iz klase **Osoba**
- I objekti klase **Zena** treba da se "odazivaju" na funkciju **koSi**, ali je teško pretpostaviti da će jedna dama otvoreno priznati svoje godine. Zato objekat klase **Zena** treba da ima funkciju **koSi**, samo što će ona izgledati malo drugačije, svojstveno izvedenoj klasi **Zena**:

```
class Osoba {
public:
  Osoba(char* ime, int godine);        // konstruktor
  virtual void koSi();                 // virtualna funkcija
protected:                             // dostupno naslednicima
  char* ime;                           // podatak: ime i prezime
  int   god;                           // podatak: koliko ima god
};


Osoba::Osoba (char* i, int g) : ime(i), god(g) {}
```

Polimorfizam

```
class Zena : public Osoba {
public:
  Zena(char* ime, char* devojacko, int godine);
  virtual void koSi();   // nova verzija funkcije koSi
private:
  char* devojacko;
};

Zena::Zena (char* i, char* d, int g)
    : Osoba(i,g), devojacko(d) {}

void Zena::koSi () {
  cout<<"Ja sam "<<ime<<", devojacko prezime "<<
    devojacko<<".\n";
}
```

Polimorfizam

- Metodologija koja omogućava da se ista promenljiva u različitim trenucima ponaša kao da ima različite tipove, tako da poziv iste metode nad istom promenljivom u različitim trenucima izaziva različite akcije naziva se polimorfizam.

- Preciznije, ovde govorimo o tzv. jakom polimorfizmu, sa obzirom da se definiše i tzv. slabi polimorfizam, koji prosto podrazumeva da promenljive različitih tipova mogu imati metode istog imena koje rade različite stvari, tako da primena iste metode nad promenljivima različitog tipa izaziva različite akcije.

Polimorfizam

- Na primer, razmotrimo sledeći programski isečak:

```
Student *s = new Student("Paja Patak", 1234);
s->Ispisi();   // indirektan poziv metode
/* pri čemu se prvi put promjenljiva s ponaša poput objekta klase
    “Student”  */
cout << endl;
delete s;
s = new DiplomiraniStudent("Miki Maus", 3412, 2004);
s->Ispisi();
/* pri čemu se drugi put promjenljiva s ponaša poput objekta klase
    “DiplomiraniStudent”  */
delete s;
```

U ovom slučaju, “s” je tipičan primer polimorfne promenljive.

Polimorfizam

- Funkcija članica koja će u izvedenim klasama imati nove verzije deklariše se u osnovnoj klasi kao *virtualna funkcija* (**virtual**). Izvedena klasa može da dâ svoju definiciju virtuelne funkcije, ali i ne mora. U izvedenoj klasi ne mora se navoditi reč **virtual**
- Drugi delovi programa, korisnici klase **Osoba**, ako su dobro projektovani, ne vide nikakvu promenu zbog uvođenja izvedene klase. Oni uopšte ne moraju da se menjaju:

```
// Funkcija ispitaj propituje osobe i
// ne mora da se menja:

void ispitaj (Osoba* hejTi) {
  hejTi->koSi();
}
```

Polimorfizam

- Svojstvo da se odaziva prava verzija funkcije klase čiji su naslednici dali nove verzije naziva se *polimorfizam* (*polymorphism*):

```
Osoba otac("Petar Petrovic",40);
Zena majka("Milka Petrovic","Mitrovic",35);
Maloletnik dete("Milan Petrovic","Petar Petrovic",12);

ispitaj(&otac);          // pozvaće se Osoba::koSi()
ispitaj(&majka);         // pozvaće se Zena::koSi()
ispitaj(&dete);          // pozvaće se Osoba::koSi()


/* Izlaz će biti:
Ja sam Petar Petrovic i imam 40 godina.
Ja sam Milka Petrovic, devojacko prezime Mitrovic.
Ja sam Milan Petrovic i imam 12 godina.
*/
```

Polimorfizam

```
#include<iostream.h>
class CPolygon {
protected:
int width,visina;
public:
void set_values (int a,int b)
{ width=a;visina=b; }
};
class CRectangle: public CPolygon { //klasa naslednik:public osnovna klasa
public:
int area (void)
{ return (width * visina); }};

class CTriangle: public CPolygon { //klasa naslednik:public osnovna klasa
public:
int area (void)
{ return (width * visina/2); }};

main() {
      CRectangle rect;
      CTriangle trgl;
      CPolygon *ppoly1=&rect;
      CPolygon *ppoly2=&trgl;
      ppoly1->set_values(4,5);
      ppoly2->set_values(4,5);
      cout<< "rect area: "<<rect.area()<<endl;
      cout<< "trgl area: "<<trgl.area()<<endl;
return 0;
}
```

Izlaz iz programa:

rect area: 20

trgl area: 10

Klase i prijateljske funkcije

- ***Prijateljske funkcije klasa***
- Prijateljske funkcije neke klase su funkcije koje nisu članovi te klase ali imaju pravo pristupa do privatnih članova te klase. Prijateljske funkcije mogu da budu obične (globalne) funkcije ili da budu metode drugih klasa.
- Da bi funkcija postala prijateljska funkcija neke klase, potrebno je u definiciji te klase da se navede njen prototip ili definicija sa modifikatorom *friend* na početku:

- ***friend tip funkcija (argumenti);***
- ***friend tip funkcija (argumenti) blok***

- Nije bitno da li se funkcija proglašava prijateljskom funkcijom u privatnom ili javnom delu klase, pošto ona nije član posmatrane klase. Ako se navede definicija funkcije, modifikator *inline* se podrazumeva, kao i za članove klase. Uprkos tome, identifikator funkcije neće imati klasni doseg, već pripašće dosegu identifikatora cele klase. Najčešće je to datotečki doseg.

Klase i prijateljske funkcije

- To su slobodne funkcije, koje imaju pristup svim elementima klase (osim polju **this* koji je pokazivač objekta na samog sebe), bilo da su *public* ili *private*. Ova funkcija nije fizički član te klase nego je logički tesno vezana za tu klasu i nalazi se u istom modulu sa klasom.

- Dodavanjem identifikatora *"friend"* ispred deklaracije funkcije, proglašavamo tu funkciju prijateljskom funkcijom klase u kojoj se nalazi. Prilikom proglašavanja funkcija za prijateljske, princip skrivanja informacija ne sme biti narušen. Da se to ne bi desilo klasa proglašava funkciju prijateljskom.

- I klase se mogu proglasiti prijateljskim, a takođe i metode jedne klase u drugoj.

- Korišćenjem prijateljskih funkcija dobijamo povećanu brzinu izvršenja funkcije, jedna funkcija moţe pristupiti članovima nekoliko klasa. Koriste se kod preklapanja operatora.

Prijateljska (friend) metoda

- U C++ postoji mogućnost da i druge klase pristupe privatnim ili zaštićenim atributima i metodama, ako se deklarišu kao prijateljske (*friend*) .

Primer

```
class KONTO
{
friend class KlasaPrijatelj; // friend-deklaracija
private: //Deklaracija atributa i metoda klase KONTO
…
};
```

- Svi objekti klase KlasaPrijatelj mogu pristupiti privatnim atributima i metodama klase KONTO.

Prijateljske funkcije klasa

```
//primer klase CRectangle sa uvodjenjem prijateljske funkcije
#include<iostream.h>
class CRectangle {
int width,visina;
public:
void set_values (int,int);  // funkcija
int area (void) { return (width*visina); }/ /f-ja clanica klase
friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b)
{ width=a; visina=b; }
CRectangle duplicate (CRectangle rectparam)
{CRectangle rectres;
rectres.width=rectparam.width*2;
rectres.visina=rectparam.visina*2;
return (rectres); }

main(){
      CRectangle recta,rectb;//novi objekat, instanca klase
      recta.set_values(2,3);
      cout<< "recta area: "<<recta.area()<<endl;
      rectb=duplicate (recta);
      cout<< "rectb area: "<<rectb.area()<<endl;
}
```

Izlaz iz programa:

recta area: 6

rectb area: 24

Virtualne funkcije

```
#include<iostream.h>
class CPolygon
{protected:
 int width,visina;
 public:
 void set_values (int a,int b)
{ width=a;visina=b; }
 virtual int area (void) =0;};
class CRectangle: public CPolygon //klasa naslednik:public osnovna klasa
{public:
int area (void)
{ return (width * visina); }};
class CTriangle: public CPolygon { //klasa naslednik:public osnovna klasa
public:
int area (void)
{ return (width * visina/2); }};

main() {
      CRectangle rect;
      CTriangle trgl;
      CPolygon *ppoly1=&rect;
      CPolygon *ppoly2=&trgl;
      ppoly1->set_values(4,5);
      ppoly2->set_values(4,5);
      cout<< ppoly1->area()<<endl;
      cout<< ppoly2->area()<<endl;
return 0;}
```

Izlaz iz programa:

20

10

Prijateljske klase

```
//primer klase CRectangle sa uvodjenjem prijateljske klase CSquare
#include<iostream.h>
class CSquare;
class CRectangle {
int width,visina;
public:
int area (void) {return (width*visina);}    //f-ja clanica klase
void convert (CSquare a);
};
class CSquare{
private:
int side;
public:
void set_side (int a)
{ side=a; }
friend class CRectangle; };
void CRectangle::convert (CSquare a)
{ width=a.side;
  visina=a.side; }

main(){
CSquare sqr;
CRectangle rect;
      sqr.set_side(4);
      rect.convert(sqr);
      cout<< "rect area: "<<rect.area()<<endl;
return 0;
}
```

Izlaz iz programa:

rect area: 16

Primeri:

```
// A simple C++ program
#include <iostream.h>
class Counter {
public:
  Counter (int initVal);
  void inc ();
  int val();
private:
  int counter; } ;
Counter::Counter (int initVal) {
  counter = initVal; }
void Counter::inc () {
  counter = counter + 1; }
int Counter::val () {
  return counter;}

  void main () {
  Counter* c1 = new Counter(0);
  Counter* c2 = new Counter(3);
  c1->inc(); c2->inc();
  cout<<c1->val()<<" "<<c2->val()<<endl;
  c1->inc(); c1->inc();
  cout<<c1->val()<<" "<<c2->val()<<endl;
  delete c1;
  delete c2;
}
```

Izlaz iz programa:

1 4

3 4