

# Objektno orjentisano programiranje – C++

## Klase i objekti



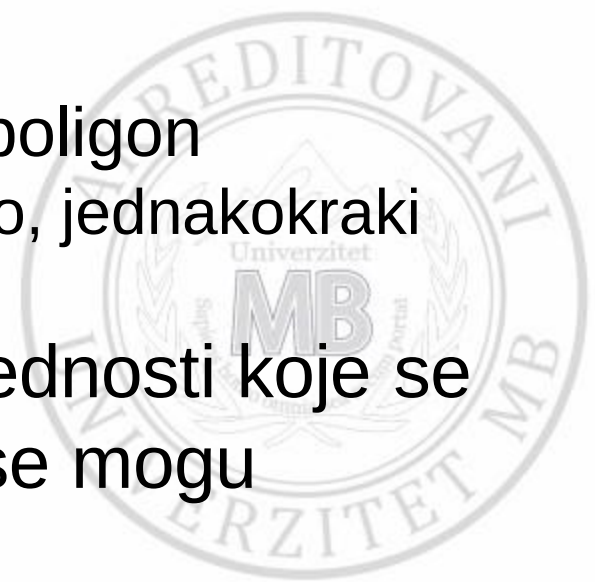
# Teme

- Apstraktni tipovi podataka Abstract Data Types
- Objektno-Orijentisano Programiranje-OOP
- Uvod u klase i objekte
- Definisanje funkcija članica (engl. member function)
- Privatne funkcije članice
- Konstruktori
- Destruktori
- Prenošnje (engl. passing) objekata funkcijama
- Odvajanje definicija od implementacije klasa i koda klijenta



# Apstraktni tipovi podataka

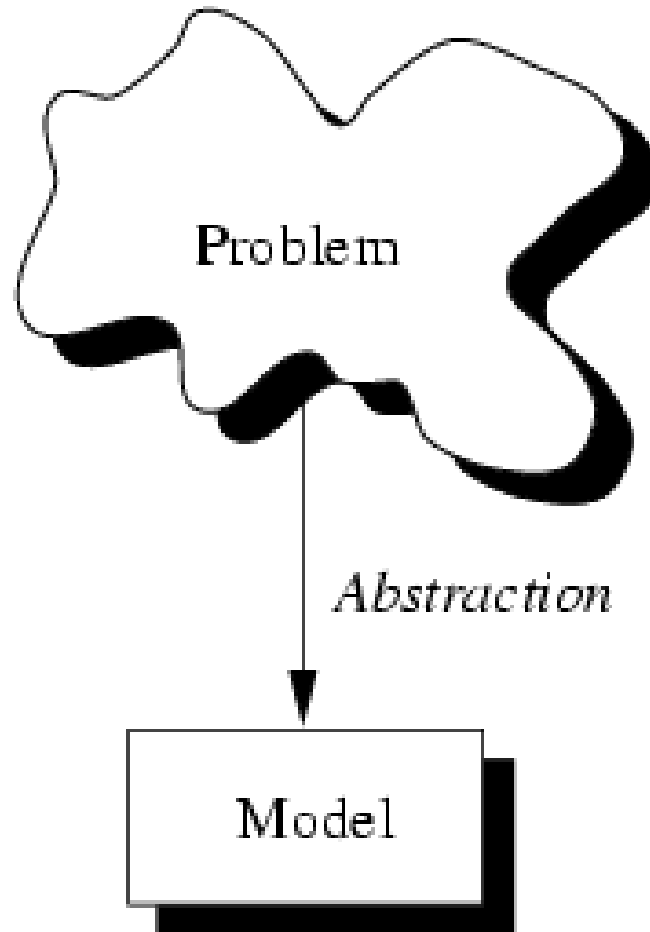
- **Apstrakcija:** uočavanje zajedničkih osobina objekata i njihovo grupisanje u klasu
- **Apstrakcija:** definicija koja obuhvata opšte karakteristike, bez detalja
  - apstraktni trougao je trostranični poligon
    - specifičan trokut može biti trougao, jednakokraki ili jednakostraničan trougao
- **Vrsta podataka:** određuje vrstu vrednosti koje se mogu uskladištiti i operacija koje se mogu izvršavati nad njima



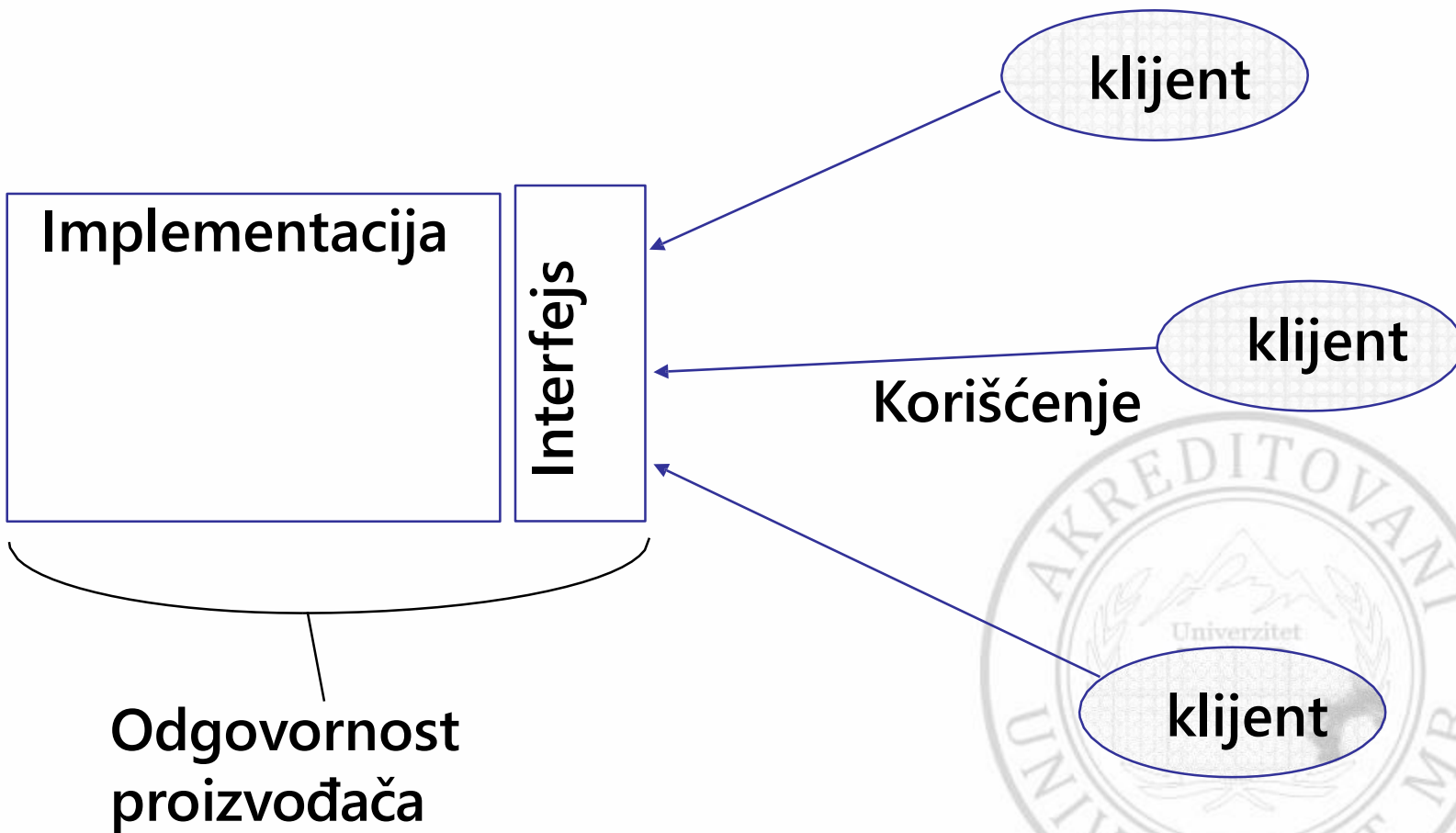
## Apstraktni tipovi podataka (2)

- Vrste podataka koje je kreirao programer i koje određuju
  - dozvoljene vrednosti koje se mogu uskladištiti
  - operacije koje se mogu izvršiti nad tim vrednostima
- Korisnik apstraktnog tipa podataka (engl. Abstract Data Type-ADT) ne treba da zna detalje implementacije (npr. kako se podaci čuvaju ili kako se izvode operacije nad podacima)

# Kreiranje modela iz apstrakcije



# Klijenti i proizvođači



# Strukturalno i OO programiranje

- Da bi se pisali OO programi, moraju se koristiti **klase i objekti**
- OO orjentisana aplikacija sastoji se od **skupa objekata** koji međusobno komuniciraju slanjem poruka i odgovaranjem na njih
- OOP zasniva se na principu definisanja sopstvenih tipova podataka



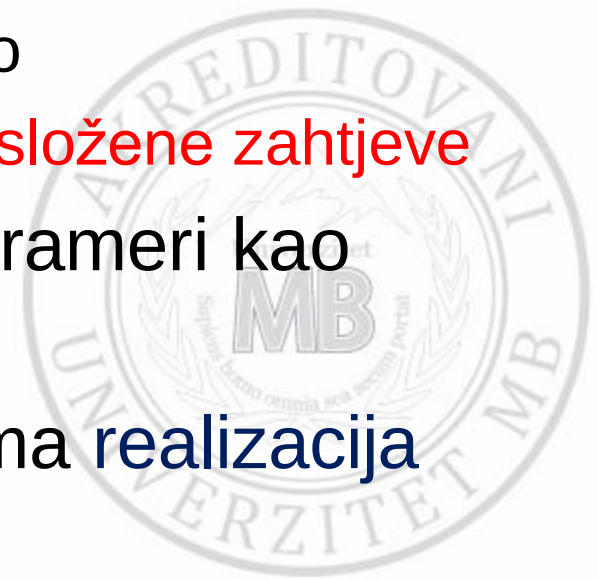
# Objektno orjentisano programiranje-OOP

- **Proceduralno programiranje** koristi varijable za spremanje podataka, fokusira se na procese/funkcije koje se javljaju u programu. Podaci i funkcije su odvojeni i različiti.
- **Objektno orjentisano programiranje-OOP** je bazirano na objektima koji kapsuliraju podatke i funkcije koje manipulišu podacima.



# Zašto je nastalo OOP?

- OOP je pristup realizaciji softvera kao modela realnog sveta
  - odgovor na softversku krizu
    - softver se **projektuje duže** nego što je predviđeno
    - **košta više** nego što je predviđeno
    - **ne zadovoljava sve postavljene složene zahtjeve**
- OO programiranje su smislili programeri kao način da sebi olakšaju život
- Najvažnije je projektovanje, a sama realizacija se odlaže za kasnije



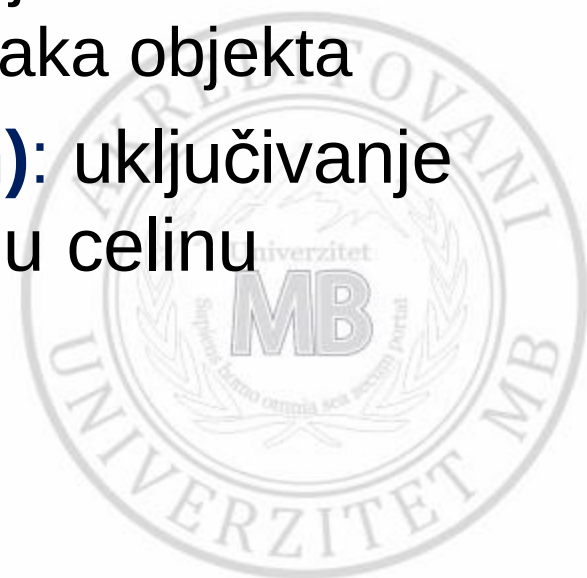
# OOP terminologija

- **Objekat**: softverska celina koja objedinjuje podatke i funkcije koje manipulišu podacima u **pojedinačnoj jedinici**
- **Atributi**: vrste podataka nekog objekta, uskladištene u varijablama članicama
- **Funkcije članice** (engl. member functions) ili metode: procedure/funkcije koje manipulišu atributima neke klase



## OOP terminologija (2)

- **Skrivanje podataka** (engl. data hiding): ograničavanje pristupa određenim članovima objekta.
  - namera je omogućiti samo funkcijama članicama direktan pristup i menjanje podataka objekta
- **Kapsulacija** (engl. encapsulation): uključivanje podataka i metoda objekta u jednu celinu



# Šta je klasa?

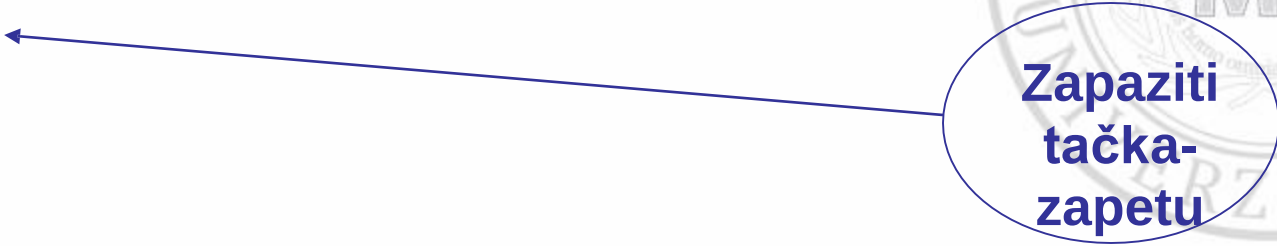
- **Klasa** je korisnički definisan tip podataka kojim se **modeliraju** objekti sličnih svojstava
- Karakteristika - uočavanje zajedničkih osobina (svojstava) objekata i njihovo grupisanje u klasu (apstrakcija)



## Šta je klasa? (2)

- **Klasa:** Tip podataka za definisanje objekata koji kreira programer
- Način kreiranja objekata
- Format definisanja klase

```
class Name  
{  
    izjave;  
    izjave;  
};
```



Zapaziti  
tačka-  
zapetu

## Šta je klasa? (3)

- Klasa je opisana svojim:
  - **atributima** (podacima članovima)
  - **operacijama** (funkcijama članicama, metodama)
- Atributi najčešće predstavljaju unutrašnjost klase, tj. njenu realizaciju, a metode njen interfejs, odnosno ono što se može raditi sa njom



# Šta je klasa? (4)

```
class Point { //definicija klase
    int x, y; // atributi
public:      // ovde počinje interfejs
    // funkcije članice
    void setX(const int val);
    void setY(const int val);
    int getX() { return x; }
    int getY() { return y; }
};
Point ObjectName; //kreiranje objekta
```

# Primer

- **Varijable članice**  
(atributi)

```
int side;
```

- **Funkcije članice**

```
void setSide(int s)
```

```
{
```

```
    side = s;
```

```
}
```

```
int getSide()
```

```
{
```

```
    return side;
```

```
}
```

- **Varijabla **side** objekta**  
Square

- **Funkcije objekta Square**

**setSide** – određuje stranicu kvadrata

**getSide** – vraća stranicu kvadrata



# Kontrola pristupa članovima klase

- Članovi (podaci ili metode) klase koji se nalaze iza ključne riječi **private**: zaštićeni su od pristupa spolja (oni su sakriveni, kapsulirani)
- Da bi se deo klase učinio javnim, tj. da bi bio vidljiv u kodu izvan klase, navodi se ključna riječ **public**:
  - članovi iza ključne riječi **public**: dostupni su izvan klase i nazivaju se javnim članovima

# Funkcije članice (metode) klase

- Zaštićenim (**private**) članovima klase obično se pristupa preko javnih (**public**) funkcija članica
- Kada **funkcija članica** koristi privatni podatak član svoje klase, pristupa mu direktno, bez korišćenja operatora .
- Funkcija članica se uvek poziva za neki objekat svoje klase

```
char * Osoba:: koSi () {  
    return imePrezime;  
}
```



# Specifikatori pristupa – public i private

- Koriste se za kontrolu pristupa članovima klase.
- Mogu biti navedeni **u bilo kojem redosledu** u klasi, mogu se pojaviti više puta u klasi
- Ako nisu određeni (engl. default), zadani su kao **private**



# Specifikatori pristupa: Primer

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

```
class Example {  
    // private as default ...
```

```
public:
```

```
    // what follows is public until ...
```

```
private:
```

```
    // ... here, where we switch back to private ...
```

```
public:
```

```
    // ... and back to public.
```

```
};
```



# Primer klase #1

```
class Square {  
    private:  
        int side;  
    public:  
        void setSide(int s) {  
            side = s;  
        }  
        int getSide() {  
            return side;  
        }  
};
```

specifikatori  
pristupa



## Primer klase #2

```
class Time {  
    public:  
        void setTime(int, int, int);  
        void getTime(int&, int&, int&);  
        void printTime();  
        bool equalTime(const clockType&);  
    private:  
        int hr;  
        int min;  
        int sec;  
};
```



# Šta je objekat?

- **Objekat** je primerak (instanca) klase
- Pošto je klasa tip, objekti se mogu smatrati promenljivama tog tipa u programu
- Objekti klase se deklarišu navođenjem imena klase iza koga slede nazivi objekata razdvojeni zarezima
- Članovima klase pristupa se pomoću znaka (.).

Osoba profesor, student, direktor;

profesor.koSi();

direktor.koSi ();



## Šta je objekat? (2)

- Svaki objekat klase ima sopstveni skup vrednosti članova svoje klase

profesor.imePrezime = "Petar Petrović"

profesor.godine = 35;

profesor.imePrezime = "Jovan Jovanović";

profesor.godine = 21;



# Primer klase, inline funkcija i objekata

```
#include <iostream>
using namespace std;
class temp {
private:
    int data1; float data2;
public:
    void int_data(int d){
        data1=d;
        cout<<"Number: "<<data1;
    }
    float float_data(){
        cout<<"\nEnter data: ";
        cin>>data2;
        return data2;
    }
};
```

```
int main(){
    temp obj1, obj2;
    obj1.int_data(12);
    cout<<"You entered "
        <<obj2.float_data();
    int k ;
    cin>>k;
    return 0;
}
```

# Implementacija funkcija članica unutar klase

- Funkcija članica definisana unutar klase naziva se **inline** funkcija
- Samo vrlo kratke funkcije, treba da budu **inline** funkcije kao što je sledeća funkcija

```
int getSide()  
{  
    return side;  
}
```



# Inline funkcije (ugrađene funkcije)

- Funkcije članice su podrazumevano **inline** ako se njihova definicija navodi unutar deklaracije klase
- Može se deklarirati **kao inline i van deklaracije klase** – navođenjem **inline** ispred deklaracije funkcije
- Mehanizam inline treba pažljivo primenjivati, samo za vrlo jednostavne funkcije



# Primer inline funkcije članice

```
class Square
{
private:
    int side;
public:
    void setSide(int s)
    {
        side = s;
    }
    int getSide()
    {
        return side;
    }
};
```

inline  
functions



## Inline funkcije (2)

```
class Brojac {  
    int i;  
public:  
    Brojac (int x) { //inline konstruktor  
        i=x;  
    }  
    int broj () { //inline funkcija članica  
        return ++i;  
    }  
}
```



## Inline funkcije (3)

```
#include <iostream>
using namespace std;
class Brojac {
public:
    int i;
    Brojac (int x) { //inline konstruktor
        i=x;
    }
    int broj () { //inline funkcija članica
        return ++i;
    }
};
```

```
int main () {
    Brojac b(19);
    cout<<b.i<<endl;
    cout<<b.broj()<<endl;
    std::cin.get();
    return 0;
}
```



# Operator dosega (engl. scope)

- Operator (::) povezuje ime klase sa njenim članom da bi kompajleru saopštio sa kojom klasom je povezan taj član
- Pošto više različitih klasa mogu da imaju funkcije članice istog imena, operator :: je neophodan
- Operator dosega mora biti korišćen u definiciji konstruktora zato što se ona nalazi izvan deklaracije klase

```
myClass :: MyClass() {} // konstruktor klase
```

# Implementacija funkcija članice van klase

- Staviti prototip funkcije u definiciju klase
- U definiciji funkcije staviti ime klase i operator “::” (engl. **scope resolution operator** ::) pre imena funkcije

```
int Square::getSide()
```

```
{  
    return side;  
}
```



# Implementacija funkcija članice van klase (2)

```
class Square {  
private:  
int side;  
public:  
    void setSide(int);  
    int getSide();  
};  
void Square::setSide (int s){  
    side = s;  
}  
int Square::getSide() {  
    return side;  
}
```



# Konstruktori klasa

- Konstruktor služi za inicijalizovanje objekta
- Konstruktor je funkcija članica koja ima isto ime kao klasa, a nema povratni tip
  - konstruktor može, ali ne mora imati argumente
  - konstruktor se može **preklopiti**, tj. za istu klasu može se definisati više konstruktora koji se razlikuju po tipu ili broju argumenata



# Implementacija konstruktora unutar klase

```
class Point {  
    int _x, _y;  
    public:  
        Point() {  
            _x = _y = 0;  
        }  
        Point(const int x, const int y) {  
            _x = x;  
            _y = y;  
            .....  
        };
```



# Implementacija konstruktora i funkcije članice van klase

```
Osoba::Osoba(char * ime, int god){  
    imePrezime =ime;  
    godine= god;  
}  
char * Osoba:: koSi () {  
    return imePrezime;  
}
```



# Konstruktori – Primeri

## Inline:

```
class Square
{
    ...
public:
    Square(int s)
    {
        side = s;
    }
    ...
};
```

## Definicija van klase

```
Square(int); //prototip dat u klasi
Square::Square(int s)
{
    side = s;
}
```

# Podrazumevani konstruktor

- Konstruktori mogu imati bilo koji broj parametara uključujući i nijedan
- Podrazumevani konstruktor nema argumenata zbog toga:
  - što nema parametara
  - svi parametri imaju zadatu vrednost



## Podrazumevani konstruktor (2)

```
class Square
{
private:
int side;

public:
Square()
{
    side = 1;
}

...
};
```

Nema  
parametre

// podrazumevani konstruktor



# Podrazumevani konstruktor (3)

```
class Square
```

```
{  
    private:  
        int side;
```

```
    public:
```

```
        Square(int s = 1) // podrazumevani konstruktor
```

```
        {  
            side = s;  
        }
```

```
        ...
```

```
};
```

Ima  
parametar,  
ali ima i  
zadanu  
vrijednost



# Podrazumevani konstruktor (4)

```
#include <iostream>
class myClass {
public:
    int x;
};
int main() {
    myClass m; // myClass () {};
    m.x=5;
    std::cout<<"x ="<<m.x<<std::endl;
}
```



# Podrazumevani konstruktor (5)

```
#include <iostream>
using namespace std;
class Line{
public:
    void setLength(double len);
    double getLength( void );
private:
    double length;
};
```

```
void Line::setLength(double len) {
    length=len;
}
double Line::getLength( void ){
    return length;
}
int main () {
    Line l; //poziv Line ();
    l.setLength(43.0);
    cout <<"length ="
        << l.getLength();
    cin.get();
    return 0;
}
```

## Podrazumevani konstruktor (6)

- Ako nijedan konstruktor nije eksplicitno deklarisan u definiciji klase, **C++ kompajler će sam generisati podrazumevani konstruktor**
- Podrazumevani konstruktor je **javan** (public)
- Ako klasa **ima konstruktore** koje je kreirao programer, programer onda **mora napisati podrazumevani konstruktor**



# Podrazumevani konstruktor (7)

```
class Date {  
private:  
    int m_year;  
    int m_month;  
    int m_day;  
public:  
    Date(int year, int month, int day {  
        m_year = year;  
        m_month = month;  
        m_day = day;  
    }  
};
```

```
int main(){  
    Date date;  
    // nema default konstruktor  
    return 0;  
}
```

**error: Can't instantiate object because default constructor doesn't exist**



# Podrazumevani konstruktor (8)

```
class Date {  
private:  
    int m_year;  
    int m_month;  
    int m_day;  
public:  
    Date () {}  
    Date(int year, int month, int day) {  
        m_year = year;  
        m_month = month;  
        m_day = day;  
    }  
};
```

```
int main()  
{  
    Date date;  
    .....  
    return 0;  
}
```



# Konstruktori (sa parametrima)

```
include <iostream>
using namespace std;
class Osoba {
char *imePrezime;
    int godine;
public:
    char *koSi();
    Osoba (char *, int);
};
Osoba::Osoba(char * ime, int god){
    imePrezime =ime;
    godine= god;
}
```

```
char * Osoba:: koSi () {
    return imePrezime;
}
int main () {
    Osoba profesor
        ("Petar Petrovic", 47);
    Osoba student
        ("Jovan Jovanović", 20);
    cout<<student.koSi()<<endl;
}
```



# Destruktori klase

- Funkcija članica koja se automatski poziva kada objekat prestaje da živi zove se **destruktor**
- Ime destruktora je isto kao ime klase, sa prefiksom **~ (tilde)**
- Podrazumevani (engl. default) destruktor nema povratni tip, nema argumente, ne može biti preklopljen i javan je (public)
- Programer može napraviti i sopstvenu verziju destruktora



# Primer konstruktora i destruktora

```
#include <iostream>
using namespace std;
class mojaKlasa {
public:
    int i;
    mojaKlasa (); //konstruktor
    ~mojaKlasa (); //destruktor
};
mojaKlasa ::mojaKlasa () {
    i=10;
}
```

```
mojaKlasa::~mojaKlasa () {
    cout<< "Uništavam ..."
        <<endl;
}
int main () {
    mojaKlasa objekat1, objekat2;
    cout << objekat1.i << " "
        << objekat2.i <<endl;
    cin.get();
    return 0;
}
```



# Pokazivač this

- Svaka funkcija članica koja se izvršava sadrži skriveni pokazivač na objekat koji ju je pozvao
  - taj pokazivač zove se **this**
- Kada u funkciji članici koristimo direktno neki podatak član pristupom preko imena, kompajler automatski razume da koristimo **(\*this).ime\_člana**, što se kraće piše kao **this->ime\_člana**



# Operator this: Primer

```
#include <iostream>
using namespace std;
class mojaKlasa {
    int x;
public:
    void setX(int x) {
        this->x=x;
    }
    int getX() {
        return x;
    }
};
```

```
int main () {
    int k=7;
    mojaKlasa objekat;
    objekat.setX(15);
    cout<<"i = "<<objekat.getX();
    cin.get();
    return 0;
}
```



## Pokazivač this (2)

- Ako je potrebno, pokazivač **this** se može i direktno pozvati u funkciji članici, npr. ako ona vraća pokazivač na tekući objekat

```
Osoba::Osoba(char *imePrezime, int godine) {  
    this→ imePrezime = imePrezime;  
    this→ godine = godine;  
}
```



# Ugneždene klase

```
#include <iostream>
using namespace std;
class A{
public:
    void p() {
        B b(3);
        cout << "i = "
             << b.getl()<< endl;
    }
private:
    class B {
    public:
        B(int newl) {
            i = newl;
        }
    };
};
```

```
int getl() {
    return i;
}
private:
    int i;
};

int main(){
    A a; //podrazumijevani
        //konstruktor
    a.p();
    return 0;
}
```

i = 3

# Nizovi objekata

- Objekti se mogu smeštati u niz isto kao prosti tipovi, i pristupa im se uobičajenom sintaksom indeksiranja



# Nizovi objekata (2)

```
#include <iostream>
using namespace std;
class mojaKlasa {
    int x;
public:
    void setX(int i) {
        x=i;
    }
    int getX() {
        return x;
    }
};
```

```
int main () {
    mojaKlasa objekti[4];
    int i;
    for (int i=0;i<4;i++)
        objekti [i].setX(i);

    for (int i=0;i<4;i++)
        cout<< "objekti ["<<i<<"].getX() = "
            <<objekti[i].getX()<<endl;
    cin.get();
    return 0;
}
```

```
objekti [0].getX() = 0
objekti [1].getX() = 1
objekti [2].getX() = 2
objekti [3].getX() = 3
```

# Inicijalizovanje nizova objekata

- Ako klasa ima konstruktor sa argumentima, niz objekata se može inicijalizovati



# Inicijalizacija nizova objekata

```
#include <iostream>
using namespace std;
class mojaKlasa {
    int x;
public:
    mojaKlasa (int i) {
        x=i;
    }
    int getX() {
        return x;
    }
};

int main () {
    mojaKlasa objekti[4] ={-1,-2,-3, -4};
    for (int i=0;i<4;i++)
        cout << "objekti ["<<i<<"].getX() = "
            <<objekti[i].getX()<<endl;
    cin.get();
    return 0;
}
```

```
objekti [0].getX() = -1
objekti [1].getX() = -2
objekti [2].getX() = -3
objekti [3].getX() = -4
```

# Pokazivači na objekte

- Za pristup pojedinačnim članovima koristi se operator ->

**p -> funkcija() je ekvivalentno sa (\*p).funkcija().**

- Zagrade oko \*p su neophodne zato što **operator (.) ima veći prioritet od operatora dereferenciranja (\*)**
- Adresa objekta se dobija primenom operatora &.
- Reference na objekte se deklarišu na isti način kao reference na obične promenljive

## Pokazivači na objekte (2)

```
#include <iostream>
using namespace std;
class mojaKlasa {
    int x;
public:
    void setX (int i) {
        x=i;
    }
    void prikaziMe() {
        cout<<x<<endl;
    }
};
```

```
int main () {
    mojaKlasa objekat, *pobjekat;
    objekat.setX(1);
    objekat.prikaziMe();
    pobjekat = &objekat;
    pobjekat->setX(10);
    pobjekat->prikaziMe();
    cin.get();
    return 0;
}
```

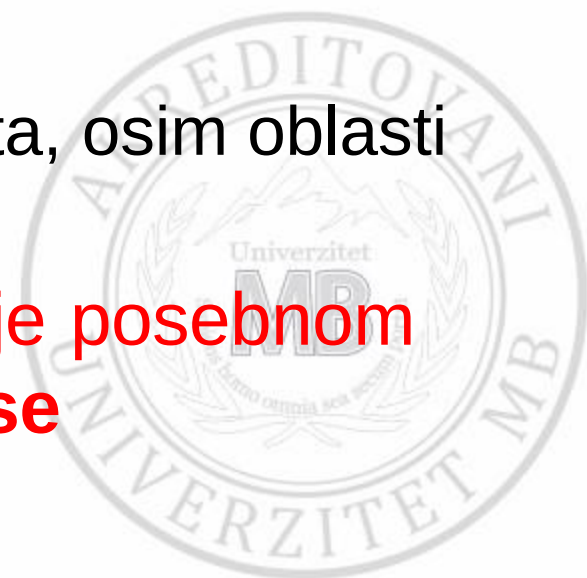
# Zajednički članovi klase

- Pri kreiranju objekata klase, za svaki objekat se kreira poseban **komplet podataka članova**
- Moguće je definisati podatke članove za koje postoji samo jedan **primerak za celu klasu**, tj. za sve objekte klase
- Ovakvi članovi nazivaju se **statičkim članovima**, i deklarišu se pomoću reči **static**

```
class X {  
    static int i; //postoji samo jedan i za celu klasu  
    int j;       //svaki objekat ima svoje j  
    ....  
};
```

## Zajednički članovi klase (2)

- Statički član klase
  - ima životni vek kao i globalna promenljiva: nastaje na početku programa i traje do kraja programa
  - ima sva svojstva globalnog objekta, osim oblasti važenja klase i kontrole pristupa
- Statički član mora da se inicijalizuje posebnom deklaracijom **van deklaracije klase**



## Zajednički članovi klasa (3)

- Statičkom članu može da se pristupi iz funkcije članice, ali i van funkcija članica
  - pristupa mu se preko operatora `::` (`X::i`)
  - smanjuju potrebu za globalnim objektima i tako povećavaju čitljivost programa, jer je, za razliku od globalnih objekata, moguće ograničiti pristup do njih
  - zajednički članovi logički pripadaju klasi i "upakovani" su u nju

# Zajedničke funkcije članice

- **Funkcije članice** mogu da se deklarišu kao zajedničke za celu klasu, dodavanjem reči **static** ispred deklaracije funkcije članice
  - ne poseduju pokazivač **this** i ne mogu neposredno (bez pominjanja konkretnog objekta klase) koristiti nestatičke članove klase
- **Statičke funkcije mogu neposredno koristiti samo statičke članove te klase**
- Statičke funkcije predstavljaju **operacije klase**, a ne svakog posebnog objekta
- **C++ ne podržava statičke konstruktore**

# Operator new ()

- Operator new kreira dinamički objekat, tj. alokira memoriju za njega
- Operator new za operand ima identifikator tipa i eventualne inicijalizatore (argumente konstruktora)
- **Operator new vraća pokazivač na dati tip**
- Ako nema dovoljno slobodne heap memorije da bi se zadovoljio alokacioni zahtjev, new vraća **null pokazivač**

Complex \* pc1 =new Complex(1.3, 5.6);

Complex \* pc2 =new Complex(-1.0, 0.);

## Operator new (2)

- Objekat kreiran pomoću operatora **new** naziva se **dinamički objekat**, jer mu je životni vek poznat tek u vreme izvršavanja programa.
- Ovakav objekat nastaje kada se **izvrši operator new**, a traje sve dok se ne ukine operatorom **delete** (može da traje i po završetku bloka u kome je nastao)



# Operator delete ()

- Operator delete ima pokazivač na neki tip.
- Ovaj pokazivač mora da ukazuje na objekat nastao pomoću operatora new
- Operator delete poziva destruktor za objekat na koji ukazuje pokazivač, a zatim oslobađa zauzeti prostor
- **Operator delete vraća void**



# Operatori new i delete: Primeri

```
#include <iostream>
using namespace std;
int main (){
    double* p = NULL;
    p = new double; // zahtev za memorijom za varijablu
    *p = 20000;      // uskladištiti vrednost
    cout << "Vrednost za *p : " << *p << endl;
    delete p;        // osloboditi memoriju
    cin.get();
    return 0;
}
Vrednost za *p : 20000
```



# Dodela jednog objekta drugom

- Moguće je dodeliti jedan objekat drugom objektu ako pripadaju istoj klasi
  - u tom slučaju dodela se obavlja direktnim **kopiranjem sadržaja po principu član po član**
- Nakon dodele, **postojeće dva posebna objekta** čiji će sadržaj biti identičan, pa naknadne izmene članova jednog objekta neće imati uticaja na drugi

# Dodela jednog objekta drugom (2)

```
#include <iostream>
using namespace std;
class myClass {
public:
    int n;
};
int main (){
    myClass A;
    A.n = 5;
    myClass B = A;
    B.n = 7;
    cout << A.n << " " << B.n << endl;
    cin.get();
    return 0;
}
```



# Objekat kao povratni tip funkcije

- Kada funkcija vraća objekat, pravi se privremeni objekat koji sadrži povratnu vrednost
- Čim se vrednost vrati, objekat se uništava pozivom destruktora



# Objekat kao povratni tip funkcije (2)

```
#include <iostream>
using namespace std;
class data{
public:
    int a,b;
    data sum (data,data);
    void get();
};
data sum(data a1,data a2){
    data a3;
    a3.a=a1.a+a2.a;
    a3.b=a1.b+a2.b;
    return a3;
}
```

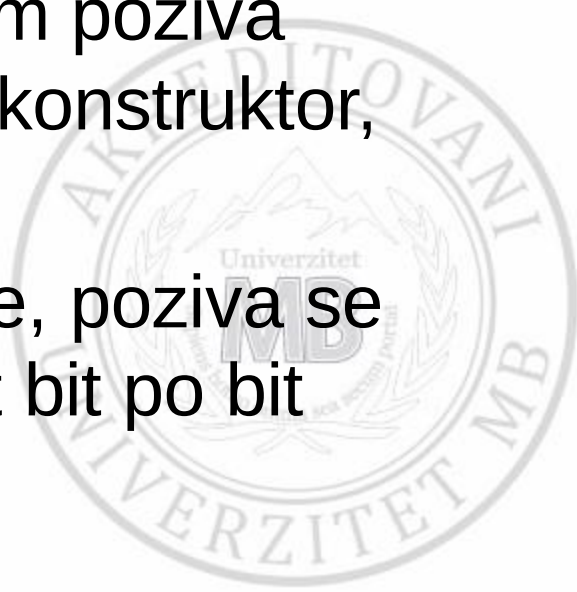
```
int main(){
    data a1,a2,a3;
    a1.get();
    a2.get();
    a3=sum(a1,a2);
    cout<<"a= "<<a3.a<<endl;
    cout<<"b= "<<a3.b<<endl;
    cin.get();
    return 0;
}

void data::get(){
    a=44;
    b=27;
}
```

|                |
|----------------|
| a= 88<br>b= 54 |
|----------------|

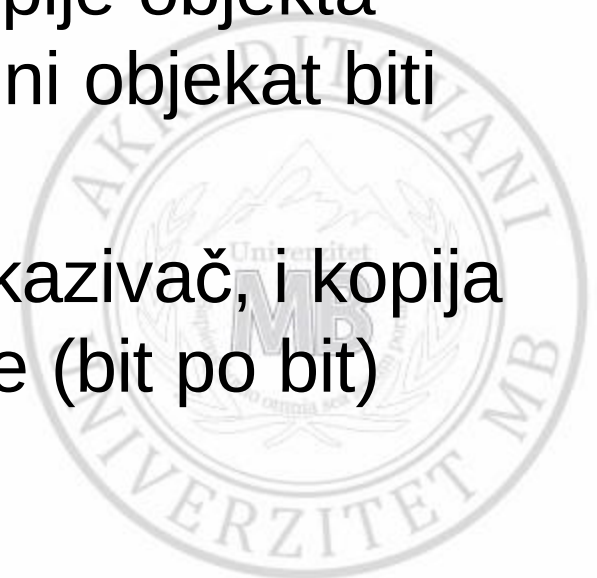
# Prosleđivanje objekata funkciji

- Objekti se u funkcije prenose standardno po vrednosti, tj. pravi se njihova kopija (novi objekat) koji postaje parametar
- Kada se pravi kopija objekta tokom poziva funkcije, ne poziva se standardni konstruktor, već tzv. **konstruktor kopije**
- Ako klasa nema konstruktor kopije, poziva se **default verzija** koja kopira objekat bit po bit



# Prosleđivanje objekata funkciji - problemi

- Ako **objekat koristi neki resurs** (npr. datoteku, memoriju), **ako se u funkciju prenosi po vrednosti**, prilikom uništavanja kopije objekta uništiće se i resurs, pa će originalni objekat biti oštećen
- Ako objekat sadrži član koji je pokazivač, i kopija koju je napravio konstruktor kopije (bit po bit) pokazivaće na isti objekat



# Rešenje problema

Rešenje br.1:

- Prenositi objekte po referenci (adresi)
  - ovo nije uvek moguće

Rešenje br. 2

- Redefinisati **konstruktor kopije** tako da radi ono što želimo



# Prosleđivanje objekata po adresi

```
#include <iostream>
using namespace std;
class mojaKlasa {
    int x;
public:
    mojaKlasa (int i) {
        x=i;
        cout<<"U konstruktoru" <<endl;
    }
    ~mojaKlasa () {
        cout<<"U destrukturu" <<endl;
    }
}
```

```
void setX (int i) {
    x=i;
}
int getX() {
    return x;
}
};
void prikazi (mojaKlasa &ref){
    cout<<ref.getX()<<endl;
}
```

# Prosljeđivanje objekata po adresi (2)

```
void promeni (mojaKlasa &ref){  
    ref.setX(100);  
}
```

```
int main () {  
    mojaKlasa obj (10);  
    prikazi (obj);  
    promeni (obj);  
    prikazi (obj);  
    cin.get();  
    return 0;  
}
```

U konstruktoru  
10  
100

# Konstruktor kopije (engl. copy constructor)

- U C++ jeziku postoje dva slučaja kada se vrednost jednog objekta dodeljuje drugom:
  - **dodela** (objekat kome se dodeljuje već postoji,  $a2=a1$ )
  - **inicijalizacija** (`mojaKlasa x, y; x=y`)
- Konstruktor kopije se primenjuje samo prilikom **inicijalizacija** koje se dešavaju u tri slučaja:
  - inicijalizacija jednog objekta drugim istog tipa
  - kopiranja objekta da bi se preneo kao argument funkcije
  - kopiranja objekta da se vrati iz funkcije
- Konstruktor kopije se ne poziva prilikom **dodela**

## Konstruktor kopije (2)

mojaKlasa x;

//slučajevi kada se poziva konstruktor kopije

mojaKlasa y=x; //eksplicitna inicijalizacija

func1(y); //y kao parametar funkcije

y = func2(x); //y prihvata privremeni objekat

mojaKlasa x;

mojaKlasa y;

x=y; //ovde se ne poziva konstruktor kopije

- Argument konstruktora ne može da bude objekat već uvek **referenca** na objekat;



## Konstruktor kopije (3)

- C ++ poziva konstruktor kopiranja da napravi kopiju objekta u svakom od navedenih slučajeva podrazumevani.
- Ako ne postoji konstruktor kopiranja definisan za klasu, C ++ koristi podrazumevani konstruktor kopiranja (engl. default copy constructor) koji kopira svako polje, odnosno, kreira **plitku kopiju** (engl. shallow copy).
- Konstruktor kopije je specijalan, preklopljen konstruktor klase koji se poziva **uvek kada se pravi kopija objekta**

## Konstruktor kopije (4)

- Ako konstruktor kopije nije definisan u klasi, sam kompajler ga definiše.
- Ako klasa ima promenljivu pokazivača, a ima dinamičku alokaciju memorije, onda mora da ima konstruktor kopije. Najčešći oblik kopiranja konstruktora je dat kao:

```
classname (const classname &obj) {  
    // body of constructor  
}
```



## Konstruktor kopije (5)

// datoteka Point.h

```
class Point {  
    public:  
        ...  
        // konstruktor kopiranja  
        Point(const Point& p);  
        ...
```

// datoteka Point.cpp

```
Point::Point(const Point& p){  
    x = p.x;  
    y = p.y;  
}  
...
```

// datoteka my\_program.cpp

```
// poziv default konstruktora  
Point p;
```

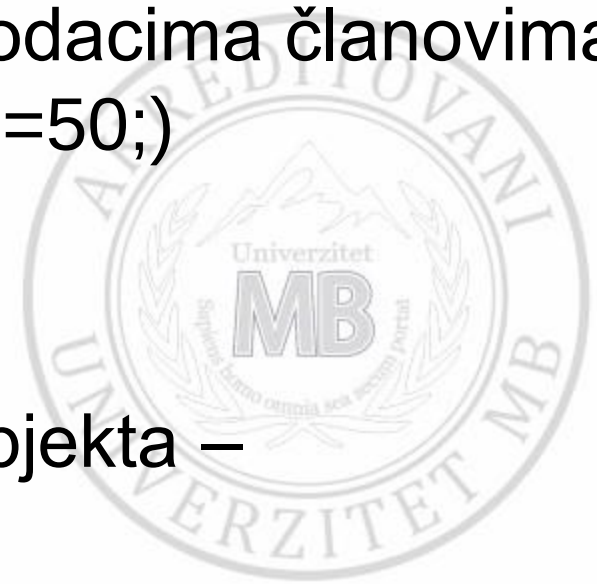
```
// poziv konstruktoru kopiranja  
Point s = p;
```

```
...  
// dodela, ne konstruktor kopiranja  
p = s;.
```



# Zaključak

- Deklaracija klase –  
`class imeKlase {lista članova}; lista objekata`
- Kreiranje objekta – `imeKlase objekat;` (npr. Osoba profesor)
- Povezivanje objekta sa njegovim podacima članovima – `objekat.član` (npr. profesor.godine=50;)
- Implementacija funkcije članice –  
`tip imeKlase :: funkcija () {}`
- Pozivanje funkcije članice preko objekta –  
`objekat.funkcija ()`



## Zaključak (2)

- Kontrola pristupa članovima klase – **public i private**
- Konstruktor – funkcija članica klase koja se automatski poziva kada se kreira objekat. Ima isto ime kao klasa.
- Destruktor – funkcija članica klase koja uništava objekat.
- **static podatakČlan** – zajednički član klase (svi objekti te klase dele taj član).



**Kraj prezentacije**

**HVALA NA PAŽNJI!**

