Objektno orjentisano programiranje – C++

Nasleđivanje, virtuelne funkcije i polimorfizam

Teme

- Osnovna i izvedene klase
- Kontrola pristupa članovima klase
- Funkcije prijatelji
- Konstruktori i destruktori izvedenih klasa
- Višestruko nasleđivanje
- Pokazivači na objekte izvedenih klasa
- Polimorfizam
- Virtuelne funkcije
- Nadjačavanje funkcija u izvedenim klasama
- Apstraktne klase

Uvod

- Nasleđivanje omogućava novoj klasi da nasledi osobine neke postojeće klase
- Klasa iz koje se izvodi je **osnovna klasa** (base class, superclass, parent) ili generička klasa
- **Izvedena klasa** (derived class, subclass, child) nasleđuje sve osobine osnovne klase, ali ima i sopstvene, jedinstvene osobine
- Opšti oblik nasleđivanja je:

```
class izvedena_klasa : access osnovna_klasa {
    //telo izvedene klase
};
```

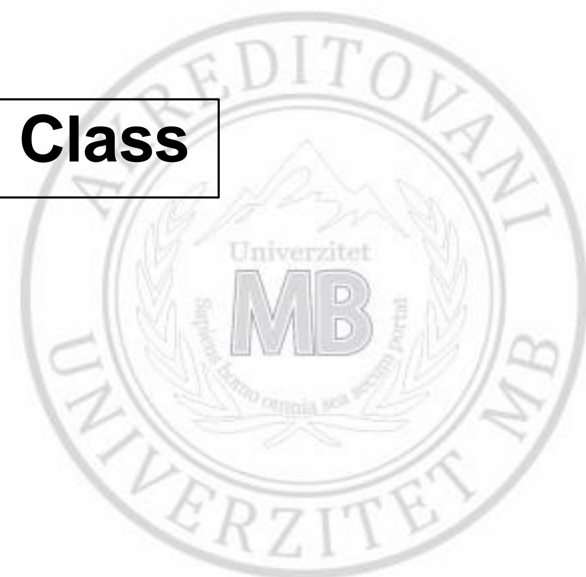
Sintaksa i notacija nasleđivanja

Dijagram

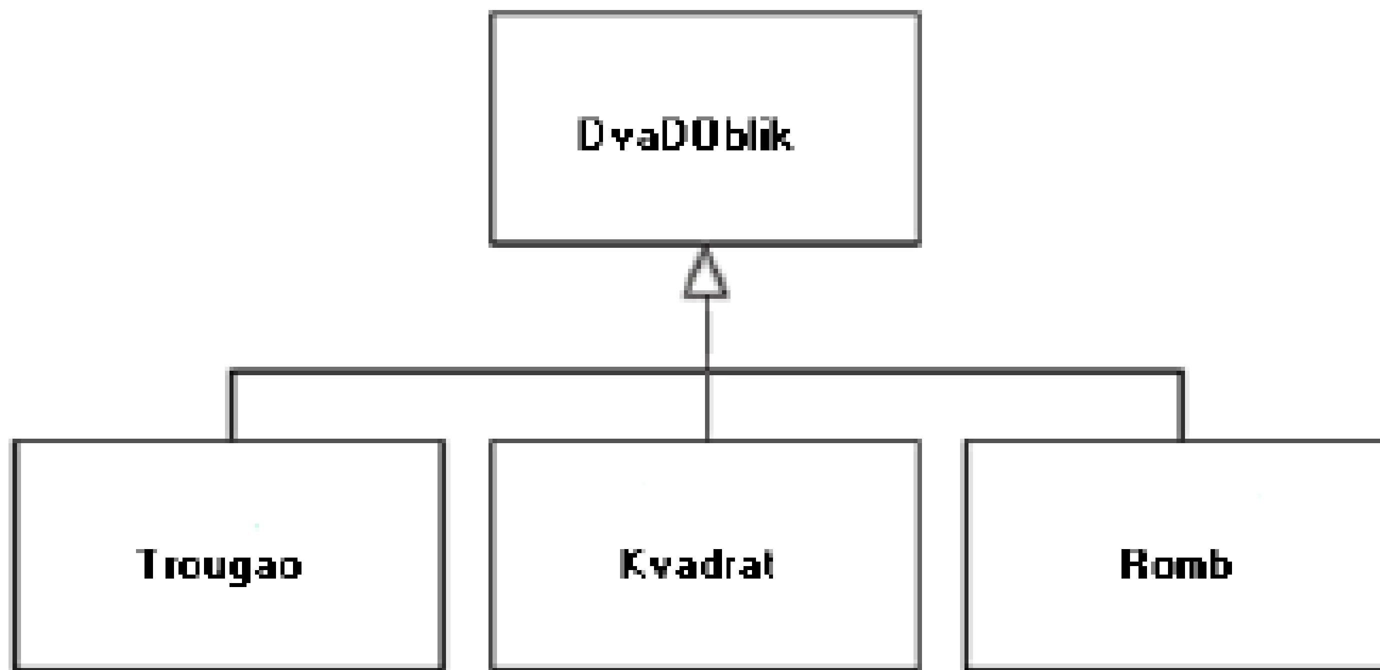
```
/ Postojeća klasa
 class Base
 {
 };
 // Izvedena klasa
 class Derived : public Base
 {
 };
```

Base Class

Derived Class



Dijagram



Primer

```
#include<iostream>
using namespace std;
class DvaDOblik {
public:
    double sirina;
    double visina;
    void prikaziDimenziju () {
      cout<<"Sirina i visina su:"
            <<sirina<< ", "
            <<visina <<endl;
    }
};
```

```
class Trougao : public DvaDOblik{
  public:
    using DvaDOblik::prikaziDimenziju;
    char stil [20];
    double povrsina() {
       return sirina * visina /2;
    }
};
int main()  {
    Trougao tr;
     tr.sirina=10;  tr.visina=20;
     cout<<"Povrsina trougla = "
             <<tr.povrsina()<<endl;
     tr.prikaziDimenziju();
     cin.get();
     return 0;
 }
```

Nasleđivanje (2)

- Osnovna klasa objedinjuje zajedničke atribute skupa objekata i iz nje se može izvesti proizvoljan broj specifičnih, izvedenih klasa
- Time se pojednostavljuje održavanje programa, a omogućava i njegovo ponovno korišćenje

Izvođenje klase

- Operator pristupa (access) u deklaraciji izvedene klase nije obavezan, ali ako postoji, mora biti :
  - public
  - protected ili
  - private
- Default vrednost specifikatora pristupa je private, ako se drugačije ne navede

Kontrola prava pristupa

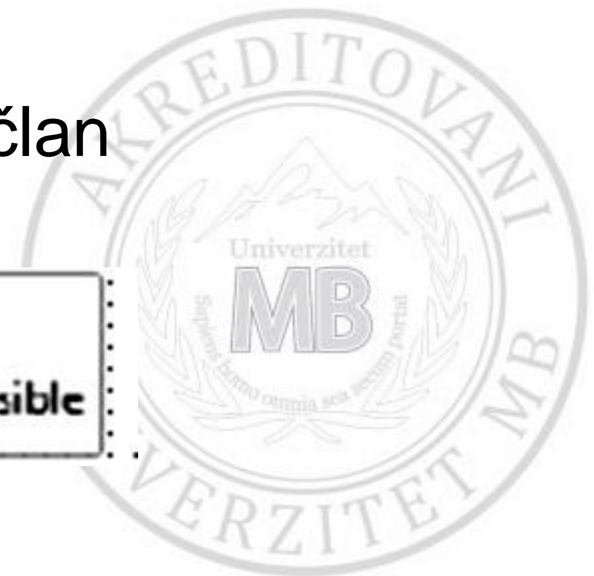
- Član klase može biti:
  - privatan (engl. private), dostupan samo funkcijama članicama iste klase
  - zaštićen (engl. protected), dostupan izvedenim klasama
  - javan (engl.public), dostupan svima
- Nasleđivanjem ne mogu da se naruše ograničenja pristupa članovima osnovne klase koja su već definisana

Kontrola prava pristupa (2)

```
int a=0;        //global
class A {
   int a;       //privatni član A::a
}
class B : public A {
   void f() {
     a=0;     // greška, član a je privatni član
   }

};
```

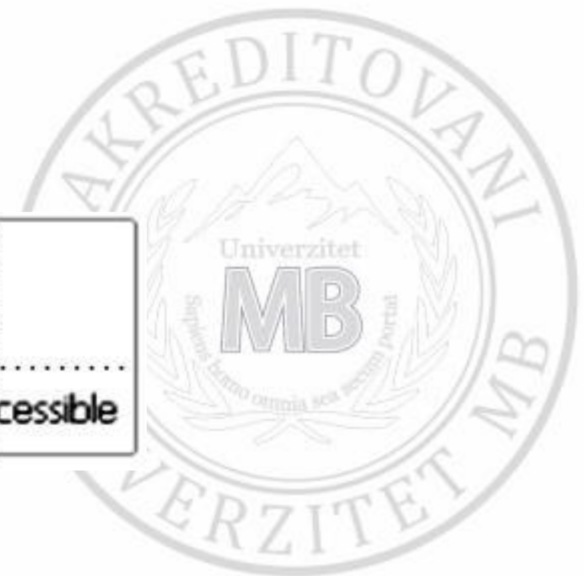
```
int A::a
Error: member "A::a" (declared at line 8) is inaccessible
```



Kontrola prava pristupa (3)

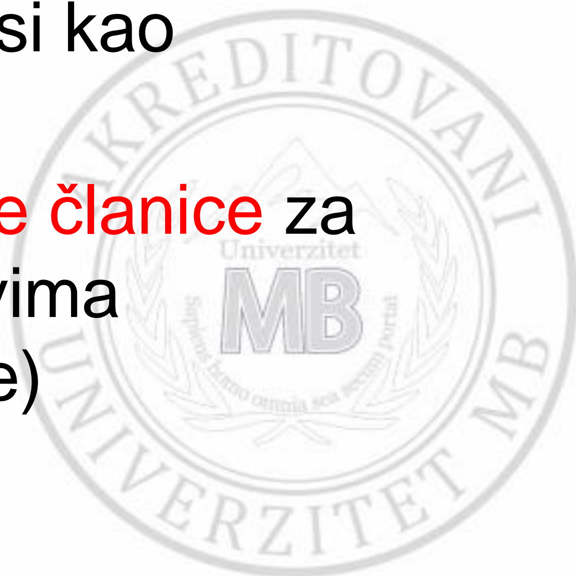
```
class X {
    void f(char);
  public:
    void f(int);
};
void g() {
  X x;
  x.f('x');
}
```

```
void X::f(char)
void X::f(int)
Error: function "X::f(char)" (declared at line 7) is inaccessible
```



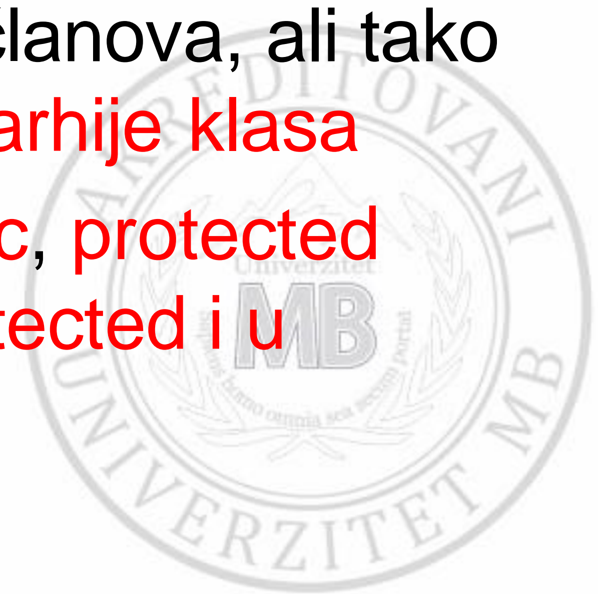
Kontrola prava pristupa (4)

- Da bi se izvedenoj klasi omogućio pristup privatnim članovima osnovne klase, postoje dva rešenja:
  - deklarisati članove u osnovnoj klasi kao **protected**
  - predvideti u osnovnoj klasi funkcije članice za pristup privatnim podacima članovima (akcesorske funkcije, tj. inspektore)



Korišćenje modifikatora protected

- Protected članu osnovne klase može da se pristupi iz izvedenih klasa
- Protected omogućuje izvođenje članova, ali tako da oni ostaju privatni unutar hijerarhije klasa
- Kada je klasa izvedena kao public, protected članovi osnovne klase ostaju protected i u izvedenoj klasi



Modifikator protected

```
#include<iostream>
using namespace std;
class Osnovna {
    int pa; //privatni član
  protected:
    int pb; //zaštićeni član
  public:
    int pc; //javni član
};
```

```
class Izvedena : public Osnovna{
public:
    void pisi (int x) {
      pb = pc = x;
      // može da pristupi javnom i
      // zaštićenom članu
      //pa =5;
      //ali ne i privatnom, greška!
    }
};
```

Modifikator protected (2)

```
int main () {
   Osnovna os;
   //os.pisi();   greška
   Izvedena iz;
   iz.pisi(3);
   cin.get();
   return 0;
}
```

Izvođenje

- Objekti izvedene klase uvek nasleđuju sve članove osnovne klase, bez obzira da li je izvođenje javno ili privatno
- U zavisnosti od načina izvođenja, pristup određenim nasleđenim članovima klase je na nekim mestima dozvoljen, a na nekim ne

Javno izvođenje

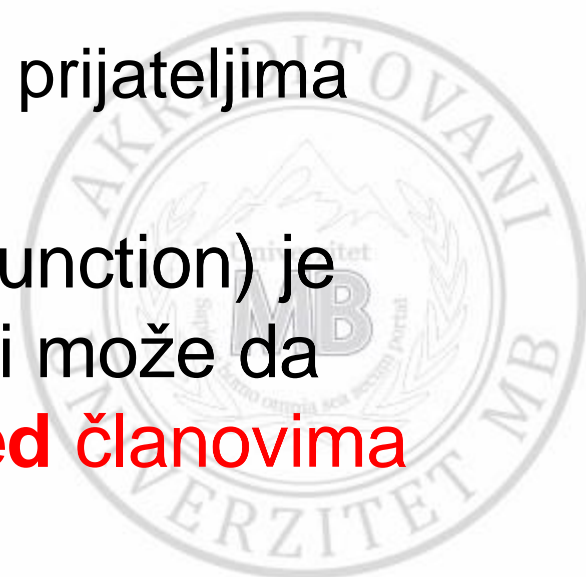
- Javni članovi osnovne klase jesu javni članovi i izvedene klase, pa korisnici izvedene klase mogu da “vide” sve osobine osnovne klase i u izvedenoj klasi

Privatno izvođenje

- Privatno izvođenje realizuje pojam sadržavanja (agregacije):
  – javni članovi osnovne klase su sakriveni u izvedenoj klasi, tj. objekti izvedene klase sadrže objekat osnovne klase u sebi
  – može se realizovati i članstvom

Definisanje funkcije prijatelja

- Ponekad je potrebno da se klasa projektuje tako da ima i "povlašćene" korisnike, odnosno funkcije ili druge klase koje imaju pravo pristupa do njenih privatnih i zaštićenih članova
  - takve funkcije i klase nazivaju se prijateljima (friends)
- Prijateljska funkcija (engl. friend function) je funkcija koja nije članica klase, ali može da pristupi njenim **private i protected** članovima



Definisanje funkcije prijatelja (2)

```
class aClass  {
private:
    int x;
    friend void OtherClass::fSet  (aClass &c, int a);
 };
class OtherClass  {
  public:
     void fSet(aClass &c, int a) {
           c.x = a;
     }
 };
```

Funkcije prijatelji

- Funkcije prijatelji se deklarišu unutar deklaracije klase, navođenjem specifikatora friend ispred deklaracije
- Ime funkcije prijatelji nije u oblasti važenja klase, pa se funkcija prijatelji ne poziva pomoću operatora pristupa članu (->), osim ako nije članica neke druge klase

Funkcije prijatelji (2)

```
#include <iostream>
using namespace std;
class C1;
class C2 {
public:
  void print(C1& x);
};
class C1{
  int a, b;
  friend void C2::print(C1& x);
public:
  C1() : a(1), b(2) { }
};
```

```
void C2::print(C1& x) {
  cout << "a = " << x.a<< endl;
  cout << "b = " << x.b << endl;
}

int main(){
  C1  xobj;
  C2 yobj;
  yobj.print(xobj);
  cin.get();
  return 0;
}
```

```
a = 1
b = 2
```

AKREDITOVANI UNIVERZITET MB

Funkcije prijatelji (3a)

```
#include <iostream>
using namespace std;
class Y {
   public:
      void h() {}
};
class X {
    friend void g(int, X&); //prijateljska funkcija nečlanica
    friend void Y::h();      //prijateljska funkcija članica klase Y
    int i;
  public:
      void f(int ip){
       i = ip;
    }
};
```

Funkcije prijatelji (3b)

```
void g (int k, X&x){
   x.i=k;   //prijateljska funkcija može da pristupa
}           //privatnim članicama klase

int main () {
   X x;
   x.f(5);  //pristup preko članice
   g(6, x);  //pristup preko prijatelja
   cin.get();
   return 0;
}
```

Prijatelj klase

- Prijatelj klase u C ++ može pristupiti "private" i „protected" članovima klase u kojoj je deklarisana kao prijatelj.
- Ako je potrebno da sve funkcije članice neke klase Y budu prijateljske funkcije klasi X, onda se klasa Y deklariše kao prijateljska klasa (engl. friend class) klase X
  - prijateljstvo se ne nasleđuje
  - prijateljstvo nije tranzitivna relacija

Prijatelj klase (2a)

```
#include <iostream>
using namespace std;
class MyClass {
   friend class SecondClass;
   public:
      MyClass() : Secret(0){}
      void printMember(){
         cout << Secret << endl;
      }
   private:
     int Secret;
};
```

Prijatelj klase (2b)

```
class SecondClass{
  public:
      void change(MyClass& yourclass,int x ){
            yourclass.Secret = x;
      }
};
void main(){
      MyClass my_class;
      SecondClass sec_class;
      my_class.printMember();
      sec_class.change( my_class, 5 );
      my_class.printMember();
}
```

Prijatelj klase (3)

```
#include <iostream>
using namespace std;
class X {
   int a;
   friend class Y;
};
class Y {
  public:
     int f() {
        X x;
        x.a=10;
       //OK iako je a privatni član
       return x.a;
     }
};
```

```
int main () {
    Y y;
     cout<<"y.f() ="<<y.f();
     cin.get();
     return 0;
}
```

```
y.f() =10
```

Prijatelj klase (4)

```
#include <iostream>
using namespace std;
class B {
   friend class A;
// klasa A je prijatelj klase B
   int i;
 public:
  int k;
   void set(int j){
  k=j;
   }
};
```

```
class A {
public:
A() {}
    A(B b) {
        b.i = 0;
        cout <<"b.i = "<<b.i <<endl;
    }
};
int main(){
    A a;
    B b;
    b.set(23);
    A a1 (b);
    cin.get();
    return 0;
}
```

b.i = 0

Konstruktori i destruktori

- Izvedene klase nasleđuju sve funkcije članice osnovne klase osim konstruktora, konstruktora kopije, destruktora i operatora dodele (=)
- Prijateljstvo se takođe ne nasleđuje
- Zbog toga se ove specijalne funkcije članice moraju definisati za svaku izvedenu klasu

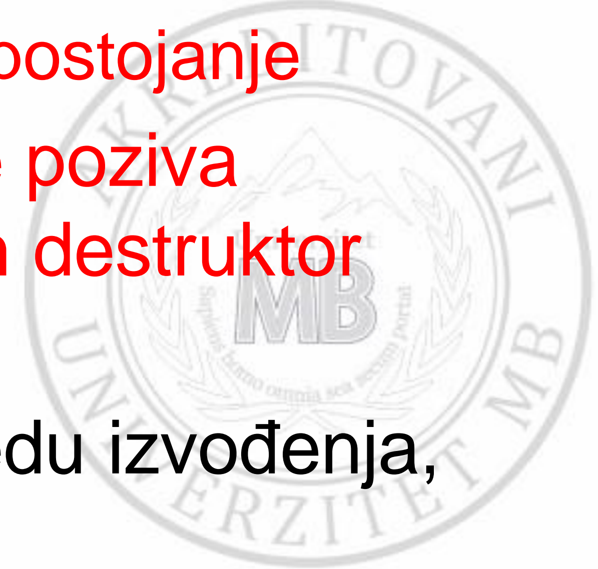
Konstruktori i destruktori (2)

- Kada osnovna klasa ima konstruktor, on se mora eksplicitno pozvati sintaksom

```
izvedeni_konstruktor (lista_argumenata) :
osnovni_konstruktor (lista_argumenata) {
    //telo izvedenog konstruktora
 }
derived (int x, int y, int z): base1 (y){
   //telo izvedenog konstruktora
}
```

Konstruktori i destruktori (3)

- Konstruktor osnovne klase prvo kreira osnovni podobjekat, pa zatim konstruktor izvedene klase kreira izvedeni deo objekta
  - osnovna klasa ne može da pristupi elementima u izvedenoj klasi, niti zna za njeno postojanje
- Kada se objekat uništava, prvo se poziva destruktor izvedene klase, a zatim destruktor osnovne klase
- Konstruktori se pozivaju u redosledu izvođenja, a destruktori u obrnutom smeru



Konstruktori i destruktori: Primer

```
#include <iostream>
using namespace std;
class X {
   public:
      X() {
        cout<<"Konstruktor klase X"
                <<endl;
      }
      ~X() {
        cout<<"Destruktor klase X"<<endl;
      }
};
```

Konstruktori i destruktori: Primer (2)

```
class Osnovna {
  public:
    Osnovna() {
      cout<<"Konstruktor klase Osnovna"<<endl;
    }
    ~Osnovna() {
      cout<<"Destruktor klase Osnovna"<<endl;
    }
};
```

Konstruktori i destruktori: Primer (3)

```
class Izvedena : public Osnovna {
   X x;
   public:
      Izvedena () {
        cout<<"Konstruktor klase Izvedena "<<endl;
      }
      ~Izvedena () {
        cout<<"Destruktor klase Izvedena "<<endl;
      }
};
```

Konstruktori i destruktori: Primer (4)

```
int main () {
  Izvedena i;
  i. ~Izvedena ();
  cin.get();
  return 0;
}
```

Redosled izvršavanja

```
// Student –osnovna klasa
// UnderGrad – izvedena klasa
// Imaju konstruktore i destruktore
int main()
{
   UnderGrad u1;
   ...
   return 0;
}// end main
```

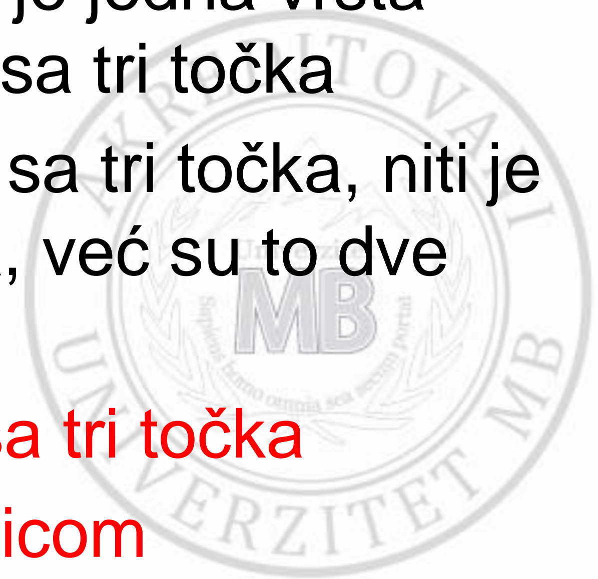
Izvršiti **Student** konstruktor, onda izvršiti **UnderGrad** konstruktor

Izvršiti **UnderGrad** destruktor, onda izvršiti **Student** destruktor

AKREDITOVANI UNIVERZITET MB

Višestruko nasleđivanje

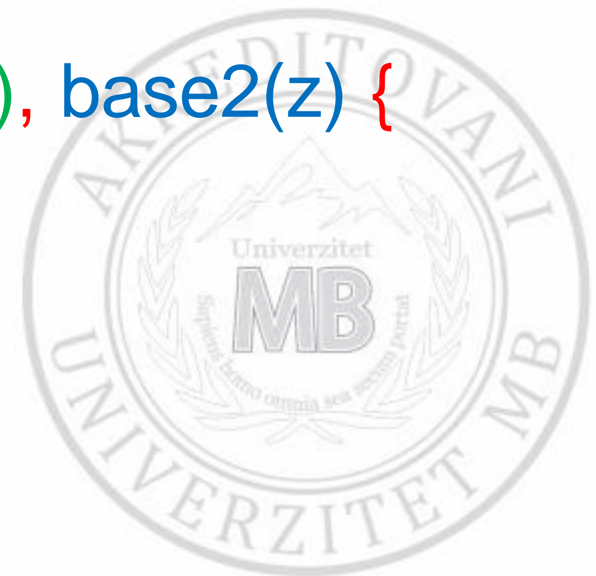
- Klasa može da nasledi nekoliko osnovnih klasa
- Tada svaki objekat izvedene klase nasleđuje sve članove svih svojih osnovnih klasa
  - na primer, motocikl sa prikolicom je jedna vrsta motocikla, ali i jedna vrsta vozila sa tri točka
  - pri tom, motocikl nije vrsta vozila sa tri točka, niti je vozilo sa tri točka vrsta motocikla, već su to dve različite klase
- Osnovne klase – motocikl i vozilo sa tri točka
- Izvedena klasa – motocikl sa prikolicom



Višestruko nasleđivanje (2)

- Izvedena klasa se višestruko izvodi tako što se u zaglavlju njene deklaracije navode imena osnovnih klasa razdvojena zarezima

```
derived (int x, int y, int z): base1(y), base2(z) {
   j=x;
   cout << "j= "<<j<<endl;
}
```



Višestruko nasleđivanje: Primer

```
#include <iostream>
using namespace std;
class B1 {
   public:
      void p1() {
         cout << "from B1" << endl;
      }
};
```

Višestruko nasleđivanje: Primer (2)

```
class B2 {
   public:
      void p2() {
      cout << "from B2" << endl;
   }
};
class A: public B1, public B2 {
   public:
      void p3() {
        cout << "from A" << endl;
      }
};
```

```
int main(){
   A a;
   a.p1();
   a.p2();
   a.p3();
   return 0;
}
```

Pokazivači na objekte izvedenih klasa

- Po pravilu, pokazivač na jedan tip ne može da pokazuje na podatak drugog tipa
- Pokazivači na objekte osnovne i izvedene klase su važan izuzetak od ovog pravila
  - u jeziku C++, pokazivač na objekat osnovne klase može se koristiti i kao pokazivač na objekat bilo koje klase koja je **izvedena** iz te osnovne klase
  - obrnuto ne važi: objektu osnovnog tipa ne može se pristupiti preko pokazivača na izvedeni tip
- Isto važi i za reference

Pokazivači na objekte izvedenih klasa (2)

- Svojstvo da se pokazivač na osnovnu klasu može koristiti i kao pokazivač na izvedenu klasu je fundamentalno svojstvo jezika C++ kojim se ostvaruje polimorfizam

```
Osnovna osnobj;
Izvedena izvobj;
Osnovna *po;
po=&osnobj; //po pokazuje  na objekat tipa Osnovna
po=&izvobj; //po pokazuje  na objekat tipa Izvedena
```

Polimorfizam

- Polimorfizam je svojstvo da svaki objekat izvedene klase izvršava metod tačno onako kako je to definisano u njegovoj izvedenoj klasi, čak i kada mu se pristupa kao objektu osnovne klase
  - osnovna klasa diktira interfejs koji će imati svi objekti izvedeni iz nje, ali ostavlja izvedenim klasama da definišu stvarni metod koji će implementirati taj interfejs
  - odatle fraza “jedan interfejs, više metoda” koja se često koristi za opisivanje polimorfizma

Polimorfizam (2)

- Polimorfizam omogućuje razdvajanje interfejsa i implementacije
  - korisno za razvoj biblioteka klasa
  - objektima izvedenim iz osnovne klase pristupaće se na isti način, mada će se operacije razlikovati od klase do klase
- Ako se za neku klasu predviđa izvođenje, a naročito ako ima virtuelne funkcije, argumente tipa te klase u sve funkcije treba prenositi preko referenci (ili pokazivača), jer se time obezbeđuje polimorfizam

Polimorfizam (3)

- Osnova za polimorfizam u jeziku C++ su:
  - nasleđivanje
  - pokazivači na osnovnu klasu
- Polimorfizam se realizuje pomoću virtuelnih funkcija

Virtuelne funkcije

- Virtuelna funkcija članica deklariše se u osnovnoj klasi navođenjem reči virtual, a nadjačava se (engl. override) u svim izvedenim klasama
  - svaka izvedena klasa ima sopstvenu verziju virtuelne funkcije
  - izvedena klasa može, ali i ne mora definisati svoju verziju virtuelne funkcije

Nadjačavanje virtuelne funkcije u izvedenoj klasi

- Nadjačavanje virtuelne funkcije u izvedenoj klasi nije specijalan oblik preklapanja funkcija (engl. overloading)!
  - za razliku od preklapanja, povratni tipovi, broj i tip argumenata virtuelne funkcije u osnovnoj i izvedenim klasama moraju da budu identični
  - ako se prototipovi ovih funkcija razlikuju, onda se ne radi o nadjačavanju već o preklapanju pa je virtuelna priroda funkcije izgubljena
- Virtuelna funkcija mora da bude članica klase, ili prijateljska funkcija druge klase

Polimorfna klasa

- Klasa koja sadrži virtuelnu funkciju zove se polimorfna klasa
  - kada se za poziv virtuelne funkcije koristi pokazivač na objekat osnovne klase, C++ određuje koju će (nadjačanu) virtuelnu funkciju pozvati na osnovu tipa objekta na koji pokazuje pokazivač, a ne na osnovu tipa pokazivača
  - ova odluka donosi se tokom izvršavanja programa i zato se zove dinamičko vezivanje (dynamic binding)

Polimorfizam: Primer

```
#include <iostream>
using namespace std;
class DvaDOblik {
protected:
    double  sirina;
  double   visina;
  public:
    virtual void koSi() {
      cout<<"2D oblik"<<endl;
    }
};
```

```
class Trougao: public DvaDOblik {
  public:
    virtual void koSi() {
      cout<<"Trougao"<<endl;
    }
};
class Krug: public DvaDOblik {
  double precnik;
  public:
    Krug() {
      precnik= sirina;
    }
    virtual void koSi() {
      cout<<"Krug"<<endl;
    }
};
```

Polimorfizam: Primer (2)

```
int main () {
      DvaDOblik dvaDoblik;
      Krug  krug;
      Trougao trougao;
      DvaDOblik *poblik;
      poblik=&dvaDoblik;
      poblik->koSi();
      poblik = &krug;
      poblik->koSi();
      poblik = &trougao;
      poblik->koSi();
      cin.get();
      return 0;
}
```

```
2D oblik
Krug
Trougao
```

Prenos parametara do osnovne klase I

```
#include <iostream>
using namespace std;
class A{
   public:
     int x, y;
     A(int a, int b) {
       x=a;
      y=b;
     }
     ~A();
     cout<<"x= "<< x <<endl;
     cout<<"y= "<< y <<endl;
}
```

```
class B : public A{
public:
public:
    B(int a, int b) : A(a, b)    {
    }
    ~B();
};
int main(){
    A* aobj =new A(44,55);
    B* bobj = new B(5,6);
    cin.get();
    return 0;
}
```

```
x= 44
y= 55
x= 5
y= 6
```

Prenos parametara do osnovne klase II

```
#include <iostream>
using namespace std;
class base1 {
protected:
  int i;
public:
  base1(int x) {
     i=x;
     cout <<"i = "<<i<<endl;
  }
  ~base1() {
     cout << "U destruktoru base1\n";
  }
};
```

Prenos parametara do osnovne klase III

```
class base2 {
protected:
int k;
public:
  base2(int x) {
     k=x;
  cout <<"k = "<<k<<endl;
  }
  ~base2() {
     cout << "U destruktoru base2\n";
  }
};
```

Prenos parametara do osnovne klase IV

```
class derived: public base1, public base2 {
  int j;
public:
  derived (int x, int y, int z): base1(y), base2(z) {
     j=x;
     cout << "j= "<<j<<endl;
  }
  ~derived() {
     cout <<"U destruktoru derived\n";
  }
  void show() {
     cout << i << " " << j << " " << k << "\n";
  }
};
```

Prenos parametara do osnovne klase V

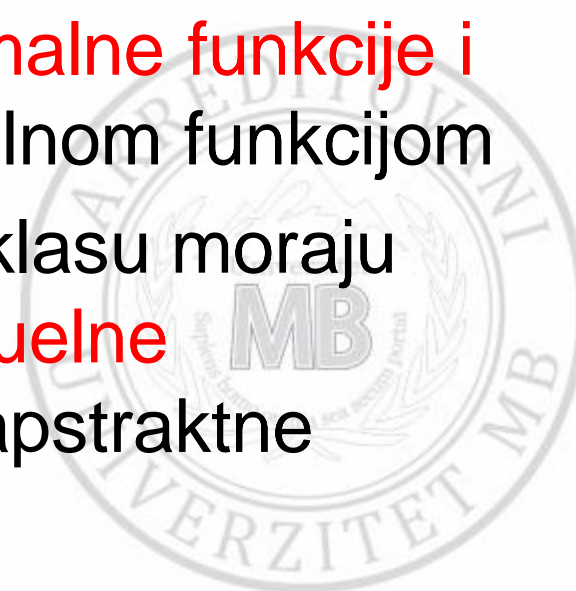
```
int main() {
  derived ob(3, 4, 5);
  ob.show();
  cin.get();
  return 0;
}
```

Apstraktne klase

- "Čista" virtuelna funkcija je ona koja nema definiciju u osnovnoj klasi, već je izvedene klase definišu. Počinju sa ključnom reči virtual i završavaju sa = 0.
- Deklariše se u osnovnoj klasi na sledeći način: **virtual tip ime_funkcije(lista_argumenata) = 0;**
- Klasa koja sadrži barem jednu "čistu" virtuelnu funkciju zove se apstraktna klasa
- Apstraktna klasa ne može imati objekte
  - ali mogu da postoje pokazivači na tip apstraktne klase, čime se ostvaruje polimorfizam

Apstraktne klase (2)

- Apstraktna klasa ne može da ima objekte, ali se mogu kreirati pokazivači i reference apstraktne klase
- Apstraktna klasa može imati normalne funkcije i varijable zajedno sa čistom virtuelnom funkcijom
- Klase koje nasleđuju apstraktnu klasu moraju da implementiraju samo čiste virtuelne funkcije, ili će oni postati takođe apstraktne klase



Čista virtuelna funkcija

```
#include <iostream>
using namespace std;
class Base {
//Apstraktna osnovna klasa
  public:
    //čista virtualna funkcija
    virtual void show() = 0;
};
class Derived: public Base{
 public:
  void show() {
      cout << "Implementacija
           virtualne funkcije u
           izvedenoj klasi";
  }
};
```

```
int main(){
    Base *b;
    Derived d;
    b = &d;
    b->show();
    cin.get();
    return 0;

}
```

Rezultat
Implementacija virtualne funkcije u izvedenoj klasi

Virtuelni destruktor

- Konstruktor je funkcija koja od “obične gomile u memoriji” pravi “živi” objekat
  - poziva se pre nego što je objekat kreiran, pa nema smisla da bude virtuelan, što C++ ni ne dozvoljava
  - kada se objekat kreira, njegov tip je uvek poznat, pa je određen i konstruktor koji se poziva
- Destruktor pretvara “živi” objekat u “hrpu bitova” u memoriji
- Destruktor može biti virtuelan

Virtuelni destruktor (2)

- Virtuelni mehanizam, tj. dinamičko vezivanje tačno određuje koji destruktor (osnovne ili izvedene klase) će biti prvo pozvan kada se objektu pristupa posredno, preko pokazivača
- Ako neka osnovna klasa ima neku virtuelnu funkciju, onda i njen destruktor treba da bude virtuelan

Virtuelni destruktor: Primer

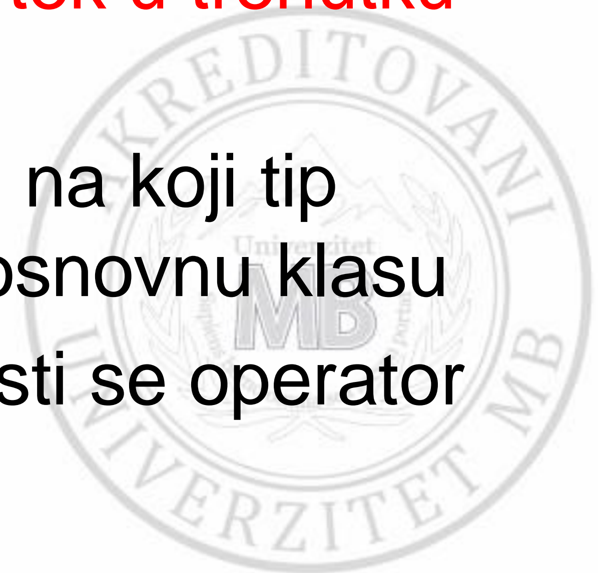
```
#include <iostream>
using namespace std;
class Osnovna {
public:
     virtual ~Osnovna() {
         cout<<"Virtuelni destruktor
                 osnovne" <<endl;
     }
};
class Izvedena : public Osnovna {
  public:
     virtual ~Izvedena() {
         cout<<"Virtuelni destruktor
                     izvedene"<<endl;
     }
};
```

```
void release (Osnovna *po) {
//poziv virtuelnog destruktora
     delete po;
}
int main () {
   Osnovna *pb = new Osnovna;
   Izvedena *pi = new Izvedena;
   //poziva se ~Osnovna
   release (pb);
  //poziva se ~Izvedena
   release (pi);
   cin.get();
   return 0;
  }
```

```
Virtuelni destruktor osnovne
Virtuelni destruktor izvedene
```

RTTI (engl. Runtime Type Identification)

- U polimorfnim jezicima kakav je C++ postoje slučajevi kada je tip objekta nepoznat tokom kompajliranja jer postaje poznat tek u trenutku izvršavanja
- Nije uvek moguće unapred znati na koji tip objekta pokazuje pokazivač na osnovnu klasu
- Da bi se saznao tip objekta, koristi se operator *typeid*



Operator typeid

- Operator type def se uključuje zaglavljem <typeinfo> i najčešće se koristi u obliku typeid (objekat)
- Operator typeid vraća referencu na objekat tipa type_info koji opisuje tip objekta

Operator typeid (2)

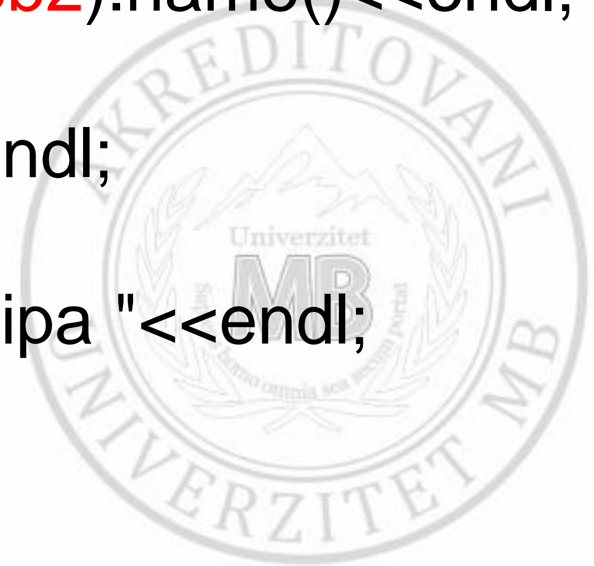
- Najvažnija upotreba operatora typeid je kada se primenjuje preko pokazivača na polimorfnu osnovnu klasu (tj. na klasu koja ima barem jednu virtuelnu funkciju)
- U tom slučaju typeid tokom izvršavanja može da prepozna stvarni tip objekta na koji pokazuje pokazivač na osnovnu klasu
- Ako se typeid primenjuje na pokazivač nepolimorfne klase, vraća pokazivač na osnovnu klasu

Operator typeid: Primer #1

```
#include <iostream>
using namespace std;
class mojaKlasa1 {};
class mojaKlasa2 : public mojaKlasa1{};
int main () {
  int i,j;
  float f;
  char *p;
  mojaKlasa1 ob1;
  mojaKlasa2 ob2;
```

Operator typeid: Primer #1 (2)

```
cout<<"Tip varijable i je: "<<typeid(i).name()<<endl;
cout<<"Tip varijable f je: "<<typeid(f).name()<<endl;
cout<<"Tip varijable p je: "<<typeid(p).name()<<endl;
cout<<"Tip varijable ob1 je: "<<typeid(ob1).name()<<endl;
cout<<"Tip varijable ob2 je: "<<typeid(ob2).name()<<endl;
if (typeid(i) == typeid (j))
   cout<<"Variable i i j su istog tipa "<<endl;
if (typeid(ob1) != typeid (ob2))
   cout<<"Variable ob1 i ob2 nisu istog tipa "<<endl;
cin.get();
return 0;
}
```



Operator typeid: Primer #1 (3)

Tip varijable i je: int
Tip varijable f je: float
Tip varijable p je: char *
Tip varijable ob1 je: class mojaKlasa1
Tip varijable ob2 je: class mojaKlasa2
Variable i i j su istog tipa
Variable ob1 i ob2 nisu istog tipa

Operator typeid: Primer #2

```
#include <iostream>
using namespace std;
class Sisar {
   public:
      virtual bool leze_jaja() {
         return false;
      }
};
class Macka : public Sisar{
};
class Kljunar :  public Sisar{
   bool leze_jaja() {
        return true;
   }
};
```

Operator typeid: Primer #2 (2)

```
int main () {
  Sisar *p, biloKojiSisar;
  Macka macka;
  Kljunar kljunar;
  p = &biloKojiSisar;
  cout<<"p pokazuje na objekat tipa : "<<typeid(*p).name()<<endl;
  p = &macka;
  cout<<"p pokazuje na objekat tipa : "<<typeid(*p).name()<<endl;
  p = &kljunar;
  cout<<"p pokazuje na objekat tipa : "<<typeid(*p).name()<<endl;
  cin.get();
  return 0;
}
```

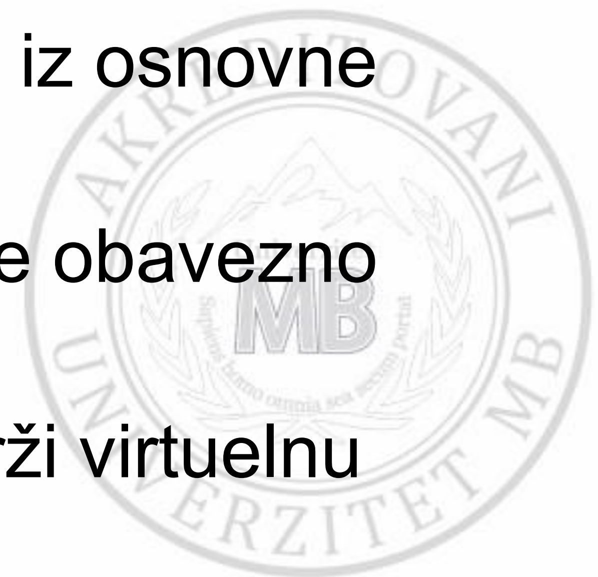
```
p pokazuje na objekat tipa : class Sisar
p pokazuje na objekat tipa : class Macka
p pokazuje na objekat tipa : class Kljunar
```

Zaključak

- Osnovna klasa – izvedene klase
- class izvedenaKlasa : specifikatorIzvođenja osnovnaKlasa { tijelo izvedene klase }
- Specifikatori pristupa – public, private, protected
- Inspektori – funkcije koje samo čitaju vrednosti podataka članova
- Mutatori – funkcije koje menjaju vrednosti podataka članova

Zaključak (2)

- Konstruktori i destruktori
- Višestruko nasleđivanje – kada klasa nasleđuje više osnovnih klasa
- Nadjačavanje funkcija – kada izvedena klasa definiše sopstvenu verziju funkcije iz osnovne klase
- Virtuelna funkcija – funkcija koja će obavezno biti nadjačana u izvedenoj klasi
- Polimorfna klasa – klasa koja sadrži virtuelnu funkciju



Kraj prezentacije

HVALA NA PAŽNJI!