**Sveučilište u Zagrebu**
**PMF – Matematički odjel**

# Objektno programiranje (C++)

Vježbe 06 – Nasljeđivanje

**Vinko Petričević**

# Nasljeđivanje

- Objektno orijentirano programiranje proširuje objektno-temeljeno mogućnošću uvođenja odnosa između tipova i njihovih podtipova
  - to se ostvaruje naslijeđivanjem (izvođenjem klasa)
    - bazna klasa (nadklasa)
    - izvedena klasa (potklasa)
  - hijerarhija naslijeđivanja reprezentira odnos između baznih i izvedenih klasa

- Primjer: Imamo tip koji predstavlja osobu:
  - ```
    class Covjek {
        string m_ime;
    public:
        Covjek(char* ime) : m_ime(n) {}
        void print() const {
            cout << m_ime << endl;
        }
    };
    ```
- Definirajmo i tip koji predstavlja radnika:
  - ```
    class Radnik {
        string m_ime;
        int m_placa;
    public:
        // ... sve analogno kao u klasi Covjek
    };
    ```

# Nasljeđivanje klasa

- Uočimo kako ovakva definicija ima bitan konceptualni nedostatak:

  `Radnik` sintaktički nije `Covjek`

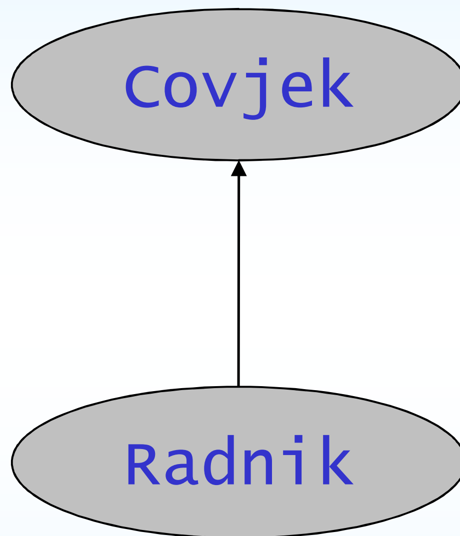
  -> npr. ne možemo napraviti:

```
list<Covjek> l;
l.push_back(Radnik(“Pcela"));
```

- Ispravan pristup treba uzeti u obzir da je `Radnik` također i `Covjek` -> koristimo nasljeđivanje klasa:

```
• class Radnik : public Covjek {
      int m_placa;
      // ...
  };
```

# Nasljeđivanje klasa




- Klasa `Radnik` je izvedena iz klase `Covjek`
  - `Covjek` – bazna klasa (nadklasa)
  - `Radnik` – izvedena klasa (potklasa)
- Izvedena klasa ima sve što ima i bazna klasa; plus još neke svoje specifičnosti
- Radnik JEST (IS A) Covjek
- Radnik IMA (HAS A) placa...

# Nasljeđivanje klasa

- Izvođenjem klase `Radnik` iz klase `Covjek`, `Radnik` postaje podtip od `Covjek` (Radnik je specijalna vrsta Covjeka)
  - instancu klase `Radnik` moguće je koristiti svugdje gdje je moguće i instancu klase `Covjek`
  - ```
    Radnik r;
    Covjek c;
    ```
  - ```
    list<Covjek> l;
    l.push_front(r);
    l.push_front(c);
    ```
- Primjer:
  - ```
    Covjek* pc = &r; // ok, Radnik ima "višak" podataka
    Radnik* pr = &c; // greska, Covjek-u "nedostaju" neki podaci
    pc->m_placa = 2000; // greska, Covjek nema placu
    pr = static_cast<Radnik*>(pc); // brute force 😐
    pr->m_placa = 2000; // ok sintaksno
    ```
    ```

# Što se tu stvarno događa u memoriji?

- Primjer:
  - ```
    class A {
          int x;
    };
    ```
  - ```
    class B : public A {
          int y;
    };
    ```
- Što će ispisati slijedeća linija?
  - `std::cout << sizeof(a) << " " << sizeof(b) << "\n";`
- Koja od slijedećih inicijalizacija su ispravne?
  - `A* pa = &b;`
  - `B* pb = &a;`
  - `A& ra = b;`
  - `B& rb = a;`

# Kvalifikatori pristupa

- Primjer:
  - ```
    class Covjek {
        string m_ime;
        // ...
    public:
        void print() const;
        string get_ime() const
        { return m_ime; }
    };
    ```
  - ```
    class Radnik : public Covjek {
        // ...
    public:
        void print() const;
    };
    ```

# Konstruktori u izvedenoj klasi

```
• class Radnik : public Covjek {
        ine m_placa;
  public:
        Radnik (char *ime, int placa) :
              Covjek(ime),
              m_placa(placa)
        { }
  };
```

poziv konstruktora bazne klase;
ne ide drugačije jer su članovi od Covjek private, pa im ne možemo pristupiti direktno iz izvedene klase;
ako ne spomenemo Covjek na ovom mjestu, automatski se poziva default konstruktor od Covjek-a;

# Kvalifikatori pristupa

- U izvedenoj klasi možemo koristiti `public` i `protected` članove bazne klase
  - ```
    void Radnik::print() const {
        cout << "ime: " << get_name() << endl;
    }
    ```
- Ali nemamo pristup njenim privatnim dijelovima
  - ```
    void Radnik::print() const {
        // greska
        cout << "ime: " << m_ime << '\n';
    }
    ```
- Protected dijelove klase
  - ```
    class Covjek {
    protected:
        string m_ime;
    ```

  možemo koristiti u njezinoj potklasi:

  - ```
    void Radnik::print() const {
        cout << m_ime << " "<< m_placa;
        // Covjek::print(); cout << m_placa << endl;
    }
    ```

# Načini izvođenja i kvalifikatori pristupa

| način izvođenja / kvalifikatori u nadklasi | public | private | protected |
|---|---|---|---|
| public | public | private | protected |
| private | private | private | private |
| protected | protected | private | protected |

# Konstruktori i destruktori

- Bitna napomena:
  - Instance klasa konstruiraju se "bottom up":
    1. bazna klasa
    2. članovi izvedene klase
    3. sama izvedena klasa
  - Destrukcija se odvija obrnutim redoslijedom:
    1. sama izvedena klasa
    2. članovi izvedene klase
    3. bazna klasa
  - Članovi klase i njene bazne klase konstruiraju se onim redoslijedom kojim su deklarirani, a destruiraju obrnutim redoslijedom

# Kopiranje

- Primjer:
  ```
  class Covjek {
      // ...
      Covjek& operator=(const Covjek&);
      Covjek(const Covjek&);
  };
  ```
- Što će se dogoditi prilikom slijedećih poziva copy konstruktora, odnosno operatora pridruživanja?
  ```
  void f(const Radnik& m) {
      Covjek e = m;
      e = m;
  }
  ```
- Pozivaju se odgovarajuće funkcije iz bazne klase

# Kopiranje

- Primjer:
  - ```
    class Covjek {
        string ime; ...
        Covjek& operator=(const Covjek&);
        Covjek(const Covjek&);
    };
    ```
  - ```
    class Radnik : public Covjek {
        int placa;
    };
    ```
- Što će se dogoditi prilikom slijedećih poziva copy konstruktora, odnosno operatora pridruživanja?
  - ```
    void f(Radnik r) {
        Radnik s;
        s = r;
    }
    ```
    ```

# Pointeri na funkcije

- Pitanje: Pretpostavimo da imamo slijedeću funkciju:
  - ```
    int lex_cmp(const string& s1,
                const string& s2) {
        return s1.compare(s2);
    }
    ```

  Kojeg je tipa `lex_cmp`?
- Ukoliko deklariramo `pf` ovako:
  - ```
    int (*pf)(const string &, const string &);
    ```

  `pf` je pointer na funkciju istog tipa kao i `lex_cmp`.
  - `pf` može biti inicijaliziran adresom bilo koje funkcije koja je tog tipa
    - ```
      int size_cmp(const string &, const string &);
      pf = size_cmp;
      ```
  - Kao što je ime polja adresa, tako je i ime funkcije adresa
      ```

# Pointeri na funkcije

- Neka je dana funkcija:
  - ```
    int min(int* a, int sz) {
        int minVal = a[0];
        for (int i = 1; i < sz; ++i)
            if (minVal > a[i])
                minVal = a[i];
        return minVal;
    }
    ```
- Pointer na funkciju
  - `int (*pf)(int*, int);`

  možemo inicijalizirati na dva ekvivalentna načina:
  - `pf = min;` ili `pf = &min;`
- Invokacija funkcije preko pointera:
  - ```
    int a[aSize] = { 7, 4, 9, 2, 5 };
    pf(a, aSize); ili (*pf)(a, aSize);
    ```

# Pointeri na funkcije

- Zadatak: Napišite funkciju `sort()` koja sortira polje `int`-ova. Parametri funkcije trebaju biti pointer na prvi i posljednji element polja:

```
void sort(int* first, int* last);
```

  - Riješite zadatak na dva načina: u terminima pokazivača i u terminima polja (računajući size=first-last)
- Zadatak: Prepravite vašu implementaciju tako da definirate funkciju koja vrši uspoređivanje elemenata tipa `int`, te potom `sort()` parametrizirate odgovarajućim pointerom na funkciju tog tipa.

# Pointeri na funkcije

- Rješenje:
  - bool less(int x, int y) { return x < y; }
  - typedef bool (*PF_cmp)(int, int);
  - void sort(int* first, int* last, PF_cmp compare = less) {

```
    if (first < last) {
        int elem = *first;
        int* low = first;
        int* high = last + 1;
        for (;;) {
            while (compare(*++low, elem) && low < last);
            while (compare(elem, *--high) && high > first);
            if (low < high)
                swap(*low, *high);
            else break;
        }
        swap(*first, *high);
        sort(first, high - 1, compare);
        sort(high + 1, last, compare);
    }
} // qsort
```

# Virtualne funkcije

```
class Covjek {
protected:
   string m_ime;
public:
   Covjek(const char*ime) : m_ime(ime) { }
   void print() {
        cout << name << endl;
   }
};

class Radnik : public Covjek {
   int m_placa;
public:
   void print() {  // novi print() koji sakrije definiciju starog printa
        cout << m_ime << " "<< m_placa << endl;
   }
};
```

# Virtualne funkcije

- Primjer:
  - Covjek c1(“Sure");
    Covjek c2(“Luja");
    Radnik r(“Cajper");
  - list<Covjek*> pl;
    pl.push_back(&c1);
    pl.push_back(&c2);
    pl.push_back(&r); // radnik!
  - for (list<Covjek*>::iterator it = pl.begin(); it != pl.end(); ++it)
        (*it)->print();
- Što se dogodilo? Koji print() se poziva kod Radnika koji je spremljen kao Covjek?
- print() od Covjek-a, bez obzira koji se objekt ustvari nalazi ispod

# Virtualne funkcije

- Koncept virtualnih funkcija omogućava deklaraciju funkcija u baznoj klasi koje mogu biti redefinirane u svakoj izvedenoj klasi
- Primjer:
  - ```
    class Covjek {
        // ...
    public:
        virtual void print();
        // ...
    };
    ```
  - Ključna riječ `virtual` ukazuje da `void print()` predstavlja jedinstveno sučelje za funkciju `print()` definiranu u klasi `Covjek` i svim izvedenim klasama
- Metode u većini OO jezika (npr. Java, C#) su zapravo virtualne funkcije
    ```

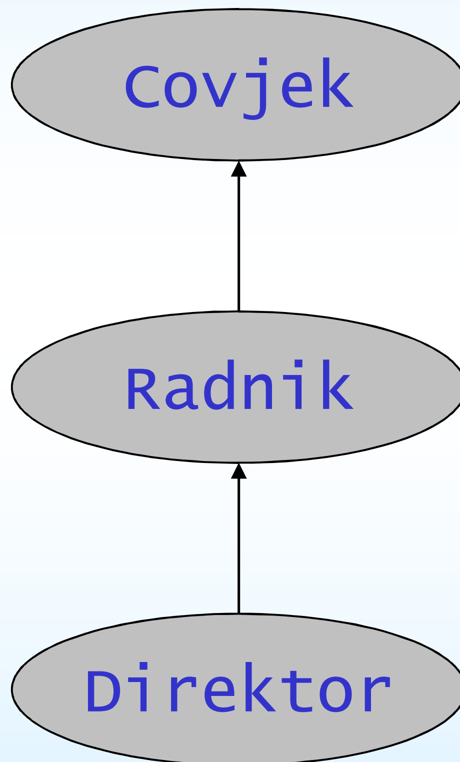
# Virtualne funkcije

- Virtualna funkcija mora biti definirana u klasi u kojoj je prvi puta deklarirana (osim ako je deklarirana kao tzv. čista virtualna funkcija)
  - ```
    void Covjek::print() {
        cout << m_ime << endl;
    }
    ```
- U izvedenoj klasi može, ali i ne mora biti definirana
  - ```
    class Radnik : public Covjek {
        // ...
    public:
        void print();
        // ...
    };
    void Radnik::print() {
        Covjek::print();
        cout << m_placa << endl;
    }
    ```

# Virtualne funkcije

- Za funkciju u izvedenoj klasi koja ima isto ime i tipove ulaznih parametara kažemo da redefinira (eng. overrides) virtualnu funkciju iz bazne klase
- Tipovi koji sadrže virtualne funkcije nazivaju se polimorfni tipovi
- Da bi se u C++u omogućilo polimorfno ponašanje, članske funkcije koje se pozivaju moraju biti virtualne, a objekti se moraju manipulirati putem pointera ili referenci
  - ukoliko se objekt manipulira direktno, definicija tipa objekta mora biti poznata prilikom prevođenja, pa se u tom slučaju ne može ostvariti run-time polimorfizam
- Pozivanje funkcije pomoću operatora dosega (kao npr. u funkciji `Covjek::print()`) osigurava da se tom prilikom funkcija poziva statički

# Hijerarhija klasa




- Primjer:
  - class Covjek
    { /* ... */ };
  - class Radnik : public Covjek
    { /* ... */ };
  - class Direktor : public Radnik
    { /* ... */ };

# Zadatak

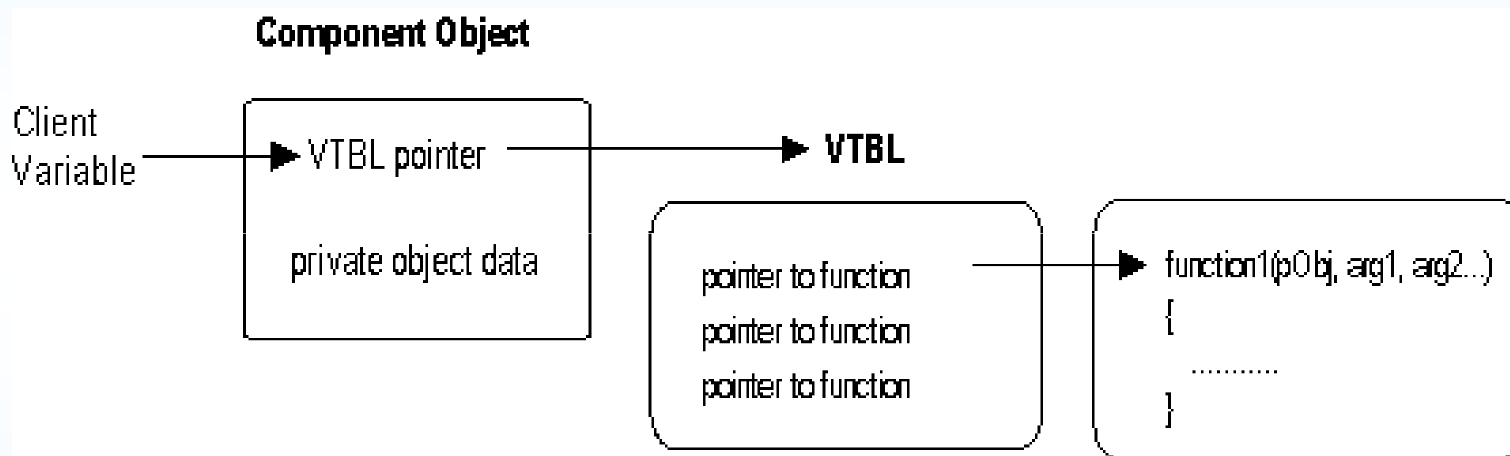
- Napravite hijerarhiju klasa kao na prethodnoj slici i definirajte print() tako da ispisuje sve podatke.
- Ubacite u listu <Covjek*> razne objekte i pozovite print na njima. Ovo isprobajte sa i bez virtual.
- Napravite listu <Covjek> (bez pokazivača) i isprobajte istu stvar.

# Virtualne funkcije

- Unutar implementacije funkcija klase, naredbe se izvršavaju virtualno, ako čovjeku dodamo naredbu ispis()...
  - ```
    void Covjek::ispis() {
        cout << "Zovem se: ";
        print();
    }
    void f(Covjek& c) {
        c.ispis();
    }
    ```
  - ```
    Radnik r("rrr", 2000);
    r.ispis();
    f(r);
    ```

# Kako su implementirane virtualne funkcije u C++u?

- Svaka polimorfna klasa ima (statičku) tablicu pointera na virtualne funkcije (VTBL), i (nestatički) pokazivač na tu tablicu (VTBL pointer)




- Pridruživanje pokazivača na tu tablicu se događa prije izvršavanja kôda pojedinog konstruktora (a i destruktora!)
- neoprezno korištenje virtualnih funkcija za vrijeme izvršavanja konstruktora može izazvati greške

# Virtualni destruktor

- bilo bi dobro da destruktori uvijek budu virtualni, jer ako netko dinamički kreira objekt, postoji mogućnost da se u protivnom uništi samo bazni dio klase na destruktoru

# Apstraktne klase

- U nekim slučajevima klasa predstavlja apstraktan koncept čije instance ne mogu postojati
  - npr. klasa `Shape` ima smisla postojati jedino kao bazna klasa iz koje će se dalje izvoditi druge klase
- Kako sintaktički riješiti slijedeći problem?
  - ```
    class Shape {
    public:
        virtual void move(int, int) { error("Shape::move") ; }
        virtual void draw() { error("Shape::draw") ; }
        // ...
    };
    ```
  - `Shape s; // instanca klase Shape nema previse smisla`
    ```

# Apstraktne klase

- Deklarirat ćemo virtualne funkcije u klasi `Shape` kao čiste virtualne funkcije (eng. pure virtual functions)

```
class Shape { // Shape je apstraktna klasa
public:
        virtual void move(int, int) = 0;
        virtual void draw() = 0;
};
```

- Klasa s jednom ili više čistih virtualnih funkcija naziva se apstraktna klasa
- Nije moguće kreirati instance apstraktnih klasa
  - Shape s; // greska: instanca apstraktne klase Shape
- Sučelje (eng. interface) ≈ apstraktna klasa koja sadrži samo čiste virtualne funkcije

# Sučelja

C++

```
class Shape {
public:
    virtual void move(int, int) = 0;
    virtual void draw() = 0;
};
```

C#

```
interface Shape {
    void move(int, int);
    void draw();
}
```

Java

```
interface Shape {
    public void move(int, int);
    public void draw();
}
```

- U C++u sučelja i općenito apstraktne klase nasljeđuju se na isti način kao i ne-apstraktne klase
  - ```
    class Point { /* ... */ };

    class Circle : public Shape {
    public:
        void move(int, int) { } // redefinira
                                // Shape::move
        void draw() { } // redefinira
                        // Shape::draw
        Circle(Point p, int r);
    private:
        Point center;
        int radius;
    };
    ```
    ```

- Čista virtualna funkcija koja nije definirana u izvedenoj klasi ostaje takvom, pa je u tom slučaju izvedena klasa također apstraktna
  - ```
    class Polygon : public Shape { // apstraktna klasa
    public:
       // ...
    };
    ```
  - ```
    Polygon b; // greska: instanca apstraktne klase Polygon
    ```
  - ```
    class Irregular_polygon : public Polygon {
       list<Point> lp;
    public:
       void draw() ; // redefinira Shape::draw
       void move(int, int) ; // redefinira Shape::move
       // ...
    };
    ```
  - ```
    Irregular_polygon p(tocke) ; // ok
    ```

# Zadaci za vježbu

- Zadatak: Definirajte klase `GeometrijskoTijelo`, `Valjak`, `Kugla`, `Poliedar`, `Kvadar`, te odgovarajućim dijagramom naznačite odnose među njima (tj. nacrtajte hijerarhiju klasa).
  U svakoj od klasa neka je definirana virtualna funkcija `double volumen()`.
  Napišite funkciju koja za dobivenu listu objekata računa njihov ukupni volumen.

# Osmislite hijerarhiju klasa za rad s prozorima

- Klasa Prozor ima metodu PosaljiPoruku(int x, int y, string p) Prozor zauzima prostor na ekranu (x,y,w,h)
- Klasa aplikacija sadrži listu *glavnih* prozora. Svaki novododani prozor je gornji. Imamo naredbe dodajProzor, te PosaljiPoruku koja proslijedi poruku odgovarajucem prozoru.
- Iz Prozora izvedite Klasu gumb koja ima ime i na svaku primljenu poruku ispisuje svoje ime i tekst poruke
- Iz Prozora izvedite Klasu Dialog koji ima 2 gumba, OK i Cancel (koje aplikacija ne sadrži). Ako je kliknuto na njih, neka oni izvrše poruku, ako ne, neka je izvrši dialog.

# Zadaci za vježbu

- `typeid()` operator vraća `type_info` tip (točnije const referencu)
- `type_info` tip ima `.name()` metodu koja vraća ime tipa
- `type_info` tip se ne može kopirati

- Zadatak: Napišite funkciju koja za dobivenu listu objekata tipa Covjek* računa koliko ima direktora.
- Zadatak: Napišite funkciju koja za dobivenu listu objekata tipa Covjek* računa koliko ima studenata.
- Zadatak: Napišite funkciju koja za dobivenu listu objekata tipa Covjek* računa koliko ima radnika koji još nisu direktori.

- Zadatak: Definirajte klase koje reprezentiraju slijedeće tipove vozila: automobil, avion, bicikl, brod, kamion, tegljač (uvedite po potrebi apstraktne klase na odgovarajuća mjesta u hijerarhiji), te odgovarajućim dijagramom naznačite odnose među njima (tj. nacrtajte hijerarhiju klasa). U svakoj od klasa neka je definirana virtualna funkcija `double nosivost()`.
  Napišite funkciju koja za dobivenu listu objekata računa njihovu ukupnu nosivost.
- Napišite funkciju koja, za dobiveni listu objekata i tip traženog objekta, računa njihovu prosječnu nosivost.
- Napišite funkciju koja, za dobiveni listu objekata vraća tip najveće prosječne nosivosti.

# Zadaci za vježbu

- Zadatak: Što ispisuje sljedeći program:
  - ```
    struct A { int x; };
    struct B { int y; };
    struct C : public A, public B { int z; };
    ```
  - ```
    C c;
    cout << &c << endl;
    C* pc = &c; cout << pc << endl;
    A* pa = pc;
    B* pb = pc;
    cout << pa << " " << pb << " " << pc << endl;
    C* p = (C*) pa; cout << p << endl;
    p = (C*) pb; cout << p << endl;
    ```

# Konverzije pri nasljeđivanju

- pc = (C*) pa;
  pc = static_cast<C*> (pb); // opasno

- pc = reinterpret_cast<C*> (pb); // lako moguća greška

- Ako imamo polimorfni tip (s bar jednom virtualnom funkcijom – npr. destruktorom), možemo koristiti:
  - pc = dynamic_cast<C*> (pb);
  - ako nije moguće napraviti konverziju, rezultat je null.

  - const_cast

# const_cast

- služi da uklonimo `const`, `volatile`...
- moramo biti oprezni da tada ne napravimo greške
- Primjer:
  - ```
    string s = "abcd";
    char *c = const_cast<char*> (s.c_str());
    c[2] = '\0';
    printf("%s %i\n", s.c_str(), s.size());
    // ab 4
    ```
  - ```
    set<int> si;
    si.insert(5); si.insert(10); si.insert(15);
    set<int>::iterator i = si.begin(); ++i;
    int& ri = const_cast<int&>(*i);
    ri = 17;
    for(i = si.begin(); i != si.end(); ++i)
        cout << *i << " ";
    // 5 17 15
    ```

# Doseg kod naslijeđivanja

- Prilikom naslijeđivanja doseg izvedene klase je ugniježđen unutar dosega bazne klase
  - ovakvo hijerarhijsko gniježđenje dosega omogućava da se članovima nadklase pristupa kao da je riječ o članovima izvedene klase
- Primjer:

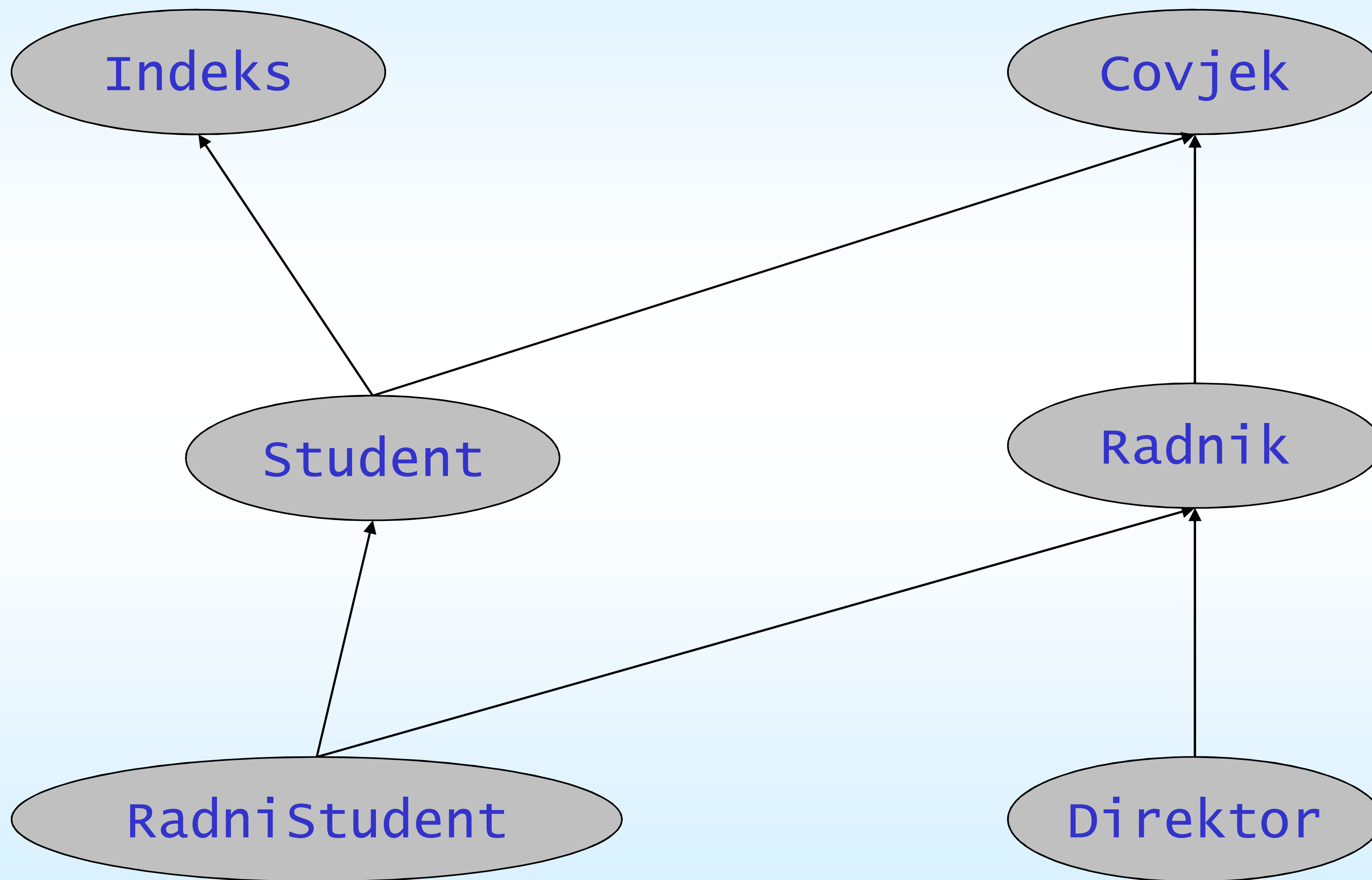
```
class ZooAnimal {
public:
    int ival;
};
```

```
class Bear : public ZooAnimal {
    int ival;
public:
    int mumble(int);
};
```

```
int ival; // varijabla u globalnom namespaceu
```

```
int Bear::mumble(int ival) {
    return ival + // lokalni
           ::ival + // globalni
           ZooAnimal::ival + // ZooAnimal
           Bear::ival; // Bear
}
```

# Hijerarhija klasa



# Hijerarhija klasa

- Primjer:
  - class Covjek { /* ... */ };
    class Radnik : public Covjek { /* ... */ };
    class Direktor : public Radnik { /* ... */ };
  - class Indeks { /* ... */ };
  - class Student : public Indeks, public Covjek { /* ... */ };
  - class RadniStudent : public Student, public Radnik { /* ... */ };

# Doseg kod višestrukog naslijeđivanja

- `struct Covjek { string s;};`
- ```
  struct Radnik : public Covjek { };
  struct Student : public Covjek { };
  ```
- ```
  struct StudentRadnik: public Radnik, Student {
      void ispis() {
          cout << Radnik::s
               << ",“ << Student::s << endl;
      }
  };
  ```
- ```
  StudentRadnik sr;
  //((Radnik)(sr)).s = "Ivica";
  ((Radnik*)(&sr))->s = "Ivica";
  // ili ((Radnik&)sr).s = "Ivica";
  sr.Student::s = "ivica";
  sr.ispis();
  ```
  ```

# Doseg kod višestrukog nasljeđivanja

- ```
  struct Covjek { string s;};
  ```

- ```
  struct Radnik : public Covjek { };
  ```

- ```
  struct DvoimeniRadnik: public Radnik, Covjek {
      void ispis() {
           cout << Radnik::s
                << ",“ << Covjek::s << endl;
      }
  };
  ```

- ```
  DvoimeniRadnik sr;
  ((Radnik*)(&sr))->s = "Ivica";
  sr.Covjek::s = "ivica";
  sr.ispis();
  ```

# Virtualno nasljeđivanje

- Da bi izbjegli dvostruku klasu Covjek u klasi StudentRadnik, potrebno je koristiti virtualno nasljeđivanje.
  - ```
    struct Radnik : virtual public Covjek { };
    struct Student : virtual public Covjek { };
    ```
  - ```
    struct StudentRadnik: public Radnik, Student {
     void ispis() {
           cout << Radnik::s
                << ",“ << Student::s << endl;
     }
    };
    ```
  - ```
    StudentRadnik sr;
    ((Radnik&)sr).s = "Ivica";
    sr.Student::s = "ivica";
    sr.ispis();
    ```
    ```

# Virtualno nasljeđivanje – konstruktori

- redoslijed izvršavanja konstruktora je drugačiji
- odredite redoslijed izvršavanja konstruktora prilikom kreiranja svake od klasa.
  - struct A { A() { cout << "A "; } };
  - struct V { V() { cout << "V "; } };
  - struct W : A, virtual V { W() { cout << "W "; } };
  - struct B : A, W {};
  - struct C : virtual V, W {};
  - struct D : B, virtual W, virtual V {};

# Virtualno nasljeđivanje – konstruktori

- redoslijed izvršavanja konstruktora je drugačiji
- kod svake izvedene klase moramo navesti konstruktore svih virtualno naslijeđenih klasa
  - struct Covjek { string s; Covjek(string ime) { s = ime; cout << ime << endl; } };
  - struct Radnik : virtual public Covjek { Radnik(string ime) : Covjek(ime) {} };
  - struct Student : virtual public Covjek {  Student(string ime) : Covjek(ime) {} };
  - struct StudentRadnik: public Radnik, Student {
    StudentRadnik(string ime) : Covjek(ime), Radnik(ime), Student(ime) {}
    };
  - void main() {
    StudentRadnik sr("Ivica");
    }