ФАКУЛТЕТ ЗА ЕЛЕКТРОТЕХНИКА И ИНФОРМАЦИСКИ ТЕХНОЛОГИИ

Обопштување и виртуелни функции

– Податочни структури и програмирање–

Наследување при изведувањето на класи (1)

- За сите (`public` и `protected`) функции од базната класа важи:
  - **тие се наследуваат**
    - во изведе**ните** класи (за `public` наследување)
    - во изведе**ната** класа (за `protected` наследување)
  - ако во изведената класа се редефинира одредена наследена функција (макар и само една нејзина верзија), **сите нејзини преоптоварени верзии од базната класа** стануваат **скриени** за изведената класа

Наследување при изведувањето на класи (2)

```
class BC {
public:
        int f1(int x);
        int f1(int x, float y);
        int f2(int x);
        . . .
};
class DC : public BC {
public:
        // f1 ne se redefinira
};
int main() {
        DC d, e;
        d.f1(5);
        e.f1(2, 3.45);
        d.f2(3);
        return 0;
}
```

наследено од BC

наследено од BC

наследено од BC

```
class BC {
public:
        int f1(int x);
        int f1(int x, fl
        int f2(int x);
        ...
};
class DC : public BC
public:
        int f1(int x); //se redefinira
};
int main() {
        DC d, e;
        d.f1(5); // OK call DC::f1(int)
        d.f2(4); // OK call BC::f2(int)
        e.f1(2, 3.45); // ne moze!!!
        return 0;
}
```

штом во изведената класа се редефинира некоја функција која постои во базната класа (без оглед на интерфејсот), не се наследува ниту една верзија од таа функција!

не се наследува од BC

Публикување при `private` изведување

- При `private` изведувањето, можно е само **дел од `public` методите од базната класа да се прогласат за достапни** и во изведената класа со помош на `using` директивата

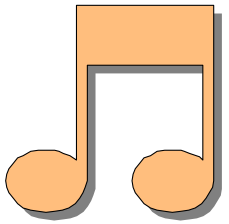
```
class Pet {
public:
        char eat() const { return 'a'; }
        int speak() const { return 2; }
        float sleep() const { return 3.0; }
        float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // Private inheritance
public:
        using Pet::eat; // Name publicizes member
        using Pet::sleep; // Both overloaded members exposed
};
```

```
int main() {
        Goldfish bob;
        bob.eat();
        bob.sleep();
        bob.sleep(1);
        //! bob.speak();
        // Error: private
        // member function
}
```

Изведување на класи и обопштување (1)

- **При изведувањето**, во изведената класа **се наследуваат сите податочни елементи** од базната класа (според дефинираните права за пристап) **и сите `public` и `protected` функции и оператори**, **ОСВЕН**:
  - **КОНСТРУКТОРОТ** (сите верзии вклучувајќи го и сору конструкторот)
  - **ДЕСТРУКТОРОТ**
- Кога се користи наследување, може да се каже дека **објект од изведената класа е истовремено и објект од базната класа** и поддржува одредена функционалност дефинирана во базната класа
- Соодветно, објект од изведената класа може да се употреби на секое место каде што се очекува објект од базната класа – **обопштување** (т.н. upcasting).

Изведување на класи и обопштување (2)



```
enum note { middleC, Csharp, Cflat };
class Instrument {
public:
        void play(note) const {}
};
class Duvacki : public Instrument {
        . . .
};
void tune(Instrument& i) {
        // ...
        i.play(middleC);
}
```

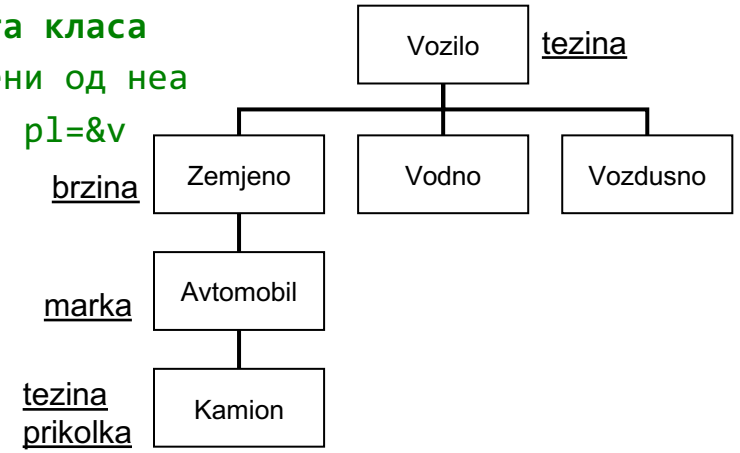
```
int main()
{
        Duvacki flejta;
        tune(flejta); //upcasting
}
```

Конверзија меѓу основната и изведените класи (1)

- Кога се користи наследување, може да се каже дека **објект од изведената класа е истовремено и објект од базната класа**
- Ова тврдење има **важни последици при доделувањето (=) на објекти**, **како и** во ситуациите **кога се користат покажувачи или референци** кон ваквите објекти
- При дозволените доделувања **автоматски се прави имплицитна конверзија** на типот од изведена класа (нешто поспецифично) во типот на основната класа (нешто поопшто)

Конверзија меѓу основната и изведените класи (2)

```
Vozilo v(1000), *vp;
Zemjeno l(1200, 145), *lp; // l е Zemjeno, но истовремено и Vozilo
Avtomobil a(790, 220, "Fiat"), *ap; // a е Avtomobil, но истовремено и Zemjeno и Vozilo
Kamion t(2600, 85, "Scania", 7000);
v = l;    // ова е OK, но, ќе се ископира само l.tezina во v.tezina
          // l=v не може, што да се ископира во l.speed ???
v = a; v = t; l = a; l = t; a = t; // се OK, но, a=v, a=l или t=a не може
vp = &v;           // покажувач од базната класа може
                   // да покажува кон објекти од базната класа
vp = &l; vp = &a;  // но, и кон објекти од класи изведени од неа
lp = &l; lp = &a;  // спротивното не е можно: ap=&l или pl=&v
Zemjeno &lr = a;
```



Ограничувања при обопштувањето (1)

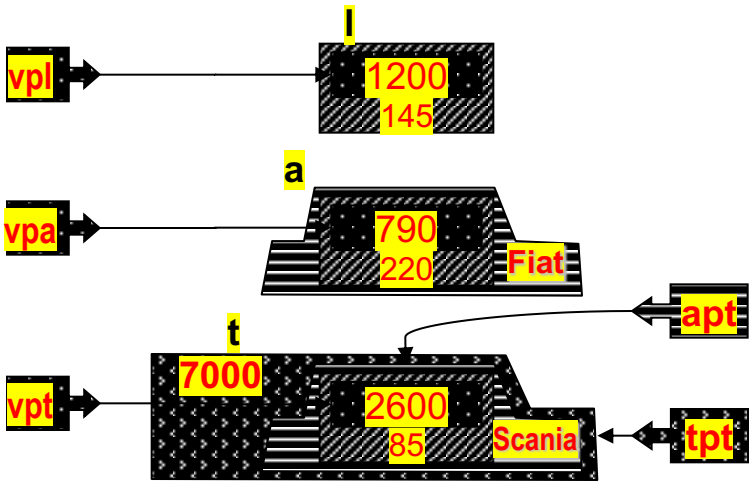
- Се разбира **со секое обопштување се губат информации** за конкретниот објект!

- Повикувајќи функции **преку покажувач на објект секогаш се повикуваат функциите од КЛАСАТА НА ПОКАЖУВАЧОТ**, **А НЕ** функциите од КЛАСАТА НА АКТУЕЛНИОТ ОБЈЕКТ на кои моментално покажува покажувачот!!!

Ограничувања при обопштувањето (2)

Vozilo::VratiTezina() дава 2600 наместо 9600

1200
790
2600
85
2600
9600 ✓




```
Vozilo *vpl, *vpa, *vpt;
Avtomobil *apt; Kamion *tpt;
Zemjeno l(1200, 145);
Avtomobil a(790, 220, "Fiat");
Kamion t(2600, 85, "Scania", 7000);
vpl = &l; vpa = &a; vpt = &t; apt = &t; tpt = &t;
cout << vpl->VratiTezina() << endl;  // се повикува Vozilo::VratiTezina()
cout << vpa->VratiTezina() << endl;  // се повикува Vozilo::VratiTezina()
                                     // vpa->VratiBrzina() не може да се повика
                                     // иако vpa покажува кон објект од класата Avtomobil
cout << vpt->VratiTezina() << endl;  // се повикува Vozilo::VratiTezina()
cout << apt->VratiBrzina() << endl;  // се повикува Avtomobil::VratiBrzina()
cout << apt->VratiTezina() << endl;  // се повикува Avtomobil::VratiTezina()
                                     // односно Vozilo::VratiTezina()
cout << tpt->VratiTezina() << endl;  // се повикува Kamion::VratiTezina()
```

Надминување на некои од ограничувањата

- Доколку ни е **точно познато кон кој тип на објект моментално покажува покажувачот** од основната класа, **можеме експлицитно да го кастираме** во покажувач кон објект од изведената класа и преку него да ни стане достапен интерфејсот на изведената класа

```
Kamion truck(2600, 85, "Scania", 7000);
Vozilo *vp;
vp = &truck; // vp sega pokazuva kon objekt Kamion
Kamion *trp;
trp = reinterpret_cast<Kamion *>(vp);
cout << "Kamion tip: " << trp->VratiIme() << endl;
cout << "Kamion tezina: " << trp->VratiTezina() << endl;
```

```
Kamion tip: Scania
Kamion tezina: 9600
```

- **РИЗИК**: Реинтерпретирањето на објектот (`Vozilo *` како `Kamion *`) ќе функционира правилно **само доколку vp навистина покажува кон објект** од класата `Kamion`. Преведувачот не може да го провери ова, бидејќи истото зависи од ситуацијата за време на извршувањето на програмата.

Полиморфизам (1)

- Повикувањето на функции преку покажувач (или референца) секогаш резултираше во повикување на функциите од класата на покажувачот, независно од типот на објектот кон кој моментално покажува покажувачот
- Ова однесување е познато како **рано или статичко поврзување** (static binding) бидејќи оваа информација е позната уште за време на преведувањето
- **Полиморфизмот** е **механизам** кој овозможува **покажувач од базната класа автоматски** (привидно) **да го промени својот тип во покажувач кон некоја од изведените класи**, во зависност од типот на објектот кон кој моментално покажува

Полиморфизам (2)

- ➔ Ова значи дека покажувачот `vp` ќе се однесува како `Kamion*` кога ќе покажува кон објект од типот `Kamion` или како `Avtomobil*` кога покажува кон објект `Avtomobil`.
- Полиморфизмот се реализира со можноста на C++ за т.н. **„задоцнето поврзување“** (late binding)
- Кај задоцнетото поврзување **одлуката која функција ќе се повика** во одреден случај (од базната или од изведените класи) повеќе не може да се донесе за време на преведувањето на програмата, туку истата **се одлага за во време на актуелното извршување на програмата**, кога единствено може да се донесе валидна одлука
- Задоцнетото поврзување во C++ се изведува со употреба на т.н. **виртуелни функции** членки на класата

Виртуелни функции (1)

once virtual – always virtual

- **Виртуелните функции се декларираат со** наведување на клучниот збор `virtual` пред декларацијата на функцијата во класата. Еднаш декларирана како виртуелна, функцијата останува виртуелна во сите изведени класи (дури и ако клучниот збор `virtual` не се повтори пред редефинирањето на функцијата во изведените класи)
- **Како виртуелни се декларираат функциите во базната класа** за **кои се очекува да бидат препокриени** (≠редефинирани) од страна на изведените класи (redefine ≠ override)
- Ефектите од задоцнетото поврзување се што при повикувањето на функција преку покажувач, ќе се повика **функцијата од класата од која е актуелниот објект** (независно од типот на покажувачот) доколку истата била **декларирана како виртуелна** во базната класа

Виртуелни функции (2)

once virtual – always virtual

```
Vozilo *vp = new Zemjeno(1000, 120);
delete vp; // object destroyed by ~Vozilo (not ~Zemjeno)
```

- **Во класите во кои е декларирана барем една виртуелна функција** и кои самите динамички алоцираат/деалоцираат меморија (или се очекува нивните наследници да го прават ова) **и деструкторот треба да биде деклариран како виртуелен**

```
class Vozilo
{
public:
        virtual ~Vozilo();
        virtual int VratiTezina() const;
};
```

- **Статичките функции не можат да бидат виртуелни** (бидејќи немаат `this` покажувач)

Виртуелни функции (3)

```
class Vozilo {
public:
        virtual int VratiTezina() const { return(tezina); }
        virtual void PostaviTezina(int wt) { tezina = wt; }
};
. . .
class Kamion : public Avtomobil {
public:
        void PostaviTezina(int engine_wt, int trailer_wt);
        int VratiTezina() const;
};
void Kamion::PostaviTezina(int engine_wt, int trailer_wt) {
        TezinaPrikolka = trailer_wt;
        Avtomobil::PostaviTezina(engine_wt);
}
int Kamion::VratiTezina() const {
        return (Avtomobil::VratiTezina() + TezinaPrikolka);
}
```

Виртуелни функции (4)

```
int main()
{
        Vozilo v(1200); // возило тешко 1200
        Kamion t(6000, 115, "Scania", 15000); // камион Scania
        // со кабина тешка 6000, брзина 115, и приколка тешка 15000

        Vozilo *vp; // generic pointer na Vozilo
        vp = &v;
        cout << vp->VratiTezina() << endl;
        // ќе се повика Vozilo::VratiTezina()
        vp = &t;
        cout << vp->VratiTezina() << endl;
        // ќе се повика Kamion::VratiTezina()
        cout << vp->VratiBrzina() << endl; // syntax error:
        // VratiBrzina() не е член на Vozilo
}
```

Преоптоварување и препокривање на функции (1)

Overloading & overriding

```
class Base {
public:
        virtual int f() const
        { cout << "Base::f()\n";
        return 1; }
        virtual void f(string) const {}
        virtual void g() const {}
};

class Derived1 : public Base {
public:
        void g() const {}
};

class Derived2 : public Base {
public:
        // Overriding a virtual function:
        int f() const
        { cout << "Derived2::f()\n";
        return 2; }
};
```

```
class Derived3 : public Base {
public:
        // Cannot change return type:
        //! void f() const{
        // cout << "Derived3::f()\n";}
};
class Derived4 : public Base {
public:
        // Change argument list:
        int f(int) const
        { cout << "Derived4::f()\n";
        return 4; }
};
```

Преоптоварување и препокривање на функции (2)

```
int main()
{
        string s("hello");
        Derived1 d1;
        int x = d1.f();
        d1.f(s);
        Derived2 d2;
        x = d2.f();
        //!  d2.f(s); // string version hidden
        Derived4 d4;
        x = d4.f(1);
        //!  x = d4.f(); // f() version hidden
        //!  d4.f(s); // string version hidden
        Base& br = d4;   // Upcast
                         //!  br.f(1); // Derived version unavailable
        br.f(); // Base version available
        br.f(s);// Base version available
}
```

Како се имплементира полиморфизмот (1)

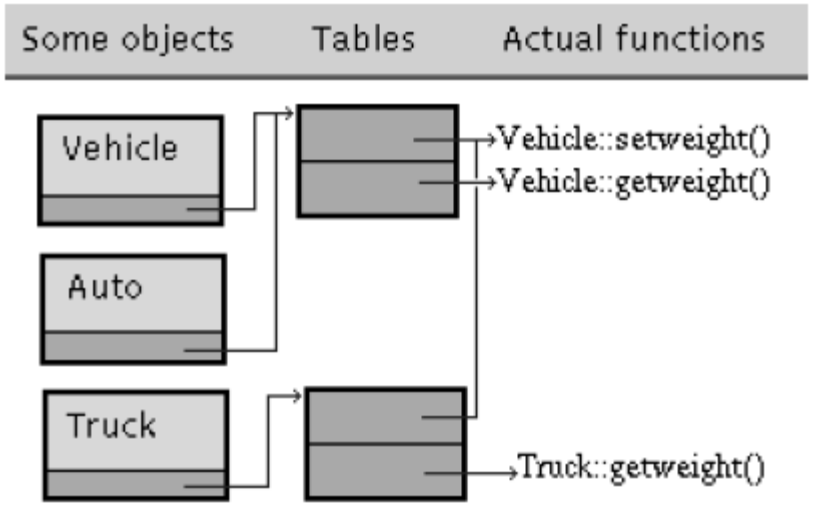
- Полиморфизмот чини **зголемени мемориски потреби**
- Со секој објект од класите кои имаат виртуелни функции, покрај меморијата потребна за чување на податочните членови **потребно е да се чува** уште еден **покажувач кон табела на виртуелни функции за класата**
- Со помош на оваа табела, за време на извршување на програмата **може да се одреди типот на објектот како и која функција треба да се повика** за него

Како се имплементира полиморфизмот (2)

```
class NoVirtual {
        int a;
public:
        void x() const {}
        int i() const { return 1; }
};

class OneVirtual {
        int a;
public:
        virtual void x() const {}
        int i() const { return 1; }
};

class TwoVirtuals {
        int a;
public:
        virtual void x() const {}
        virtual int i() const { return 1; }
};
```

```
int main() {
        cout << "int: " << sizeof(int) << endl;
        cout << "NoVirtual: "
             << sizeof(NoVirtual) << endl;
        cout << "void* : " << sizeof(void*) << endl;
        cout << "OneVirtual: "
             << sizeof(OneVirtual) << endl;
        cout << "TwoVirtuals: "
             << sizeof(TwoVirtuals) << endl;
}
```

```
int: 4
NoVirtual: 4
void* : 4
OneVirtual: 8
TwoVirtuals: 8
```



Чисти виртуелни функции – Апстрактни класи (1)

- Често пати во објектниот дизајн има потреба да се развие **базна класа** која само ќе го пропише интерфејсот за класите изведени од неа. За ова е потребен начин на кој на наследниците ќе им се наложи да препокријат одредени методи (само декларирани) во базната класа.
- Во C++ ова се постига со декларирање на т.н. **чисти виртуелни функции во базната класа**. Чистите виртуелни функции немаат дефиниција (тело) во базната класа.
- Со дефинирањето на чисти виртуелни функции се дефинира протоколот кој треба да го следат изведените класи (се наложува прототипот на функциите кои изведените класи **мораат** да ги препокријат).
- Класата која има барем една чиста виртуелна функција, се нарекува **апстрактна класа**.

Чисти виртуелни функции – Апстрактни класи (2)

- Апстрактната класа **не може да има објекти**! **Не е можно да се декларираат објекти** од апстрактна класа (дозволено да се декларираат покажувачи и референци од апстрактна класа).
- Апстрактната класа е **пред сè наменета да служи за изведување на други конкретни класи од неа**, кои се задолжени да следат некои правила – да дефинираат сопствени функции според наложениот интерфејс на чистите виртуелни функции кои мораат да ги препокријат
- **Доколку** во изведената класа **не се препокријат сите чисти виртуелни функции** од базната класа **и самата изведена класа ќе биде апстрактна**
- За декларирање на чиста виртуелна функција, пред декларацијата на функцијата се наведува **префиксот** `virtual` а на крајот се наведува **суфиксот** `=0`

Чисти виртуелни функции – Апстрактни класи (3)

```
class Shape { // Abstract class
public:
   virtual void Draw() const = 0; // pure virtual function
   virtual ~Shape() {}
};
class Circle : public Shape {
public:
   Circle(float _x, float _y, float _r) : x(_x), y(_y), r(_r) { }
   void Draw() const // must override the pure virtual function
   {
     cout << "Drawing Circle (" << x << ',' << y << ',' << r << ')' << endl;
   }
private: float x, y, r;
};
class Rectangle : public Shape {
public:
   Rectangle(float _x, float _y, float _w, float _h) : x(_x), y(_y), w(_w), h(_h) {}
   void Draw() const // must override the pure virtual function
   {
     cout << "Drawing Rectangle (" << x << ',' << y << ',' << w << ',' << h << ')' << endl;
   }
private: float x, y, w, h;
};
```

Чисти виртуелни функции – Апстрактни класи (4)

```
int main() {
        //Shape s; // не може да се декларира објект од апстрактна класа
        Shape *p; // но може покажувач
        Shape *sl[] = { new Circle(1,2,3), new Rectangle(2,3,4,5), new
                        Circle(0,0,1), 0 };
        int i = 0;
        while (p = sl[i++])
                p->Draw();
        for (int i = 0; sl[i]; delete sl[i++]);
        return(0);
}
```

```
Drawing Circle (1,2,3)
Drawing Rectangle (2,3,4,5)
Drawing Circle (0,0,1)
```

Чисти виртуелни функции – Апстрактни класи (5)

- За специјални **употреби дозволено е и чистите виртуелни функции да можат да се дефинираат** (да имаат тело) **во апстрактната класа** (која и покрај тоа ќе остане апстрактна)
- Ваквите функции **можат да се повикуваат единствено од функциите во наследниците на апстрактната класа** и тоа преку scope операторот и името на базната класа

Чисти виртуелни функции – Апстрактни класи (6)

```
class Shape { // Abstract class
public:
   virtual void Draw() const = 0;
};
void Shape::Draw() const { cout << "drawing "; }

class Circle : public Shape {
public:
   Circle(float _x, float _y, float _r) : x(_x), y(_y), r(_r) { }
   void Draw() const // still must override the pure virtual function
   { Shape::Draw(); cout << "Circle (" << x << ',' << y << ',' << r << ')' << endl; }
private: float x, y, r;
};

class Rectangle : public Shape {
public:
   Rectangle(float _x, float _y, float _w, float _h) : x(_x), y(_y), w(_w), h(_h) {}
   void Draw() const // still must override the pure virtual function
   { Shape::Draw(); cout << "Rectangle (" << x << ',' << y << ',' << w << ',' << h << ')'
<< endl; }
private: float x, y, w, h;
};
```

```
drawing Circle (1,2,3)
drawing Rectangle (2,3,4,5)
```

`Static` членови во класите

- Кога е потребно да се располага со **податоци кои се единствени за целата класа** (и не е потребно во секој објект да се чува нивна копија), **тие можат да се декларират како `static`** податочни членови на класата

```
class ST {
public:
        int x, y; // секој објект ќе има сопствени x и y
        static int z; // z е единствено - заедничко за сите објекти
};
int ST::z = -1;    // иницијализација
                   // ST::z постои и без да постојат објекти од класата
int main()
{
        ST a, b;
        a.x = 1; a.y = 2;
        cout << a.x << ' ' << a.y << ' ' << a.z << endl;
        a.z = 0;
        b.x = 6; b.y = 7;
        cout << b.x << ' ' << b.y << ' ' << b.z << endl;
        return(0);
}
```

```
1 2 -1
6 7 0
```

`Static` функции во класите (1)

- **Функциите** декларирани како `static` во класата смеат да **пристапуваат само до `static` податочните елементи**
- За разлика од другите функции членки на класата, **`static` функциите не мора** (но може) **да се повикаат преку постоечки објект од класата**. Може да се повикуваат преку името на класата со scope операторот
- `static` функциите **немаат `this` покажувач**
- `static` функциите **не можат да бидат виртуелни**

Static функции во класите (2)

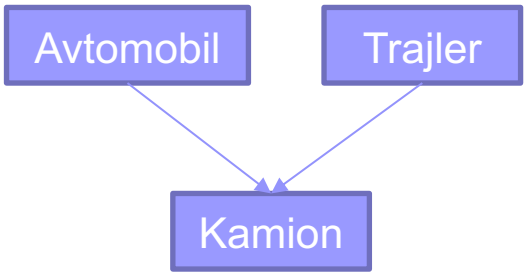
```
class ST {
public:
        ST(int a, int b) : x(a), y(b) {}
        int xo() const { return x + offset; }
        int yo() const { return y + offset; }
        static void setoffset(int a) { offset = a; }
        static int getoffset() { return offset; }
private:
        int x, y;
        static int offset;
};
int ST::offset = 0; // иако е приватна static променливата смее еднаш да се
иницијализира надвор од main()

int main() {
        ST::setoffset(ST::getoffset() + 1);
        ST a(2, 3);
        const ST b(4, 5);
        cout << b.xo() << ' ' << b.yo() << endl;
        b.setoffset(-1);
        cout << a.xo() << ' ' << a.yo() << endl;
        return(0);
}
```

```
5 6
1 2
```

Повеќекратно наследување

- Во C++ **класа може да се изведе и од две или повеќе класи истовремено**. При тоа изведената класа ги наследува особините од сите основни класи

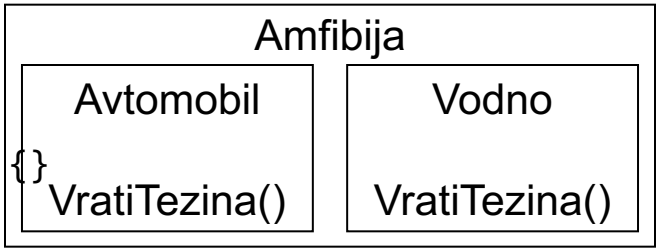
```
class Trejler
{
public:
   Trejler() {}
   Trejler(int twt) : tezinaPrikolka(twt) {}
   void PostaviTezina(int twt) { tezinaPrikolka = twt; }
   int VratiTezina() const { return tezinaPrikolka; }
private: int tezinaPrikolka;// podatoci
};

class Kamion : public Avtomobil, public Trejler {
public: // konstruktori
   Kamion() {}
   Kamion(int engine_wt, int sp, char const *nm, int trailer_wt) :
         Avtomobil(engine_wt, sp, nm), Trejler(trailer_wt) {}
   void PostaviTezina(int engine_wt, int trailer_wt) {
         Trejler::PostaviTezina(trailer_wt);
         Avtomobil::PostaviTezina(engine_wt);}
   int VratiTezina() const { return (Avtomobil::VratiTezina() + Trejler::VratiTezina());
}
   // bez podatoci
};
```

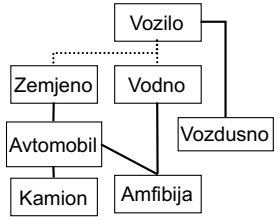


Проблеми кај повеќекратното изведување (1)

- Но, што кога се изведува класа од две или повеќе класи од кои некои имаат **заеднички предок**?

```
class Vodno : public Vozilo {
public: // konstruktori
   Vodno() {}
   Vodno(int wt, int l) : Vozilo(wt), dolzina(l) {}
   void PostaviDolzina(int l) { dolzina = l; }
   int VratiDolzina() const { return dolzina; }
private: int dolzina; // podatoci
};

class Amfibija : public Avtomobil, public Vodno {
public:
   Amfibija() {}
   Amfibija(int wt, int sp, char const *nm, int len) :
         Avtomobil(wt, sp, nm), Vodno(wt, len) {} // 2x wt !!!
};
int main() {
   Amfibija am(2500, 35, "T72", 7);
   cout << "Voziloto tezi " << am.VratiTezina() << endl;
   return (0);
}
```





Постои една VratiTezina() наследена од Vozilo преку Zemjeno и една наследена преку Vodno

```
request for member `VratiTezina' is ambiguous
candidates are: virtual int Vozilo::VratiTezina() const
                virtual int Vozilo::VratiTezina() const
```

Проблеми кај повеќекратното изведување (2)

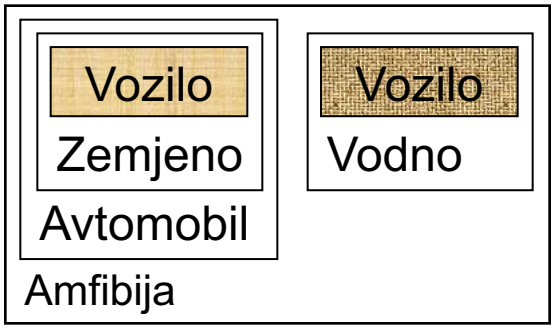
- **Еден начин** да се надмине ова е да се нагласи точно која `VratiTezina()` сакаме да ја повикаме со употреба на scope оператор

```
cout << "Voziloto tezi " << am.Avtomobil::VratiTezina() << endl;
```

- **Друг начин** би бил во класата `Amfibija` да се редефинира функцијата `VratiTezina()` во која може експлицитно да се наведе која верзија да се повикува

```
class Amfibija : public Avtomobil, public Vodno {
public:
    Amfibija() {}
    Amfibija(int wt, int sp, char const *nm, int len) :
            Avtomobil(wt, sp, nm), Vodno(wt, len) {} // 2x wt !!!
    void PostaviTezina(int wt) { Avtomobil::PostaviTezina(wt); }
    int VratiTezina() const { return Avtomobil::VratiTezina(); }
};
```

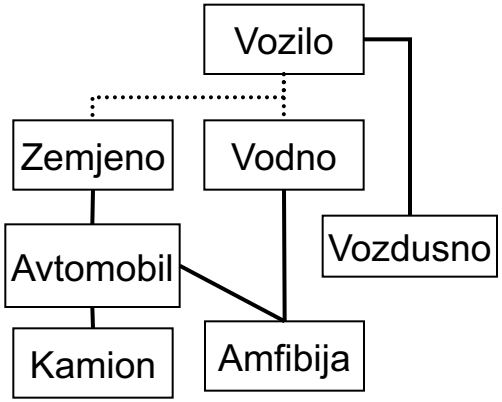
Но со ова не се надминува фактот што и понатаму во објектите од класата **`Amfibija`** има по 2 податочни члена **`tezina`** од класата **`Vozilo`** – едниот наследен преку **`Avtomobil`**, а другиот преку **`Vodno`**

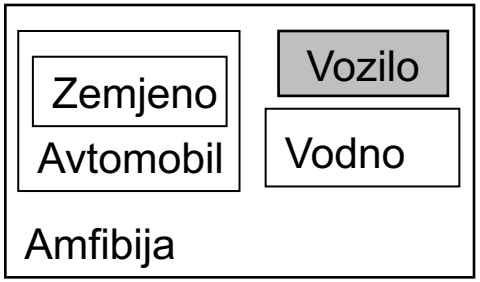


Виртуелна базна класа

- Избегнување на дуплирањето на **`Vozilo::tezina`** може да се постигне со **прогласување на базната класа** (чие дуплирање се појавува) **како виртуелна** при изведувањето на класите од неа

```
class Zemjeno : virtual public Vozilo {
... }
class Vodno : virtual public Vozilo {
... }
class Amfibija : public Avtomobil, public Vodno {
public:
   Amfibija() {}
   Amfibija(int wt, int sp, char const *nm, int len)
   { PostaviIme(nm); PostaviBrzina(sp);
     PostaviDolzina(len); PostaviTezina(wt); }
};
int main() {
   ...
   cout << "Voziloto tezi " << am.VratiTezina() << endl; }
```

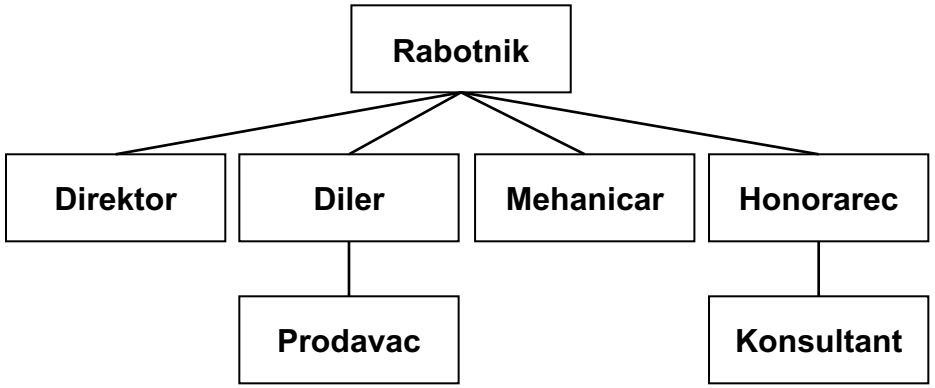




Real-world problem

- Имаме фирма за чии вработени треба да пресметуваме плата. Во фирмата работат различни профили на работници, чија плата се пресметува на различен начин. За вработените платата се изразува во бодови и вредноста за исплата се добива со множење на бодовите по вредноста на бодот кој е единствен за целата фирма. Платите на соработниците не зависат од вредноста на бодот и се одредуваат директно.
- Вработени:
  - Директор – прима фиксна плата
  - Механичар – платен е според бројот на работни часови кои работел во месецот, според одреден надомест за 1 час работа
  - Дилер – платен е според бројот на остварени продажби (добива одреден бонус од секоја остварена продажба, но нема работно време)
  - Продавач – има одредена фиксна основица за плата на која се додаваат стимулативни бонуси според остварените продажби (има работно време во продавница)
- Соработници:
  - Хонорарец – привремено ангажиран од страна на фирмата по договор за дело за договорена сума која не зависи од вредноста на бодот.
  - Консултант – ангажиран паушално со (повеќе)годишен договор со одредена динамика на исплата (месечно на пр.). Според договорот во одредени случаи му се исплаќаат и бонуси за трошоци (над договорената сума). Не зависи од вредноста на бодот.
- За сите лица (вработени и соработници) се чуваат и дополнителни информации кои не зависат од нивниот работен ангажман (име и презиме, адреса, банкарска сметка, ...)

Плати: прва итерација (1)




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

class Rabotnik {
public:
        Rabotnik(char *name = "") :ime(name) { }  // Konstruktor
        virtual ~Rabotnik() { }
        static void SetBod(float newbod) { bod = newbod; }
        const string &Ime(void) const { return ime; }
        float VrednostBod(void) const { return bod; }
        void SmeniIme(const string &novoime) { ime = novoime; }
        virtual float Plata(void) const = 0;
        virtual void Show(void) const = 0;
private:
        string ime;    // ime na vraboten
        static float bod;// vrednost na 1 bod vo denari
};
```

Плати: прва итерација (2)

```
class Direktor : public Rabotnik {
public:
        Direktor(char *ime = "", float plata = 0) : Rabotnik(ime),
bod_plata(plata) {}
        ~Direktor() { }
        float Plata(void) const { return bod_plata*VrednostBod(); }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void Osnovica(float vrednost) { bod_plata = vrednost; }
private:
        float bod_plata;// plata izrazena vo bodovi
};
class Mehanicar : public Rabotnik {
public:
        Mehanicar(char *ime = "", float plata_po_cas = 0, int rab_casovi = 0) :
                Rabotnik(ime), satnina(plata_po_cas), casovi(rab_casovi) { }
        ~Mehanicar() { }
        float Plata(void) const { return satnina*casovi*VrednostBod(); }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void RabotniCasovi(int cas) { casovi = cas; }
        void Satnica(float vrednost) { satnina = vrednost; }
private:
        float satnina;     // vrednost na eden cas vo bodovi
        int casovi;        // broj na casovi
};
```

Плати: прва итерација (3)

```
class Diler : public Rabotnik
{
public:
        Diler(char *ime = "", float prov = 0, int brprod = 0) :
                Rabotnik(ime), bonus(prov), prodazbi(brprod) { }
        ~Diler() { }
        float Plata(void) const { return bonus*prodazbi*VrednostBod(); }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void Bonus(float vrednost) { bonus = vrednost; }
        void Prodazbi(int br) { prodazbi = br; }
private:
        float bonus;       // procent od prodazbata koj e za dilerot
        int prodazbi;      // ostvarena prodazba izrazena vo bodovi
};

class Prodavac : public Diler
{
public:
        Prodavac(char *ime = "", float osnova = 0, float prov = 0, int brprod = 0) :
                Diler(ime, prov, brprod), mesecna(osnova) { }
        ~Prodavac() { }
        float Plata(void) const { return mesecna*VrednostBod() + Diler::Plata(); }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void Osnovica(float vrednost) { mesecna = vrednost; }
Private: float mesecna;    // fiksen del od mesecnata plata
};
```

Плати: прва итерација (4)

```
class Honorarec : public Rabotnik
{
public:
        Honorarec(char *ime = "", float vrednost = 0) :
                Rabotnik(ime), honorar(vrednost) { }
        ~Honorarec() { }
        float Plata(void) const { return honorar; }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void Honorar(float vrednost) { honorar = vrednost; }
private:
        float honorar;     // honorar vo denari
};
class Konsultant : public Honorarec
{
public:
        Konsultant(char *ime = "", float osnova = 0, float usluga = 0) :
                Honorarec(ime, osnova), bonus(usluga) { }
        ~Konsultant() { }
        float Plata(void) const { return bonus + Honorarec::Plata(); }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void Bonus(float vrednost) { bonus = vrednost; }
private:
        float bonus;       // dodaten varijabilen del na platata vo denari
};
```

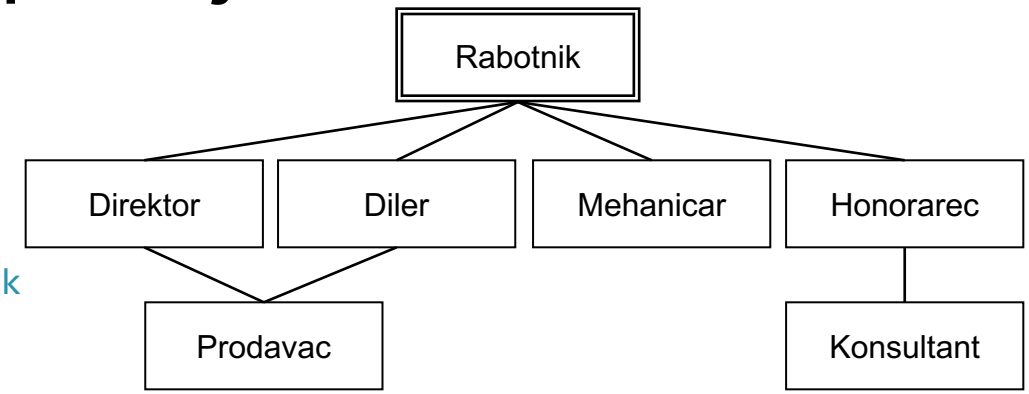
Плати: прва итерација (5)

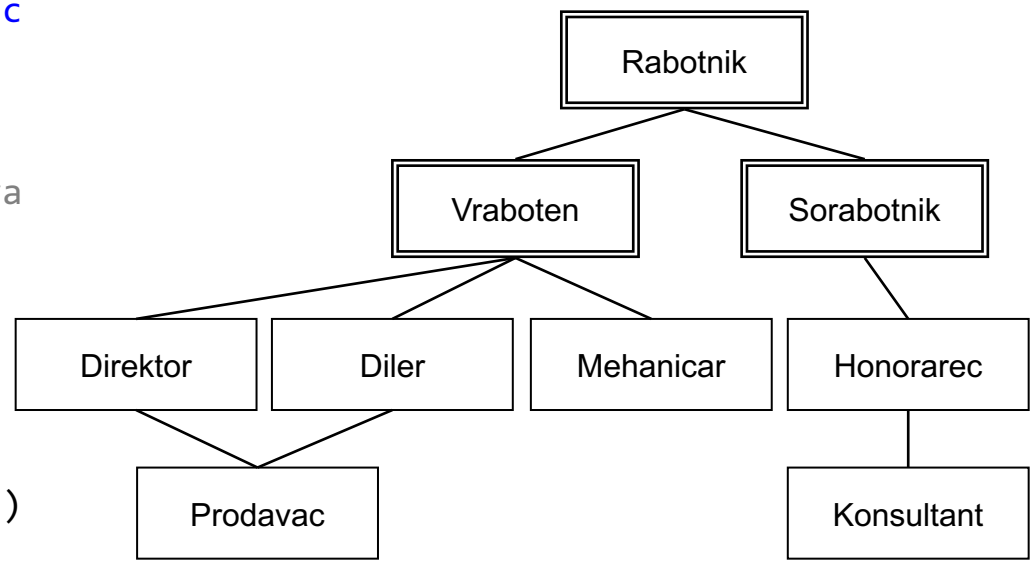
```
float Rabotnik::bod = 10.0;
int main()
{
        Rabotnik::SetBod(12.0);
        Rabotnik *Firma[] = {
                new Direktor("direktor1",1000.0),
                new Mehanicar("mehanicar1",3.5,45),
                new Prodavac("prodavac1",300.0,30.0,5),
                new Prodavac("prodavac2",400.0,25.0),
                new Diler("diler1",50.0,3),
                new Honorarec("honorarec1",100.0),
                new Konsultant("konsultant1",500.0),
                0 };

        float total = 0.0;
        Rabotnik *r;
        for (int i = 0; r = Firma[i]; i++)
        {
                r->Show(); cout << endl;
                total += r->Plata();
        }
        cout << "Vkupno za isplata " << total << endl;
        for (int i = 0; Firma[i]; delete Firma[i++]);
        return(0);
}
```

```
direktor1 12000
mehanicar1 1890
prodavac1 5400
prodavac2 4800
diler1 1800
honorarec1 100
konsultant1 500
Vkupno za isplata 26490
```

Плати: втора итерација

```
class Rabotnik {
public:
...
};
class Direktor : virtual public Rabotnik
{ ... };
class Diler : virtual public Rabotnik{
... };

class Prodavac : public Direktor, public
Diler
{
public:
   Prodavac(char *ime = "", float osnova
= 0, float prov = 0, int brprod = 0)
   { SmeniIme(ime); Bonus(prov);
Prodazbi(brprod); Osnovica(osnova); }
   ~Prodavac() { }
   float Plata(void) const { return
Direktor::Plata() + Diler::Plata(); }
   void Show(void) const { cout << Ime()
<< ' ' << Plata(); }
};
```





Плати: конечна (1)

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

class Rabotnik
{
public:
        Rabotnik(char *name = "") :ime(name) { }  // Konstruktor
        virtual ~Rabotnik() { }
        const string &Ime(void) const { return ime; }
        void SmeniIme(const string &novoime) { ime = novoime; }
        virtual float Plata(void) const = 0;
        virtual void Show(void) const = 0;
private:
        string ime; // ime na vraboten
};
```

Плати: конечна (2)

```
class Vraboten : public Rabotnik
{
public:
        Vraboten(char *name = "") :Rabotnik(name) { }  // Konstruktor
        virtual ~Vraboten() { }
        static void SetBod(float newbod) { bod = newbod; }
        virtual float Plata(void) const = 0;
        virtual void Show(void) const = 0;
protected:
        static float bod; // vrednost na 1 bod vo denari
};

class Sorabotnik : public Rabotnik
{
public:
        Sorabotnik(char *name = "") :Rabotnik(name) { }  // Konstruktor
        virtual ~Sorabotnik() { }
        virtual float Plata(void) const = 0;
        virtual void Show(void) const = 0;
};
```

Плати: конечна (3)

```
class Direktor : virtual public Vraboten
{
public:
        Direktor(char *ime = "", float plata = 0) : Vraboten(ime), bod_plata(plata) {}
        ~Direktor() { }
        float Plata(void) const { return bod_plata*bod; }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void Osnovica(float vrednost) { bod_plata = vrednost; }
private: float bod_plata; // plata izrazena vo bodovi
};

class Mehanicar : public Vraboten
{
public:
        Mehanicar(char *ime = "", float plata_po_cas = 0, int rab_casovi = 0) :
                Vraboten(ime), satnina(plata_po_cas), casovi(rab_casovi) { }
        ~Mehanicar() { }
        float Plata(void) const { return satnina*casovi*bod; }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void RabotniCasovi(int cas) { casovi = cas; }
        void Satnica(float vrednost) { satnina = vrednost; }
private:
        float satnina; // vrednost na eden cas vo bodovi
        int casovi;         // broj na casovi
};
```

Плати: конечна (4)

```
class Diler : virtual public Vraboten
{
public:
        Diler(char *ime = "", float prov = 0, int brprod = 0) :
                Vraboten(ime), bonus(prov), prodazbi(brprod) { }
        ~Diler() { }
        float Plata(void) const { return bonus*prodazbi*bod; }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void Bonus(float vrednost) { bonus = vrednost; }
        void Prodazbi(int br) { prodazbi = br; }
private:
        float bonus;        // procent od prodazbata koj e za dilerot
        int prodazbi;       // ostvarena prodazba izrazena vo bodovi
};

class Prodavac : public Direktor, public Diler
{
public:
        Prodavac(char *ime = "", float osnova = 0, float prov = 0, int brprod = 0)
        { SmeniIme(ime); Bonus(prov); Prodazbi(brprod); Osnovica(osnova); }
        ~Prodavac() { }
        float Plata(void) const { return Direktor::Plata() + Diler::Plata(); }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
};
```

Плати: конечна (5)

```
class Honorarec : public Sorabotnik {
public:
        Honorarec(char *ime = "", float vrednost = 0) :
                Sorabotnik(ime), honorar(vrednost) { }
        ~Honorarec() { }
        float Plata(void) const { return honorar; }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void Honorar(float vrednost) { honorar = vrednost; }
private:
        float honorar; // honorar vo denari
};

class Konsultant : public Honorarec {
public:
        Konsultant(char *ime = "", float osnova = 0, float usluga = 0) :
                Honorarec(ime, osnova), bonus(usluga) { }
        ~Konsultant() { }
        float Plata(void) const { return bonus + Honorarec::Plata(); }
        void Show(void) const { cout << Ime() << ' ' << Plata(); }
        void Bonus(float vrednost) { bonus = vrednost; }
private:
        float bonus; // dodaten varijabilen del na platata vo denari
};
```

```
float Vraboten::bod = 10.0;
int main() {
Vraboten::SetBod(12.0);
. . .}
```