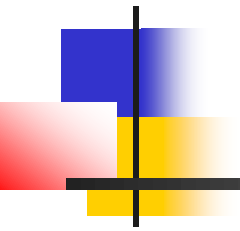


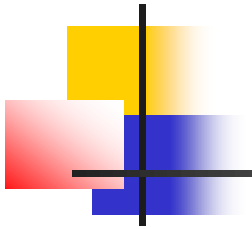
Základy objektového programování v C++

Dědičnost a spřátelenost

Jan Fesl

jfesl@prf.jcu.cz

Druhy atributů a metod v objektu



- **private** – viditelné (přístupné) pouze v rámci metod určité třídy
- **public** – viditelné (přístupné) v metodách určité třídy, i navenek

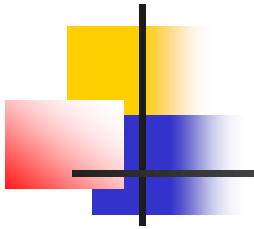
Ukázka deklarace objektu (třídy)

```
class obrazec
{
    private:
    string barva;

    public:
    int obvod;
    int obsah;
    obrazec() {}
    ~obrazec() {}
    string zjistiBarvu() {return barva}
    void nastavBarvu(string b) {barva =
    b}
};
```

```
int main(int argc, argv *char[])
{
// vytvoreni instance
obrazec o1;
// spravne
o1.obvod = 100;
// chyba
o1.barva  = "cervena";
}
```

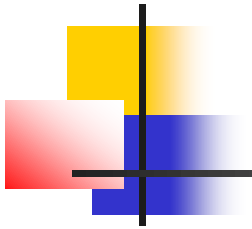
Jak to vypadá s organizací atributů v paměti ?



Dědičnost (inheritance)

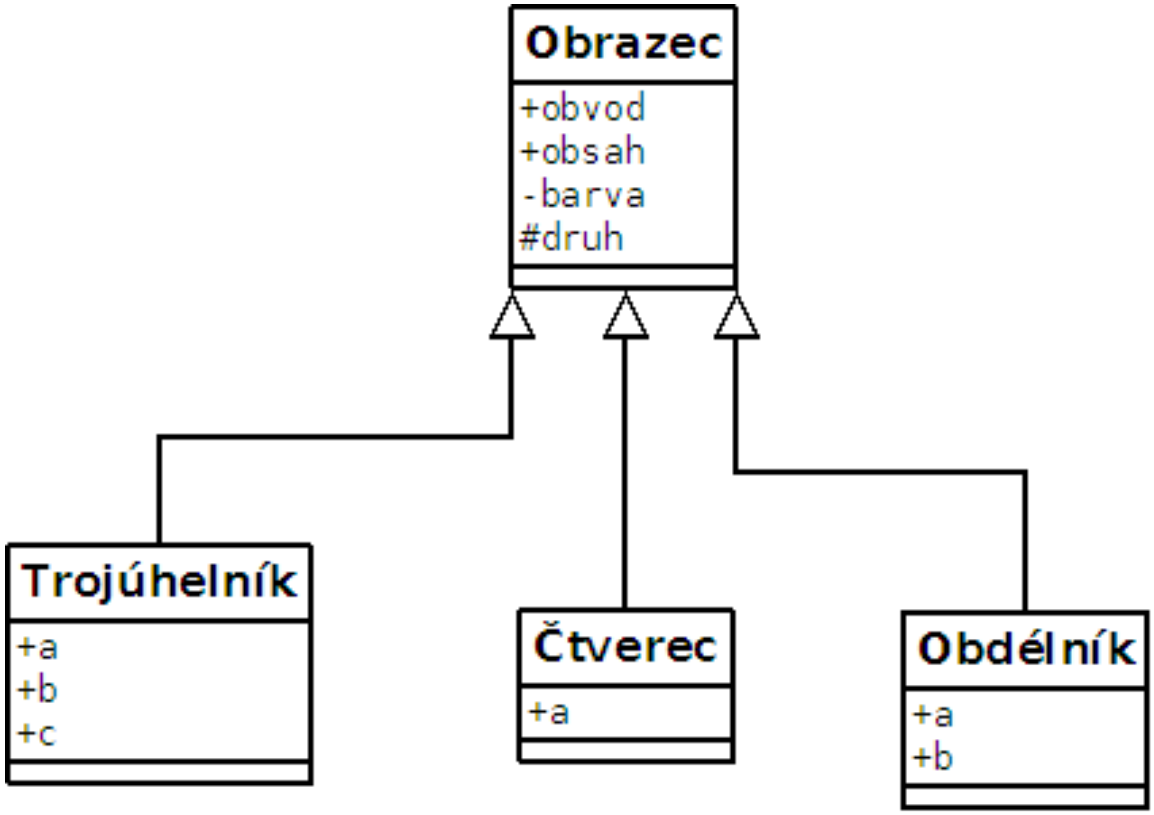
- Inspirace je ve skutečném světě
- Potomek přebírá vlastnosti (atributy) předka i jeho schopnosti (metody)
- Programátor ovšem působí jako “stvořitel”, který rozhoduje o tom, jakým způsobem k dědění dochází

Druhy atributů a metod objektu zhlediska dědičnosti



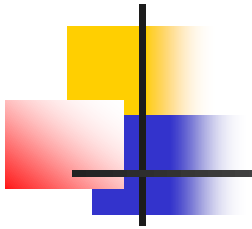
- **private** – viditelné pouze v rámci metod určité třídy
- **public** – viditelné i navenek, dále ve všech metodách třídy, i ve všech metodách tříd odvozených
- **protected** – viditelné v metodách třídy i v metodách potomků, ovšem neviditelné navenek

Hierarchie tříd a dědění



public(+), private(-), protected(#)

Způsoby dědění atributů a metod



- **private** – všechny **public, protected** atributy předka budou zděděné jako private u potomka, private atributy se dědí nepřímo a potomek k nim již přístup nemá
- **public** – všechny zděděné **public, protected** aributy předka jsou stejné i u potomka
- **protected** – všechny public a protected atributy předka jsou zděděné jako protected u potomka a private atributy předka jsou zděděné jako private

Ukázka dědění

```
class obrazec
{
    private:
    string barva;

    public:
    int obvod;
    int obsah;
    obrazec() {}
    ~obrazec() {}
    string zjistiBarvu() {return barva;}
    void nastavBarvu(string b) {barva = b;}
    void zjistiDruh() {return druh;}

    protected:
    string druh;
};
```

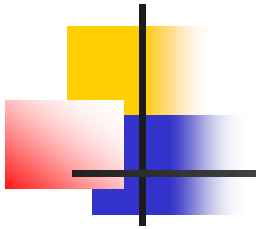
```
class trojuhelnik : public obrazec
{
private:
public:
int a;
int  b;
int c;
};
```

```
trojuhelnik t;
t.a = 1;
t.obvod = 23;
```

Přístup ke zděným atributům je shodný s přístupem k vlastním atributům

Jak to vypadá s organizací atributů v paměti ?

Přiřazení hodnoty mezi předkem a potomkem



- **Platí, že vždy můžeme předkovi přiřadit hodnotu potomka – protože potomek vždy obsahuje ty samé atributy co původní předek + nějaké navíc**

- Obráceně to implicitně nelze, protože by některé atributy zůstaly nepřiřazeny (potomek má nějaké navíc)

Překrytí atributů při dědění, modifikace přístupových práv

```
class obrazec
{
    private:
    string barva;

    public:
    int obvod;
    int obsah;
    obrazec() {}
    ~obrazec() {}
    string zjistiBarvu() {return barva;}
    void nastavBarvu(string b) {barva = b;}
    void zjistiDruh() {return druh;}

    protected:
    string druh;
};
```

```
class trojuhelnik : public obrazec
{
private:
// změna atributů public ➔ private
int obrazec::obsah;
public:
// překrytí atributu
int obvod;
// změna atributů protected ➔ public
string obrazec::druh;
int a;
int  b,c;
};
```

Přístup k překrytým atributům předka

- Pokud je atribut předka, lze k tomuto přistoupit pomocí operátoru ::
- Přistupovat lze buď v metodách potomka, či přes instanci potomka
- Pokud přistupujeme k atributům předka přes potomka mimo třídu, lze přistupovat k těmto pouze v případě, když dědíme způsobem public

Přístup k překrytým atributům předka v metodě potomka

```
class obrazec
{
    private:
    string barva;

    public:
    int obvod;
    int obsah;
    obrazec() {}
    ~obrazec() {}
    string zjistiBarvu() {return barva;}
    void nastavBarvu(string b) {barva = b;}
    void zjistiDruh() {return druh;}

    protected:
    string druh;
};
```

```
class ctverec : public obrazec
{
private:
public:
int obvod;
int a;
void vypisObvodPredka()
    {
    cout << obrazec::obvod << endl;
    }
}
```

Přístup k atributům předka mimo metodu potomka

```
class obrazec
{
    private:
    string barva;

    public:
    int obvod;
    int obsah;
    obrazec() {}
    ~obrazec() {}
    string zjistiBarvu() {return barva;}
    void nastavBarvu(string b) {barva = b;}
    void zjistiDruh() {return druh;}

    protected:
    string druh;
};
```

```
class obdelnik : public obrazec
{
private:
public:
int obvod;
int a;
int b;
}
```

```
int main(int argc, char *argv[])
{
obdelnik o;
o.obrazec::obvod = 100;
o.obsah = 200; ?? proč takto
}
```

Hierarchie volání konstruktorů v závislosti na vytvoření instancí

```
class obrazec
{
    private:
    string barva;

    public:
    int obvod;
    int obsah;
    obrazec() { cout << "obrazec" << endl;}
    ~obrazec() {}
    string zjistiBarvu() {return barva;}
    void nastavBarvu(string b) {barva = b;}
    void zjistiDruh() {return druh;}

    protected:
    string druh;
};
```

```
class obdelnik : public obrazec
{
private:
public:
obdelnik() {cout << "obdelnik" << endl;}
int obvod;
int a;
int b;
}
```

```
int main(int argc, char *argv[])
{
obdelnik o;
}
```

Vypíše se nejprve obrazec, potom obdelnik, vytváření probíhá hierarchicky od prvního objektu v řadě

Hierarchie volání destruktorů v závislosti na vytvoření instancí

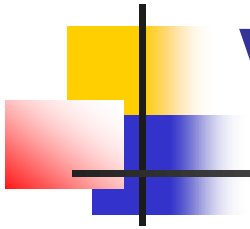
```
class obrazec
{
    private:
    string barva;

    public:
    int obvod;
    int obsah;
    obrazec() { cout << "obrazec" << endl;}
    ~obrazec()
    {cout << "obrazec-konec" << endl;}
    string zjistiBarvu() {return barva;}
    void nastavBarvu(string b) {barva = b;}
    void zjistiDruh() {return druh;}

    protected:
    string druh;
};
```

```
class obdelnik : public obrazec
{
public:
Obdelnik {cout << "obdelnik" << endl;}
~Obdelnik {cout << "obdelnik-konec" << endl;}
int obvod;
int a;
int b;
}
int main(int argc, char *argv[])
{
obdelnik *o = new obdelnik();
delete obdelnik;
}
```

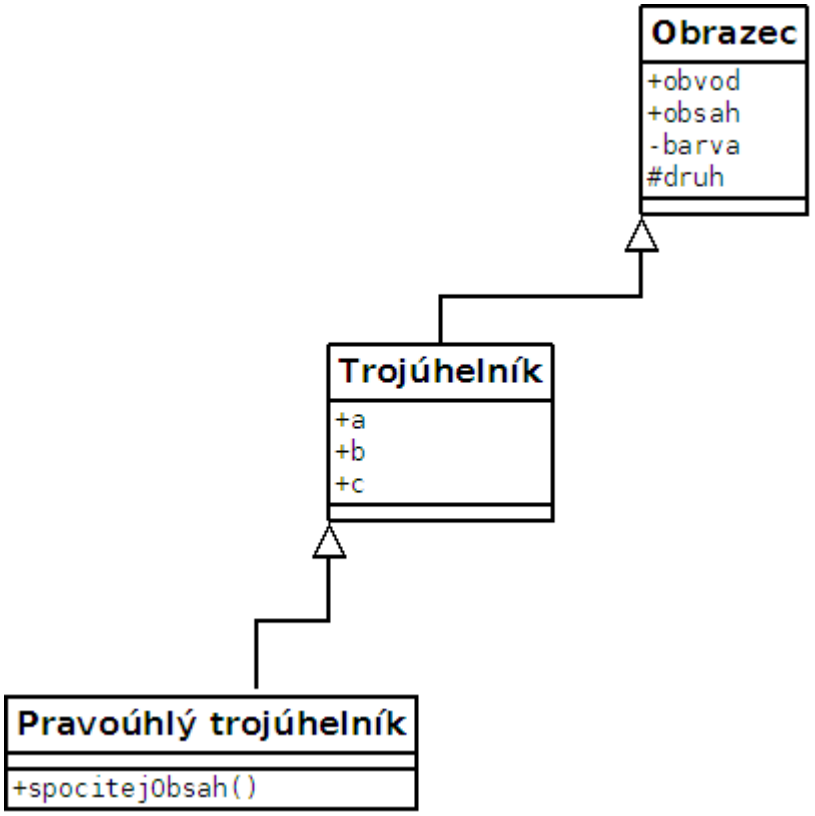
Vypíše se nejprve obdelnik-konec, potom obrazec-konec, rušení probíhá hierarchicky od posledního objektu v řadě



Vícenásobná dědičnost

- Předek může mít neomezené množství potomků
- Každý potomek může upravit přístup k atributům předka

Ukázka vícenásobné dědičnosti



Jak to vypadá s organizací atributů v paměti ?

Ukázka vícenásobné dědičnosti

```
class obrazec
{
    private:
    string barva;

    public:
    int obvod;
    int obsah;
    obrazec() {}
    ~obrazec() {}
    string zjistiBarvu() {return barva;}
    void nastavBarvu(string b) {barva = b;}
    void zjistiDruh() {return druh;}

    protected:
    string druh;
};
```

```
class trojuhelnik : public obrazec
{
private:
// změna atributů public ➔ private
int obrazec::obsah;
public:
// překrytí atributu
int obvod;
// změna atributů protected ➔ public
string obrazec::druh;
int a;
int  b, c;
};
```

```
class pravouhlyTrojuhelnik : public trojuhelnik
{
int spocitejObsah();
};
```

Dědění od více předků najednou

```
class matka
{
public:
string jmenoMatky;
matka()
{ cout << "matka" << endl;}
};
```

```
class otec
{
public:
string jmenoOtce;
otec()
{ cout << "otec" << endl;}
};
```

```
class potomek : public otec, public matka
{
public:
string jmenoPotomka;
};
```

Při volání konstruktoru se vypíše otec, matka

Jak to vypadá s organizací atributů v paměti ?

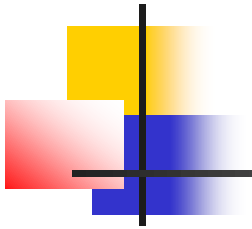
Konflikt jmen při dědění atributů se shodnými názvy

```
class A
{
  public:
   int atribut;
};
```

```
class B
{
   public:
   int atribut;
};
```

```
class  C : public A, public B
{
// chyba
atribut = 10;
// korektní přístup
A::atribut = 10;
B::atribut = 10;
};
```

Friend funkce a třídy (spřátelenost)



- Jedná se o jakousi **specialitu C++**
- **Spřátelený** objekt je funkce či třída, která má jako parametr instanci určité třídy
- Její prototyp (specifikace) musí být uveden v rámci třídy s klíčovým slovem **friend**
- Její implementace je mimo třídu
- **Friend objekt má přístup ke všem atributům a metodám dané třídy, jakoby všechny byly public**

Ukázka friend funkce

```
class trida
{
private:
int a;
int b;
public:
    int c;
    friend void nastavAtributy(trida &t, int a, int
b
};

void nastavAtributy(trida &t, int a, int b)
{
t.a = a; t.b = b;
}
```

```
int main(int argc, char *argv[])
{
trida t;
t.c = 20;
nastavAtributy(t, 10, 20);
}
```