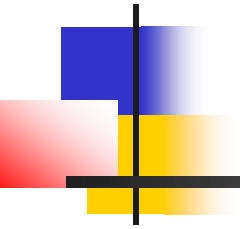


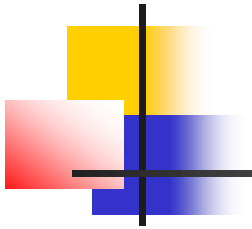
Základy objektového programování v C++

Základy objektově orientovaného programování v C++

Jan Fesl

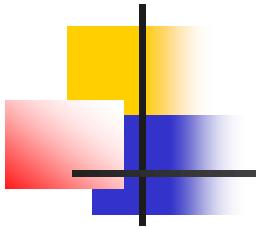
jfesl@prf.jcu.cz

Objektově orientované programování



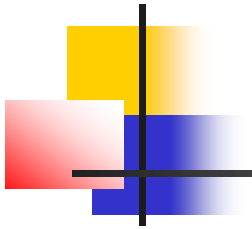
- Snahou je přiblížit programování blíže k reálnému světu
- Mezi objekty existuje hierarchie
  Rodič ➔ Potomek
- Potomci mohou po rodičích přejímat jejich vlastnosti

Objektově orientované programování

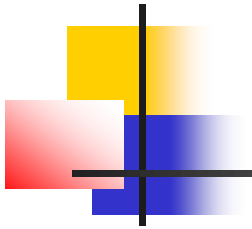


- Objekty obsahují atributy a metody
- Objekty rozšiřují možnosti klasických struktur
- Objekty mohou být jak statické tak dynamické
- Objektový přístup v programování je implementován téměř ve všech moderních programovacích jazycích

Pojmy objektově orientovaného programování

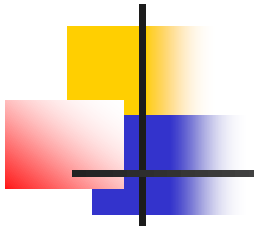


- **Abstrakce**
- **Zapouzdření**
- Dědičnost
- Polymorfismus



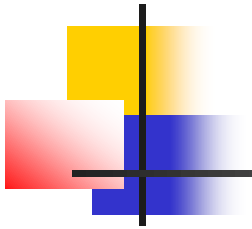
Abstrakce

- Každý objekt je navenek jakousi černou skříňkou, se kterou pracujeme pouze pomocí viditelných metod či atributů
- "nemusíte vědět, jak jsou metody implementovány, ale co dělají"



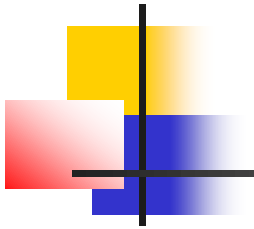
Zapouzdření dat v objektu

- Každý objekt má určité své atributy a metody
- Atributy či metody mohou být navenek viditelné (public) či neviditelné (private)
- K neviditelným atributům či metodám mají přístup pouze metody konkrétního objektu



Dědičnost

- Obdobně jako u biologických organismů, kde se předává genová informace z předka na potomka
- Dědičnost určitých atributů je možné zakázat či omezit



Objekty v C++

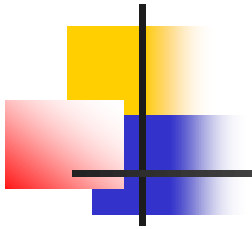
- Pro deklaraci objektu se používá klíčové slovo class
- Pro deklaraci atributů se používají pro private a public sekce

Deklarace objektu v C++

- class nazevTridy
  {
  // sekce privátních atributů a funkcí
  private:
  // sekce veřejných atributů a funkcí
  public:
  // sekce privátních či veřejných atributů a funkcí
  protected:
  };

Sekce protected má svůj význam až v době, kdy dochází k dědění mezi objekty – vysvětlíme později

Abstakce návrhu objektového typu

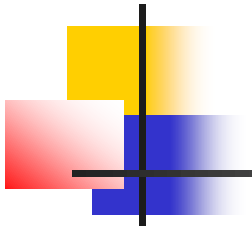


- Uvažujme určitý geometrický obrazec
- Tento obrazec má **atributy**

  a) určitou barvu

  b) kvantitativní atribut (poloměr, délku strany)
- A **metodu**, která jej vykreslí

Ukázka deklarace třídy Obrazec



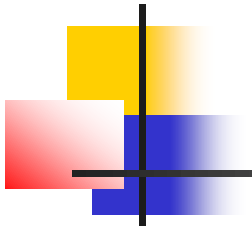
- class Obrazec

```
{
private:
public:
int barva, delkaStrany;
string druhObrazce;
void nakresliObrazec();
};
```

Deklarace metody ve třídě

```
class Obrazec
        {
        private:
        public:
        int barva, delkaStrany;
        string druhObrazce;
        void nakresliObrazec()
              {
               /* Kód metody */
              }
        };
```

Deklarace třídní metody mimo třídu

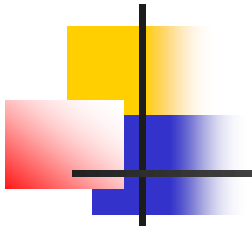


- druhNavratoveHodnotyMetody nazevTridy::nazevMetody(parametry);

Ukázka deklarace třídní metody mimo třídu

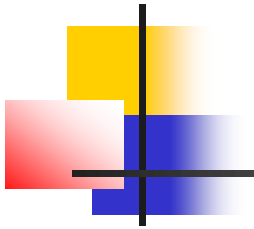
```
class Obrazec
        {
        private:
        public:
        int barva, delkaStrany;
        string druhObrazce;
        void nakresliObrazec();
};

void Obrazec::nakresliObrazec()
{
    cout << "Kreslím obrazec.. " << "\n";
}
```



Instance třídy

- Každá deklarovaná třída je pouze jakási šablona
- Reálná existence objektu je instance třídy
- Je to obdobné jako u dynamicky vázaných proměnných



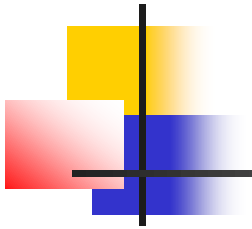
Deklarace instance třídy

- // statická deklarace instance třídy Obrazec
- Obrazec obrazec;

- // dynamická deklarace instance třídy Obrazec

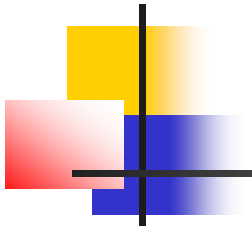
- Obrazec *pObrazec = new Obrazec ();

Dynamicky a staticky vázané atributy a metody



- Staticky vázané metody či atributy nejsou vázány na vytvoření konkrétní instance
- Jsou umístěny v paměti na překladačem definovaném místě
- Vzhledem umístění je volání statických metod rychlejší

Deklarace staticky vázaného atributu a metody



- Je shodná s běžnou deklarací s tím rozdílem, že se před typ atributu či metody uvádí klíčové slovo static

- static int a;
- static int pokus();

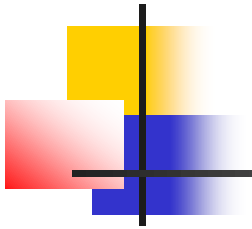
Deklarace staticky vázané proměnné

- Proměnná, která je **statická** se musí explicitně deklarovat (aby jí byla přidělena paměť)

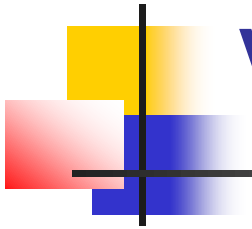
- // deklarace tridy
  class A
  {
  static int atribut;
  };

```
// alokace paměti pro proměnnou (jakoby deklarace)
int A::atribut;
// přístup k hodnotě jako k tridni promenne
cout << A::atribut << endl;
A a;
cout << a.atribut << endl;
```

Omezení staticky vázaných metod



- Mohou využívat pouze atributy a metody které jsou statické
- Dynamicky vázané metody mohou využívat obojí

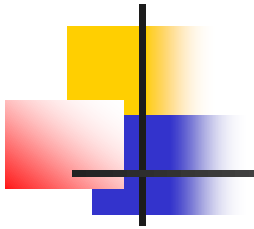


Volání staticky vázané metody

- Je možné i bez vytvoření konkrétní instance

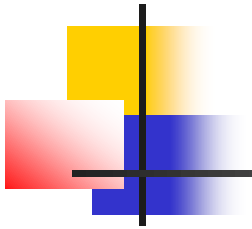
- Deklarace

  Název_třídy::metoda(parametry);



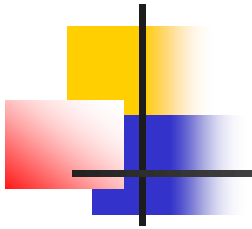
Konstruktor třídy

- Je metoda (funkce), která se volá ihned po vytvoření objektu
- Používá se pro nastavení výchozích hodnot atributů
- Tuto funkci lze deklarovat vícenásobně s různými parametry (přetěžování)



Destruktor třídy

- Je funkce která se volá těsně před zánikem určité třídy
- Zpravidla dochází v jeho těle k uvolnění alokované paměti pro dynamicky vázané proměnné

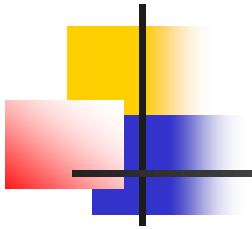


Deklarace konstruktoru

- Konstruktor je metoda (funkce) která má stejný název jako třída, jejíž je konstruktorem
- Bývá obvyklé deklarován ve veřejné sekci – public
- Volání konstruktoru je možné stejně jako jiné metody, ale není to obvyklé

Ukázka deklarace konstruktoru pro třídu Obrazec

- class Obrazec
  {
  private:
  public:
  int barva, delkaStrany;
  string druhObrazce;
  Obracec();
  Obrazec(int barvaObrazce, int delHrany);
  };



Deklarace destruktoru

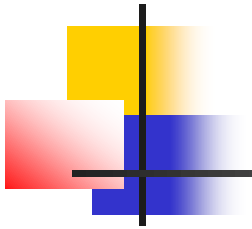
- Používá se obdoba deklarace konstruktoru – název třídy, před který se uvede ~
- Volání destruktoru je možné stejně jako jiné metody, ale není to obvyklé

Ukázka deklarace destruktoru

- class Obrazec

```
{
private:
public:
int barva, delkaStrany;
string druhObrazce;
// konstruktor
Obracec();
// destruktor
~Obrazec();
Obrazec(int barvaObrazce, int delHrany);
};
```

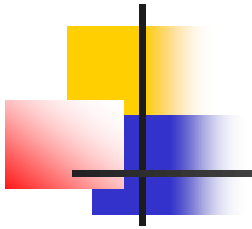
Inicializace atributů objektu pomocí konstruktoru



- Pro atributy deklarované ve třídě je možné definovat jejich implicitní (počáteční hodnoty)
- Hodnoty atributů se uvedou za : do závorek před deklarací metody ve třídě
- Inicializovat lze libovolné množství atributů

Ukázka incializace atributů v konstruktoru

- class Obrazec
  {
  private:
  public:
  int barva, delkaStrany;
  string druhObrazce;
  Obracec() : barva(0), delkaStrany(0);
  Obrazec(int barvaObrazce, int delHrany);
  };



Implicitní konstruktor

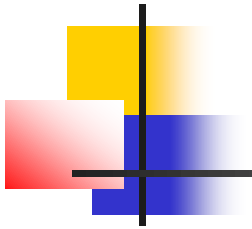
- Konstruktor, který se volá při vytvoření instace, pokud konstruktor nespecifikujeme přímo v závislosti na parametrech
- Běžně bývá implictním konstruktorem nazýván konstruktor bez parametrů ()

Deklarace implicitního konstruktoru

- class Obrazec

```
{
private:
public:
int barva, delkaStrany;
string druhObrazce;
// implicitní konstruktor
Obracec() {}
// destruktor
~Obrazec() {}
Obrazec(int barvaObrazce, int delHrany) {} ;
};
```

Implicitní konstruktor s parametry



- Pokud hodnoty parametrů v konstruktoru definujeme přímo, použije daný konstruktor v případě potřeby překladač jako implicitní

Deklarace implicitního konstruktoru s parametry

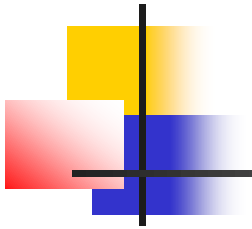
```
class Obrazec
        {
        private:
        public:
        int barva, delkaStrany;
        string druhObrazce;
        // destruktor
        ~Obrazec() {}
  // implicitní konstruktor
        Obrazec(int barvaObrazce=0, int delHrany) {}
        };
```

Kopírování pomocí konstruktoru

- Přenesení hodnot atributů jedné instance do druhé

```
Obrazec a1;
a1.barva = 2; a1.delkaStrany = 3;
Obrazec a2(a1);
cout << a2.barva << "\t" << a2.delkaStrany;
```

Problémy při kopírování atributů typu pole



- V případě, že je atributem objektu pole, vznikne při kopírování pomocí konstruktoru pouze kopie vlastního ukazatele na pole, nikoli však prvků pole.

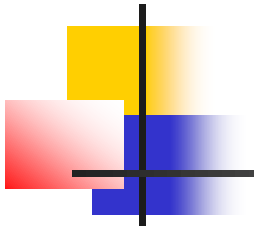
➔ mělká kopie

Kopie konstruktorem v objektu s polem

```
class A
{
private:
public:
int *pole;
A() {
     pole  = new int[10];
     for (int i=0; i < 10; i++)
          pole[i] = i;
     }
}

A a1;
A a2(a1);
```

a2.pole[0] = 5; → prepise hodnotu i v poli instance a1



Řešení ?

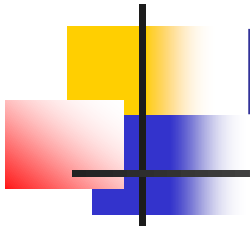
- Kopírovací konstruktor
- Konstruktor, který má jako parametr objekt stejného typu

Implementace kopírovacího konstruktoru

```
class A
  {
  private:
  public:
  int *pole;
  A() {
      pole  = new int[10];
      for (int i=0; i < 10; i++)
          pole[i] = i;
      }
  A(A &a)
   {
   pole  = new int[10];
   for (int i=0; i < 10; i++)
       {
       pole[i] = a.pole[i];
       }
   }
  }

A a1;
A a2(a1);
```

a2.pole[0] = 5; → neprepise hodnotu v poli instance a1



Hluboká kopie

- V předchozím řešení byl deklarován kopírovací konstruktor, který realokoval pro pole nové instance, nové místo v paměti
- Alokace nového místa v paměti a naplněním hodnot původní pole vytváří tzv. hlubokou kopii instance