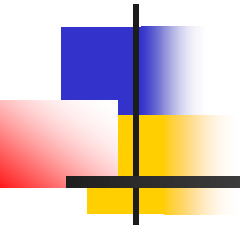
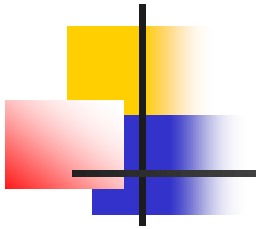


Programování v C++ I

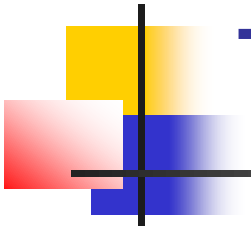
Virtuální metody, polymorfismus, abstraktní třídy, virtuální dědění, polymorfní struktury

Jan Fesl
jfesl@prf.jcu.cz



Virtuální metody

- Virtuální metoda je metoda, která se volá v závislosti na typu instance
- Je to metoda některého potomka v hierarchii, která jakoby zakrývá metodu předka
- Rozdíl mezi překrytou metodou a virtuální metodou je ve způsobu jejího volání
- Virtuální funkci označujeme klíčovým slovem **virtual**
- Pokud je některá metoda deklarována v předkovi jako virtuální, je virtuální i v potomkovi, v potomkovi se již slovo virtual nemusí uvádět
- Tyto metody jsou základem **polymorfismu**, což je jakési jejich používání, kdy potomek zastupuje předka, ačkoli se na instanci potomka odkazujeme pomocí ukazatele typu ukazatel na předka
- Implementaci tohoto mechanismu používají tzv. **polymorfní datové struktury** (**neplést se struct**)



Tabulka virtuálních metod

- Jedná se o místo v paměti, kde jsou uvedené veškeré virtuální metody
- V závisloti na typu instance objektu se vybírá pomocí této tabulky implementace (kód) konkrétní metody
- Virtuální metody jsou díky tomuto mechanismu pomalejší než klasické
- Virtuální metody jsou použitelné zásadně u ukazatelů či referencí u objektových typů

Ukázka použití virtuální metody

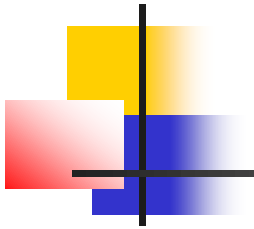
```
class obrazec
 {
 public:
 virtual void kresli();
 { cout << „kreslim obrazec" << endl; }
 };
```

```
class obdelnik : public obrazec
 {
 public:
 void kresli();
 { cout << „kreslim obdelnik" << endl; }
 };
```

```
class ctverec : public obrazec
 {
 public:
 void kresli();
 { cout << „kreslim ctverec" << endl; }
 };
```

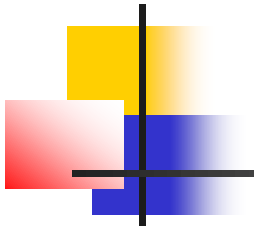
```
int main()
{
obrazec *o = new obrazec();
obrazec *o1 = new obdelnik();
obrazec *o2 = new ctverec();
o->kresli(); o1->kresli(); o2->kresli();
// vypise se
„kreslim obrazec”„ kreslim obdelnik”
 “kreslim ctverec”
}
```

Způsoby volání metod časná a pozdní vazba



- **Časná vazba** znamená určení typu proměnné v době překladu programu
- **Pozdní vazba** znamená určení typu proměnné v době běhu programu (typické pro virtuální metody)

Čistě virtuální metoda, abstraktní třída



- Čistě virtuální metoda, je virtuální metoda která ve třídě nemá implementaci, neboť bude implementována až v některém z jejích potomků
- Třída, která obsahuje alespoň jednu čistě virtuální metodu se nazývá abstraktní
- **Od abstraktní třídy není možné vytvořit instanci**
- **Nicméně je možné vytvořit ukazatel na nějaký objekt, který je typu abstraktní třída**

Čistě virtuální metoda, abstraktní třída

```
class tridaAbst
{
Public:
virtual void metodaAbs() = 0;
};

// toto je korektní
tridaAbst *A;

// toto je chyba
A = new tridaAbst();
```

metodaAbs je čistě virtuální

Díky metodě metodaAbst je tridaAbst abstraktní třídou

Virtuální dědění – proč ?

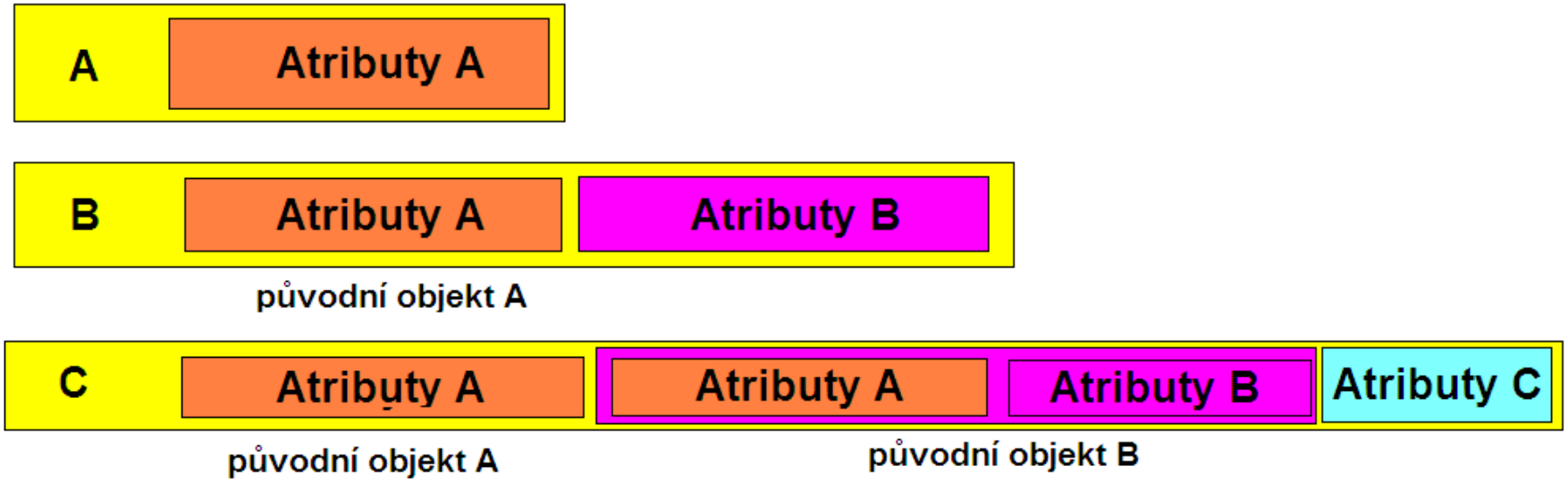
```
class A
{
  public:
   int a;
};
```

```
class B : public A
{
    public:
   int b;
};
```

```
class  C : public A, public B
{
   // problém
   // jelikož B je již potomkem A
    A::a = 10;
};
```

Co s tím ??

Nevirtuální dědění od více předků – objekt C



Je zřejmé, že u klasického dědění od vznikne konflikt jmen v objektu C kvůli atributům A, A::atribut může být C jak od A tak i od B….

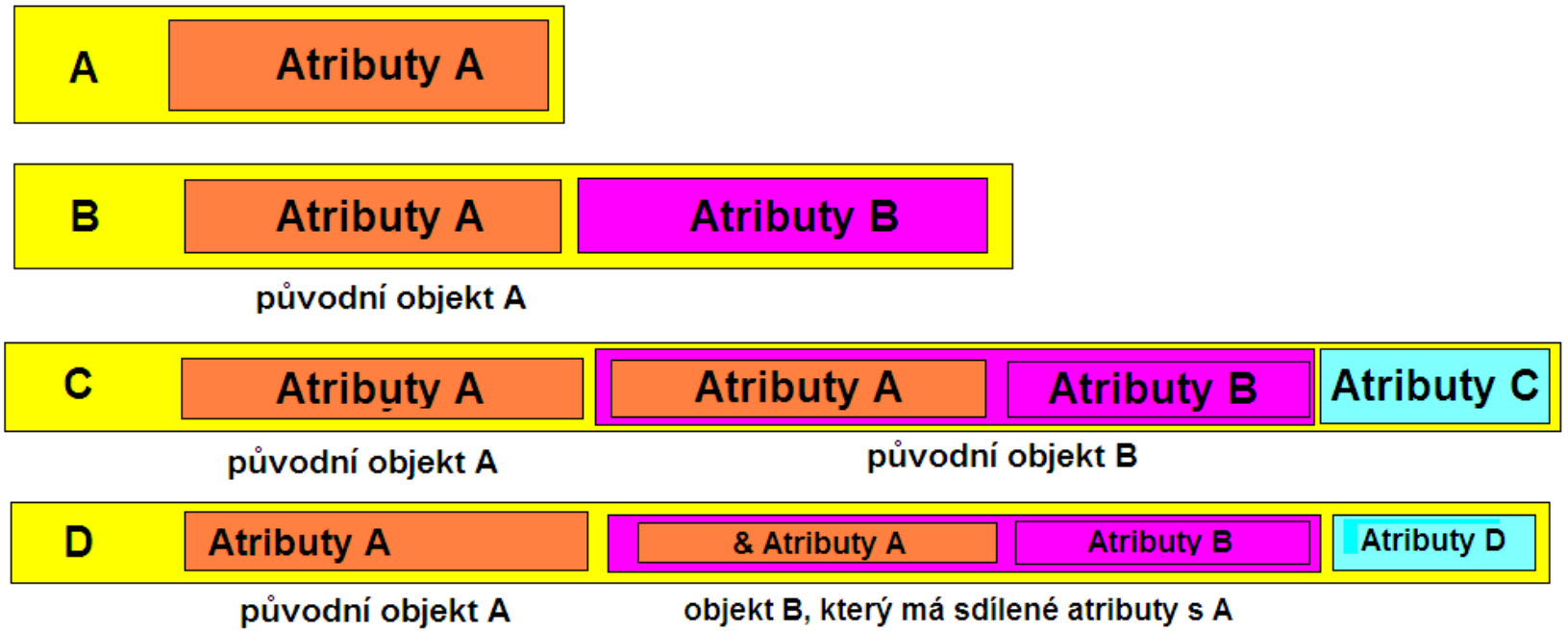
Virtuální dědění - implementace

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

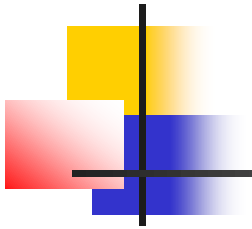
```
class B : virtual public A
{
   public:
   int b;
};
```

```
class  C : public virtual A, virtual B
{
   // teď je vše v pořádku
    A::a = 10;
};
```

Virtuální dědění – objekt D

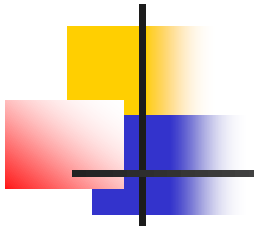


Hlavní idea za virtuálním děděním tkví v tom, že každý atribut či metoda je děděna pouze jednou a sdílí je všichni předkové a potomci mezi sebou.



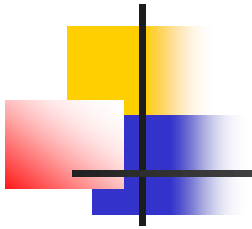
Virtutální dědění - syntaxe

- Používá se obdobně jako u metod klíčové slovo virtual
- Je výhodné tam, kde může nastat konflikt jmen
- Přístup k virtuálně zděděným atributům je ovšem pomalejší než ke staticky zděděným



Polymorfní struktury

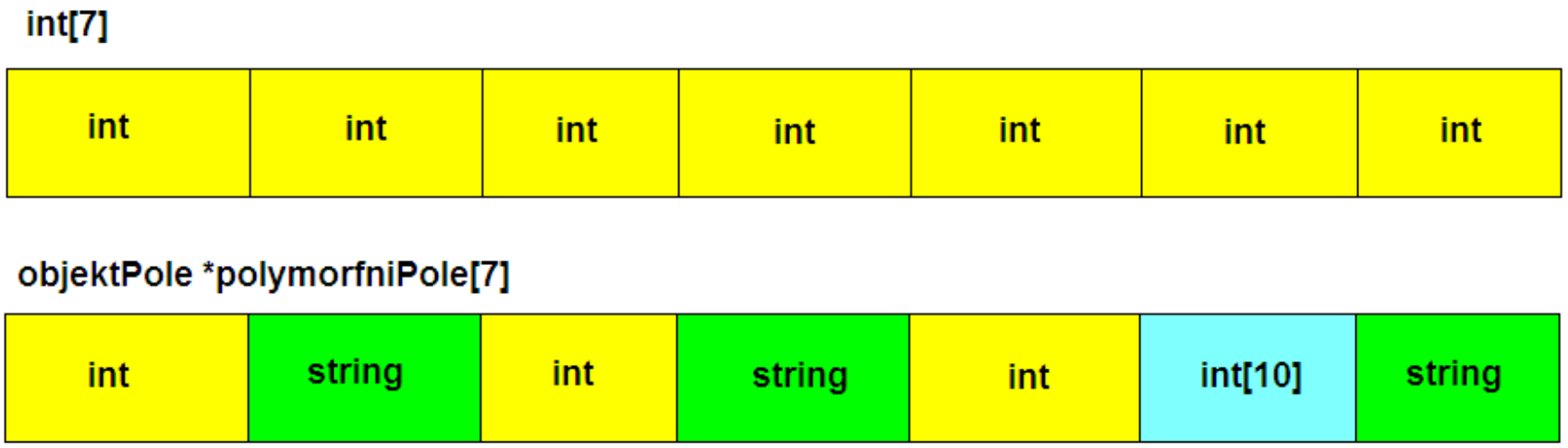
- Obecně se jedná o jakési seskupení objektů, které mají stejného předka, který definuje společné metody či atributy
- Ovšem vlastní implementace virtuálních metod je již v režii potomka
- Uvažume příklad polymorfního pole



Polymorfní pole - specifikace

- U klasického pole platí to, že se jedná o homogenní datovou strukturu
- Polymorfní pole je sice také homogenní datová struktura z pohledu typu předka
- Nicméně vlastní hodnoty prvků pole jsou různé

Polymorfní pole - ukázka



Polymorfní pole - implementace

```
class objektPole
{
public:
virtual void vypisPrvek() = 0;
};
```

Abstraktní třída – předek

```
class objekPoleInt : public objektPole
{
int hodnota;
public:
objektPoleString(int iHodnota)
{hodnota = iHodnota;}
void vypisPrvek() {cout << hodnota}
};
```

```
class objekPoleString : public objektPole
{
string hodnota;
public:
objektPoleString(string sHodnota) {hodnota = sHodnota;}
void vypisPrvek() {cout << hodnota}
};
```

2 různé prvky pole

Polymorfní pole - implementace

```
class objektPoleIntVPoli : public objektPole
{
string hodnota;
int *hodnotyPole; int velikostPole;
public:
objektPoleIntVPoli(int *pole, int velikost)
{
hodnotyPole = new int[velikost];
int velikostPole = velikost;
for (int i=0; i < velikost; i++)  hodnotyPole[i] = pole[i];
}
void vypisPrvek()
{ for (int i=0; i < velikost; i++) cout << hodnotyPole[i] << endl;}
};
```

Polymorfní pole - implementace

```
typedef prvekPole *pPrvekPole;

class polymorfniPole
{
int velikost;
objektPole **prvky;

public:
polymorniPole(int velikost)
{
prvky = new objektPole*[10]
}

void vlozPrvek(objektPole *prvek, int pozice);

void vypisObsazePrvky();
}
```

Implementace není kompletní – pouze schématicky (viz moodle)

Vytvoření a volání polymorfního pole

```
int main(int argc, char **argv)
{
        objektPoleInt       *prvekInt     = new objektPoleInt(10);
objektPoleString    *prvekString1 = new objektPoleString("abc");
objektPoleString    *prvekString2 = new objektPoleString("def");
int pole[4] =  {1,2,3,4};
objektPoleIntVPoli *poleIntVPoli = new objektPoleIntVPoli(4,pole);
polymorfniPole *pPole = new polymorfniPole(100);
pPole->vlozPrvek(prvekInt,0);
pPole->vlozPrvek(prvekString1,1);
pPole->vlozPrvek(prvekString2,2);
pPole->vlozPrvek(poleIntVPoli,3);
// vlastni vypsani prvku
pPole->vypisObsazenePrvky();
return 0;
}
```