Y36PJC Programování v jazyce C/C++

Dědění, polymorfismus

Ladislav Vagner

Dnešní přednáška

- Dědění.
- Polymorfismus.
- Statická a dynamická vazba.
- Vnitřní reprezentace.
- VMT – tabulka virtuálních metod.
- Časté chyby.

Minulá přednáška

- Přetěžování operátorů funkcemi.
- Přetěžování operátorů metodami.
- Operátor =.
- Friend funkce.

Dědění - příklad

- Modelujeme zaměstnance a vedoucí:
  - mají jméno,
  - hodinovou mzdu,
  - o vedoucích navíc ještě známe počet podřízených.
- Chceme rozhraní, které umožní výpočet měsíční mzdy, známe-li počet odpracovaných hodin v měsíci:
  - pro zaměstnance je to prostý součin s hodinovou mzdou,
  - vedoucímu navíc přísluší měsíční příplatek za vedení ve výši 500,- Kč za řízeného zaměstnace.

Dědění - příklad

- Zavedeme třídy **`CEmployee`** a **`CBoss`**.
- Třída **`CBoss`** má mnoho společných vlastností se třídou **`CEmployee`** – bude jejím potomkem.
- Ve třídě **`CEmployee`** budou členské proměnné **`name`** a **`salary`**.
- Ve třídě **`CBoss`** bude navíc ještě členská proměnná **`subordNr`** udávající počet podřízených.
- Metody pro výpočet platu budou pro každou třídu jiné, ale budou mít stejnou signaturu. Metoda třídy **`CBoss`** může využít metodu předka **`CEmployee`** jako krok ve výpočtu.

Dědění - příklad

```
class CEmployee
 {
    string name;
    int    salary;
  public:
    CEmployee ( string _name, int _salary ) :
                name (_name), salary (_salary) {}
    int MonthSalary ( int Hours ) const
     { return salary * Hours; }
 };
```

Dědění - příklad

```
class CBoss : public CEmployee
 {
    int subordNr;
   public:
    CBoss ( string _name, int _salary, int _subord ):
           CEmployee ( _name, _salary ),
           subordNr ( _subord ) { }
   int MonthSalary ( int Hours ) const
    { return CEmployee::MonthSalary ( Hours ) +
             subordNr * 500; }
 };

CEmployee x ( "Novotny", 100 );
CBoss     y ( "Novak", 200, 2 );

cout << x . MonthSalary ( 170 );
cout << y . MonthSalary ( 170 );
```

Dědění

- Vztah mezi třídami:
  - členské proměnné a metody definované v předku mohou být použity i potomkem,
  - potomek může přidat nové členské proměnné a metody,
  - potomek může změnit definice metod předka.
- Má význam tehdy, pokud se struktura předka a potomka liší jen málo.
- V C++ lze dědit vícenásobně (z více předků).
- V C++ neexistují rozhraní (interfaces) z Javy.

Polymorfismus

- Situace, kde různé objekty:
  - rozumí stejné zprávě (mají metodu se stejnou signaturou),
  - ale na zprávu reagují různě (vyvoláním jiného kódu).
- Aby měl polymorfismus praktický význam, musí mít provedený kód stejný význam (v kontextu daného objektu).
- V C++, Javě omezen - polymorfismus existuje jen mezi objekty, jejichž třídy jsou ve vztahu předek/potomek.

Dědění

- Syntaxe zápisu – předek je oddělen dvojtečkou.
- Dědění **`public`** – viditelnost zděděných metod se nemění.
- Dědění **`private`** – zděděné metody a členské proměnné budou všechny **`private`** (zachová dědění, potlačí polymorfismus).

```
class CEmployee
 {
   …
 };

class CBoss : public CEmployee
 {
   …
 };
```

Dědění

- Zapouzdření:
  - **`private`** – viditelné pouze pro danou třídu.
  - **`protected`** – viditelné pro danou třídu a všechny její potomky,
  - **`public`** – viditelné pro všechny.
- Viditelnost **`protected`** – rozumný kompromis, pokud navrhujeme předka a nevíme, kdo z něj bude dědit.
- Vyhněte se bezmyšlenkovité záplavě metod typu getter/setter pro každou členskou proměnnou:
  - samotné gettry/settry nic nedělají, jen stojí režii,
  - existují-li pro každou členskou proměnnou, jsou vlastně popřením zapouzdření.

Dědění

- Při vytváření instance se volají postupně všechny konstruktory směrem od předků k potomkům.
- Není-li určeno, který konstruktor předka se volá, bude zvolen implicitní konstruktor.
- Neexistuje-li v takové situaci implicitní konstruktor předka, dojde k chybě při překladu.
- Pravidla volby konstruktoru jsou stejná jako v případě staticky alokovaných členských proměnných.
- Jak probíhá volání kostruktorů předků v Javě?

Dědění

```
class CBoss : public CEmployee
 {
   …
   CBoss ( string _name, int _salary, int _subord )
           { name = _name; …  }    // !! chyba
   CBoss ( const CBoss & x ) { … } // !! chyba
 };

class CBoss : public CEmployee
 {
   …
   CBoss ( string _name, int _salary, int _subord ):
           CEmployee ( _name, _salary ),
           subordNr ( _subord ) { }
   CBoss ( const CBoss & x ) : CEmployee ( x ) { … }
 };
```

Dědění

- Při uvolňování instance se volají postupně všechny destruktory od potomků směrem k předkům.
- Protože destruktor je pouze jeden, nemůže dojít k nejednoznačnostem.

```
class A
 {
   A  ( int x )                  { cout << "A::A"; }
   A  ( const A & )              { cout << "A::cA"; }
   ~A ( void )                   { cout << "A::~A"; }
 };
class B : public A
 {
   B ( int x ) : A ( x )         { cout << "B::B"; }
   B ( const B & x ) : A ( x )   { cout << "B::cB"; }
   ~B ( void )                   { cout << "B::~B"; }
 };
```

Dědění

```
void foo1 ( A param ) { ... }
void foo2 ( A & param ) { ... }
void foo3 ( A * param ) { ... }
void foo ( void )
 {
   A   a(10);     // A::A
   B   b(20);     // A::A, B::B

   foo1 ( a );    // A::cA, A::~A
   foo2 ( a );    // nic
   foo3 ( a );    // nic

   foo1 ( b );    // A::cA, A::~A
   foo2 ( b );    // nic
   foo3 ( b );    // nic
 }                // B::~B, A::~A, A::~A
```

Dědění

- Překrytí metody – v potomkovi je definovaná metoda se stejnou signaturou, ale jiným tělem.
- Metodu předka lze volat s použitím čtyřtečkové notace.

```
class CEmployee
 { … int MonthSalary ( int Hours ) const {…} … };

class CBoss : public CEmployee
 {
   …
   int MonthSalary ( int Hours ) const
    {
      return CEmployee::MonthSalary ( Hours ) +
             subordNr * 500;
    }
 };
```

Dědění

- Kompatibilita vzhledem k přiřazení:
  - instanci potomka lze použít na místě typově odpovídajícím předku,
  - instanci předka nelze použít na místě typově odpovídajícím potomku.

```
class CEmployee { … };
class CBoss : public CEmployee { … };
CEmployee a, * aptr;
CBoss     b, * bptr;
a    = b;              // ok
aptr = bptr;           // ok
CEmployee & aref = b; // ok
b    = a;              // !!!
bptr = aptr;           // !!!
CBoss     & bref = a; // !!!
```

Dědění a polymorfismus

- Objekty **`CBoss`** i **`CEmployee`** rozumí zprávě **`MonthSalary`**:
  - výpočet platu probíhá v obou třídách jinak,
  - na vyšší úrovni umožní pracovat s výpočtem platu, bez nutnosti starat se o implementační detaily.
- Snazší údržba – výpočet platu je na jednom místě programu.
- Snazší rozšiřitelnost – nová třída (např. **`CManager`** s ještě jinými pravidly výpočtu platu) nenutí přepisovat zbytek programu.

Dědění a polymorfismus

```
CEmployee * dept [3];
int i;

dept[0] = new CBoss ( "Novak", 200, 2 );
dept[1] = new CEmployee ( "Novotny", 100 );
dept[2] = new CEmployee ( "Novotna", 100 );

for ( i = 0; i < 3; i ++ )
 cout << i << ". " << dept[i] -> MonthSalary ( 170 );

for ( i = 0; i < 3; i ++ )
 delete dept[i];

0.  34000    // !! ma byt 35000
1.  17000
2.  17000
```

Dědění a polymorfismus

- Statická vazba:
  - volaná metoda je určená v době kompilace,
  - rozhoduje datový typ proměnné, kterou je instance zpřístupněna,
  - volání metody je trochu rychlejší.
- Dynamická vazba:
  - volaná metoda se určí v době běhu,
  - rozhoduje datový typ instance, se kterou se pracuje,
  - volání je trochu pomalejší.
- C++ umí obě vazby, výchozí je statická.
- Dynamická vazba se vynutí klíčovým slovem **`virtual`** před deklarací metody.
- Jakou vazbu umí Java ?

Dědění a polymorfismus

```
class CEmployee
 {
   …
   virtual int MonthSalary ( int Hours ) const { … }
   …
 };

class CBoss : public CEmployee
 {
   …
   virtual int MonthSalary ( int Hours ) const { … }
   // zde jiz virtual byt nemusi, bude zdedeno
   // je ale vhodne (lepsi citelnost)
   …
 };
```

Dědění a polymorfismus

```
CEmployee * dept [3];
int i;

dept[0] = new CBoss ( "Novak", 200, 2 );
dept[1] = new CEmployee ( "Novotny", 100 );
dept[2] = new CEmployee ( "Novotna", 100 );

for ( i = 0; i < 3; i ++ )
 cout << i << ". " << dept[i].MonthSalary ( 170 );

for ( i = 0; i < 3; i ++ )
 delete dept[i];

0.  35000     // OK
1.  17000
2.  17000
```

Dědění a polymorfismus

- Dynamická vazba se uplatní:
  - metoda je označena **`virtual`**,
  - pracujeme s ní pomocí ukazatele nebo reference.

```
CBoss a ( "Novak", 200, 2 );
cout << a . MonthSalary ( 170 );   // 35000

CEmployee b = a;
cout << b . MonthSalary ( 170 );   // 34000

CEmployee * c = &a;
cout << c -> MonthSalary ( 170 );  // 35000

CEmployee & d = a;
cout << d . MonthSalary ( 170 );   // 35000
```

Dědění a polymorfismus

```
class X { void foo () {…} };
class Y : public X { void foo () {…} };

X a, *aptr = &a, & aref = a;
Y b, *bptr = &b, & bref = b;
a . foo ();       // X :: foo
b . foo ();       // Y :: foo
a = b;
a . foo ();       // X :: foo
aptr -> foo ();   // X :: foo
bptr -> foo ();   // Y :: foo
aptr = bptr;
aptr -> foo ();   // X :: foo
aref . foo ();    // X :: foo
bref . foo ();    // Y :: foo
X & cref = b;
cref . foo ();    // X :: foo
```

Dědění a polymorfismus

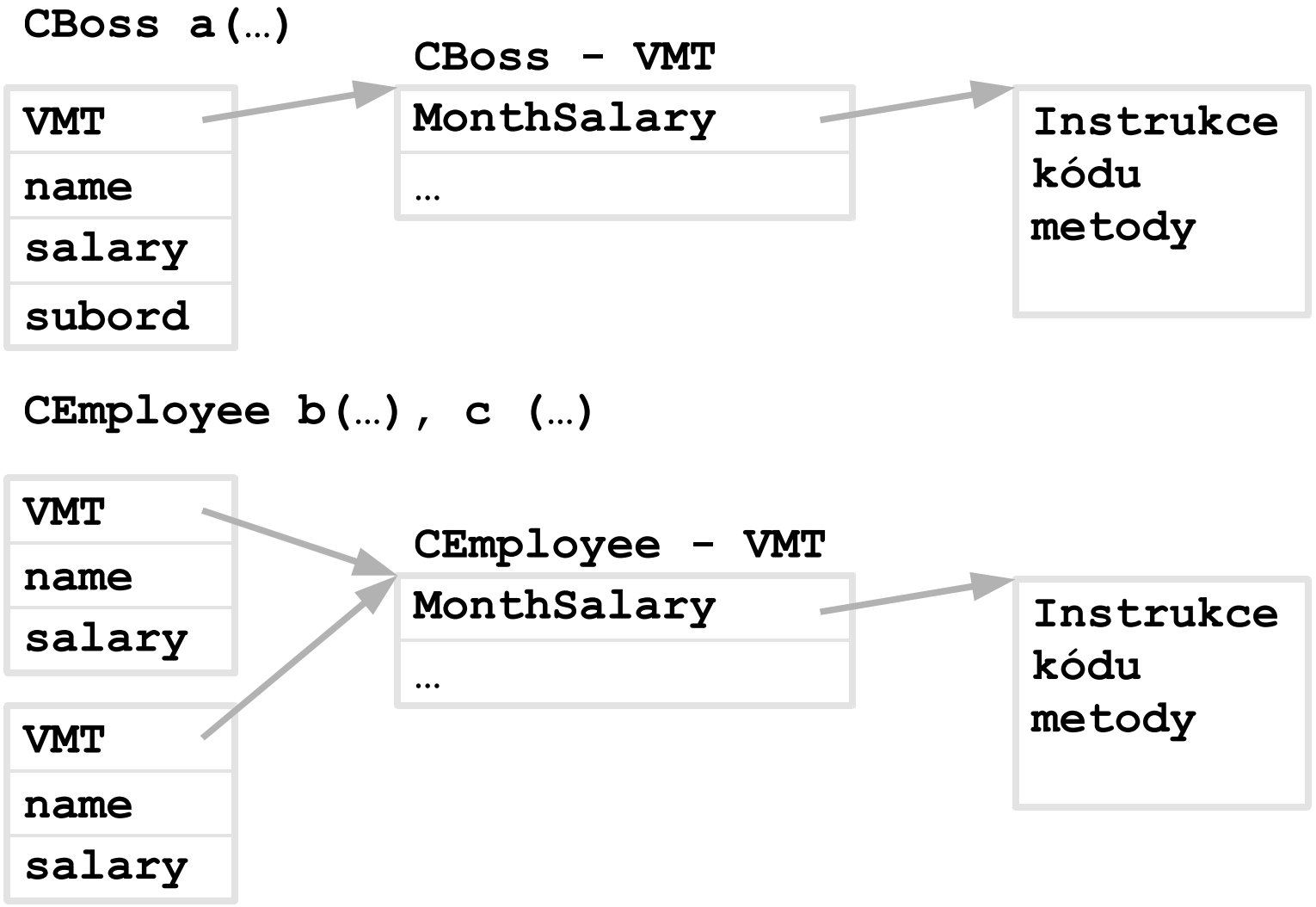
```
class X { virtual void foo () {…} };
class Y : public X { virtual void foo () {…} };

X a, *aptr = &a, & aref = a;
Y b, *bptr = &b, & bref = b;
a . foo ();       // X :: foo
b . foo ();       // Y :: foo
a = b;
a . foo ();       // !! X :: foo i přes virtual
aptr -> foo ();   // X :: foo
bptr -> foo ();   // Y :: foo
aptr = bptr;
aptr -> foo ();   // Y :: foo
aref . foo ();    // X :: foo
bref . foo ();    // Y :: foo
X & cref = b;
cref . foo ();    // Y :: foo
```

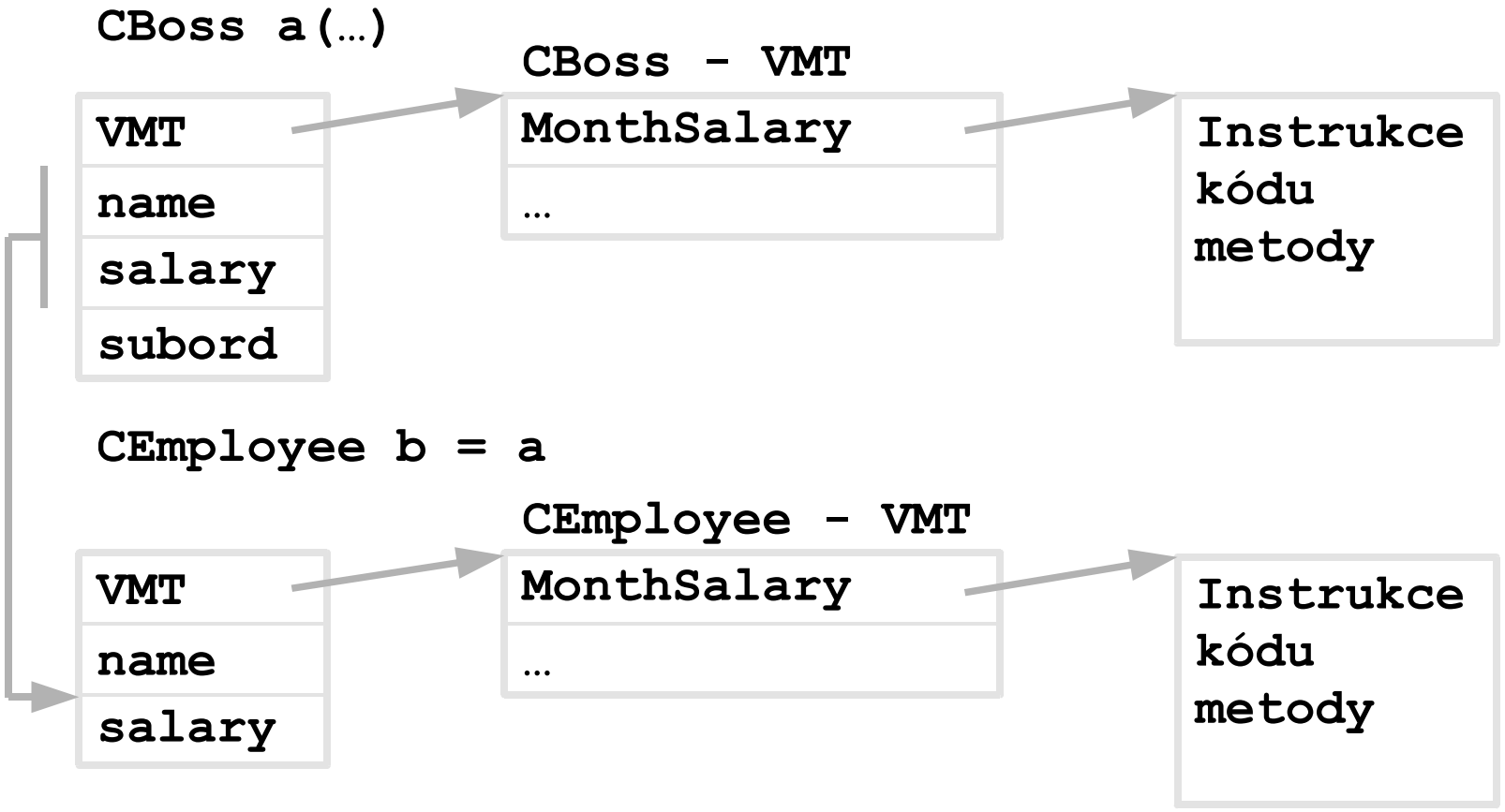
Dědění a polymorfismus

- Vnitřní realizace dynamické vazby:
  - objekt obsahuje (typicky na prvním místě) odkaz na tabulku VMT (Virtual Method Table),
  - Odkaz na VMT vyplněn v konstruktoru, pak se již nemění,
  - do VMT jsou umístěny odkazy (ukazatele) na adresy metod, které jsou **`virtual`**,
  - pozice metody ve VMT se zachovává při dědění.
- Volání virtual metody:
  - podle názvu metody se určí pozice ve VMT,
  - z VMT se vybere adresa, kam se předá řízení.

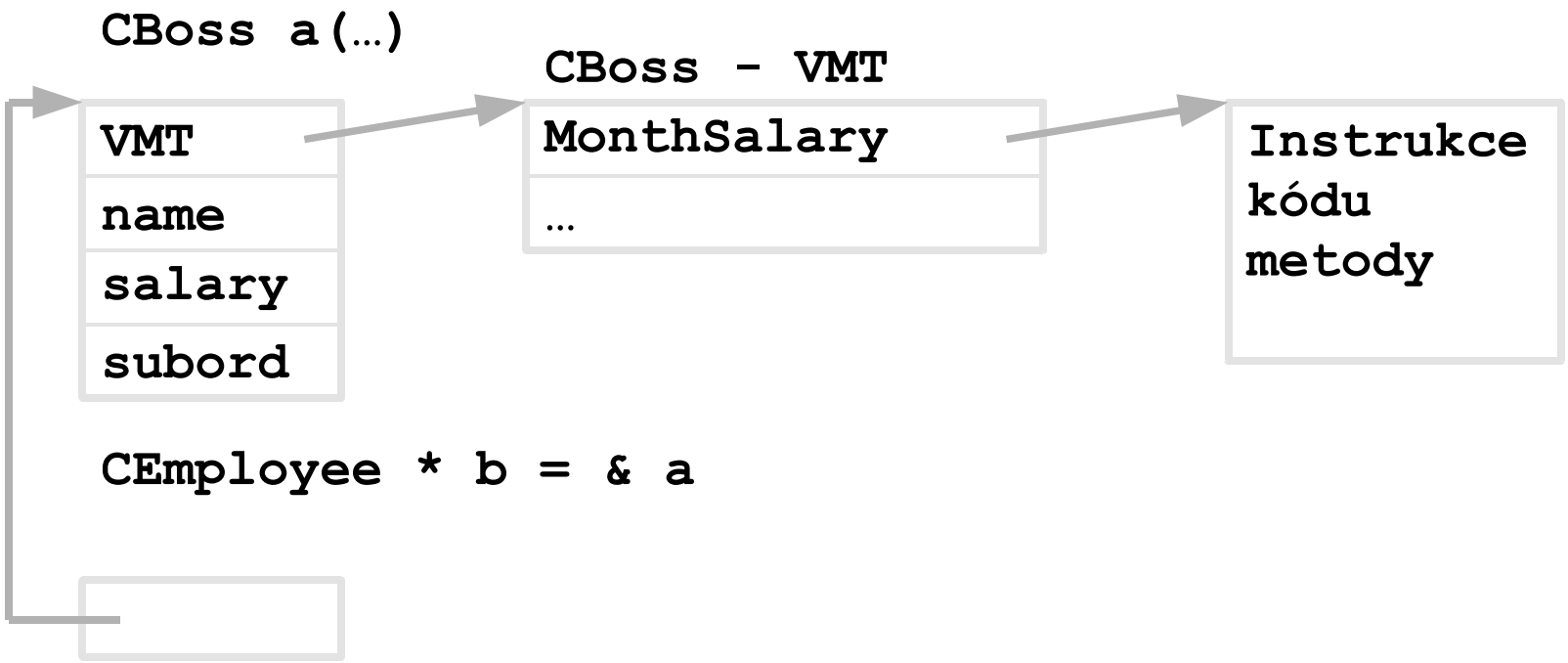
Dědění a polymorfismus



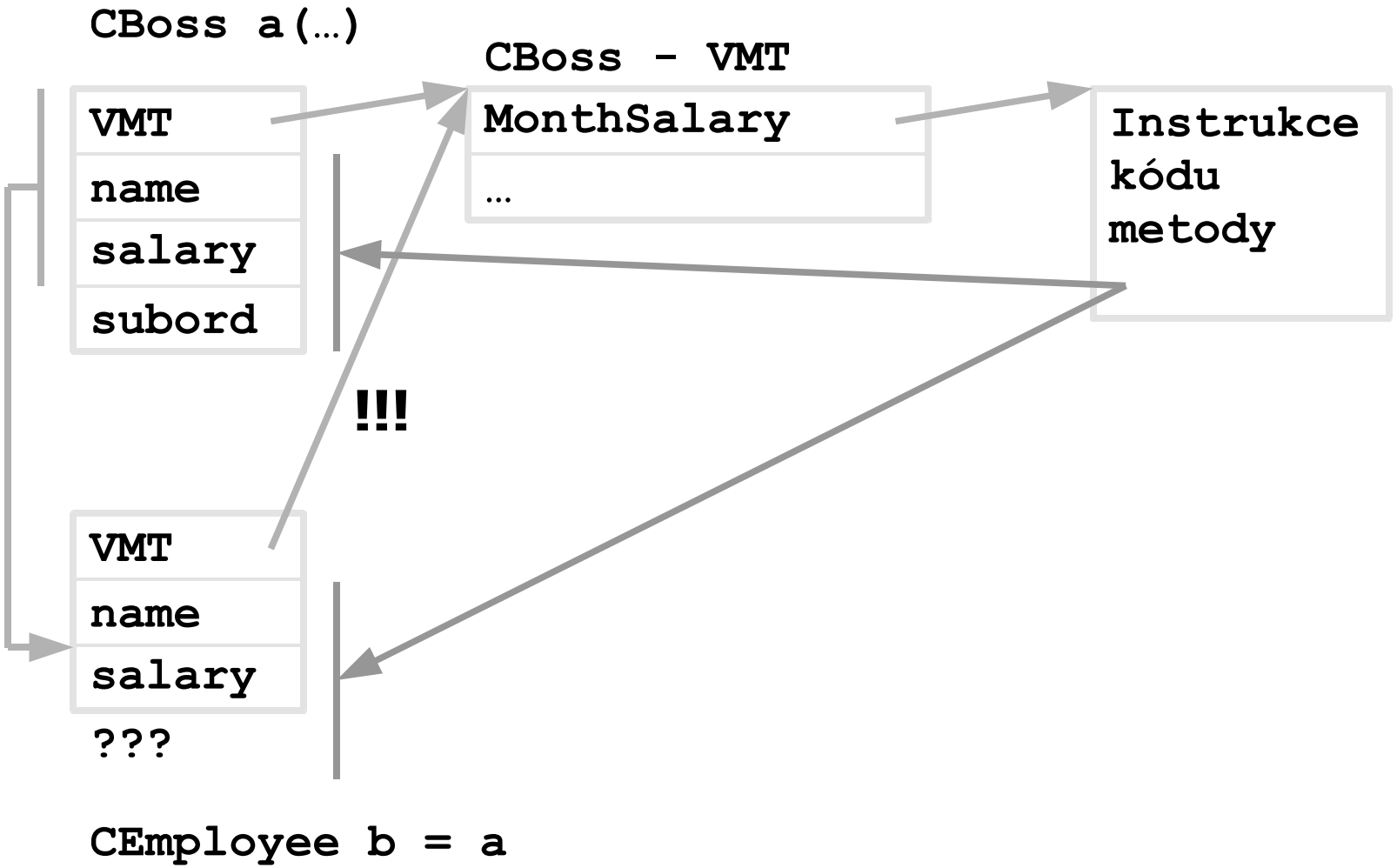
Dědění a polymorfismus



Dědění a polymorfismus



Dědění a polymorfismus



Dědění a operátor <<

```
class CEmployee
 { …
   friend ostream & operator << ( ostream & os,
                                  const CEmployee & x );
 };
ostream & operator << ( ostream & os,
                        const CEmployee & x )
 { os << "Employee " << x . name; return os; }

class CBoss : public CEmployee
 { …
   friend ostream & operator << ( ostream & os,
                                  const CBoss & x );
 };
ostream & operator << ( ostream & os,
                        const CBoss & x )
 { os << "Boss " << x . name; return os; }
```

Dědění a operátor <<

```
CEmployee dept[3];
int i;

dept[0] = new CBoss ( "Novak", 200, 2 );
dept[1] = new CEmployee ( "Novotny", 100 );
dept[2] = new CEmployee ( "Novotna", 100 );

for ( i = 0; i < 3; i ++ )
 cout << *dept[i];

for ( i = 0; i < 3; i ++ )
 delete dept[i];

Employee Novak     // !!
Employee Novotny
Employee Novotna
```

Dědění a operátor <<

- Operátor výstupu se chová jako staticky vázaný:
  - je přetížen funkcí,
  - funkce nejsou dynamicky vázané.
- Řešení:
  - přetížit jej metodou – nelze (první operand musí být typu **`ostream`**),
  - udělat virtuální funkci – nelze (**`virtual friend`** je nesmysl),
  - přidat pomocnou metodu **`print`**,
  - přetížení operátoru pro potomky (zde pro třídu **`CBoss`**) je pak zbytečné.

Dědění a operátor <<

```
class CEmployee
 { …
   friend ostream & operator << ( ostream & os,
                          const CEmployee & x );
   virtual void print (ostream & os ) const
    { os << "Employee " << name; }
 };
class CBoss : public CEmployee
 { …
   friend ostream & operator << ( ostream & os,
                          const CBoss & x );
   virtual void print (ostream & os ) const
    { os << "Boss " << name; }
 };
ostream & operator << ( ostream & os,
                        const CEmployee & x )
 { x . print ( os ) return os; }
```

Dědění a operátor <<

```
CEmployee dept[3];
int i;

dept[0] = new CBoss ( "Novak", 200, 2 );
dept[1] = new CEmployee ( "Novotny", 100 );
dept[2] = new CEmployee ( "Novotna", 100 );

for ( i = 0; i < 3; i ++ )
 cout << *dept[i];

for ( i = 0; i < 3; i ++ )
 delete dept[i];

Boss Novak          // OK
Employee Novotny
Employee Novotna
```

Dotazy...

Děkuji za pozornost.