

3. Dědičnost a polymorfismus. Výjimky.

BD5B37PPC – Programování v jazyce C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Dědičnost a polymorfismus

Polymorfismus

Přetížené operátory

Dědičnost

Statická a dynamická vazba

- Část 2 – Výjimky

Výjimky

- Část 3 – Ještě ke konstruktorům

Konverzní konstruktor

Kopírující konstruktor

Část I

Dědičnost a polymorfismus

I. Dědičnost a polymorfismus

Polymorfismus

Přetížené operátory

Dědičnost

Statická a dynamická vazba

Polymorfismus

- Polymorfismus je vlastnost OOP programovacího jazyka která umožňuje:
 - jednomu objektu volat jednu metodu s různými parametry (ad-hoc polymorfismus);
 - přetěžování operátorů neboli provedení rozdílné operace v závislosti na typu operandů, overloading;
 - objektům odvozeným z různých tříd volat tutéž metodu se stejným významem v kontextu jejich třídy, často pomocí rozhraní;
 - jedné funkci dovolit pracovat s argumenty různých typů (parametrický polymorfismus, ne ve všech programovacích jazycích).
- Rozhodnutí o tom, která metoda bude volána, je u polymorfismu prováděno až za běhu programu (tj. dynamicky pomocí virtuálních funkcí). Tím se odlišuje od přetěžování funkcí, kde je rozhodnutí o volání vhodné funkce provedeno již při překladu (tj. staticky).

I. Dědičnost a polymorfismus

Polymorfismus

Přetížené operátory

Dědičnost

Statická a dynamická vazba

Přehled

- Pro přetěžování operátorů jsou jistá omezení
 - přetížení mohou být jen existující operátory,
Není možné zavést nový operátor, např. operátor \$.
 - aritu, asociativitu a prioritu operátorů nelze změnit,
Např. operátor += musí být binární.
 - nelze přetěžovat operátory vestavěných typů.
Nelze změnit způsob sčítání čísel typu integer.
- Přetížení je třeba si dobře rozmyslet
 - přetížené operátory mají význam v matematice (např. naše komplexní čísla, velká čísla, vektory, ...) a řetězce (konkatenace – spojování),
 - kolekce (jako vektory, množiny, tabulky, ...) používají přetížené operátory pro přístup k uloženým hodnotám,
 - jiné třídy mohou vyžadovat přetížené operátory <<, >> a =,
 - jste-li na pochybách, dejte přednost metodě před přetíženými operátory.

Přehled operátorů

- Operátory, které lze přetížit

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>	<code>&</code>	<code> </code>
<code>~</code>	<code>!</code>	<code>=</code>	<code><</code>	<code>></code>	<code>+=</code>	<code>-=</code>	<code>*=</code>
<code>/=</code>	<code>%=</code>	<code>⋆=</code>	<code>&=</code>	<code> =</code>	<code><<</code>	<code>>></code>	<code>>>=</code>
<code><<=</code>	<code>==</code>	<code>!=</code>	<code><=</code>	<code>>=</code>	<code>&&</code>	<code> </code>	<code>++</code>
<code>--</code>	<code>->*</code>	<code>,</code>	<code>-></code>	<code>[]</code>	<code>()</code>	<code>new</code>	<code>delete</code>
<code>new []</code>	<code>delete[]</code>						

- Operátory, které nelze přetížit

<code>.</code>	<code>.*</code>	<code>::</code>	<code>?:</code>	<code>sizeof</code>
----------------	-----------------	-----------------	-----------------	---------------------

Binární operátory

`+`, `-`, `*`, `...`, `+=`, `-=`, `...`

- zápis operace: `x @ y`
- signatura: `T1 @ T2 -> T3`
- metoda v `T1`: `T3 operator @ (T2)`
- alt. volání: `x.operator @ (y)`
- funkce: `T3 operator @ (T1, T2)`
- alt. volání: `operator @ (x, y)`

Přetížení operátoru `@` automaticky nepřetíží operátor `@=`. Je-li to požadováno, operátor musí být přetížen separátně.

Příklad přetížení binárního operátoru

- Deklarace operátoru spojování řetězců

```
class String {
public:
    ...
    String & operator+= (const String &s) {
        _length += s._length;
        char *temp = _data;
        _data = new char[_length +1];
        strcpy(_data,temp);
        strcat(_data,s._data);
        delete [] temp;
        return *this;
    }
};
```

- `y+=z` je ekvivalentní `y.operator+=(z)`.

Unární prefixové operátory

`+`, `-`, `*`, `&`, `~`, `!`, `++`, `--`

- zápis operace: `@ x`
- signatura: `@ T1 -> T2`
- metoda v `T1`: `T2 operator @ ()`
- alt. volání: `x.operator @ ()`
- funkce: `T2 operator @ (T1)`
- alt. volání: `operator @ (x)`

Příklad přetížení unárního operátoru

- Deklarace operátoru, který testuje, zda je řetězec prázdný

```
class String
{
public:
    ...
    bool operator! ()
    {
        return this->data == "";
    }
    ...
};
```

- `!s` je ekvivalentní volání funkce `s.operator!()`.

Unární postfixové operátory

++, --

- zápis operace: `x @`
- signatura: `T1 @ -> T2`
- metoda v `T1`: `T2 operator @ (int)`
- alt. volání: `x.operator @ (0)`
- funkce: `T2 operator @ (T1, int)`
- alt. volání: `operator @ (x, 0)`

Přřazení

- zápis operace: `x = y`
- signatura: `T = T1 -> T`
- metoda v `T1`: `T & operator = (const T1 &)`
- alt. volání: `x.operator = (y)`
- funkce: -
- alt. volání: -

- Obvykle bývají `T1` a `T` stejné typy.
- Návrátová hodnota může být buď typu `T &`, nebo `void`.

Indexování

- zápis operace: `x[y]`
- signatura: `T1[T2] -> T3`
- metoda v T1 (a): `T3 & operator [] (T2)`
- metoda v T1 (b): `const T3 & operator [] (T2) const`
- alt. volání: `x.operator [] (y)`
- funkce: -
- alt. volání: -

- První způsob přetížení je určen pro nekonstantní objekty a umožňuje použít výsledek jako l-value (tzn. na levé straně přiřazení).
- Druhý způsob přetížení je určen pro konstantní objekty, výsledek není modifikovatelný.

Dereference

- zápis operace: `x -> met`
- signatura: `T -> T1*`
- metoda v `T`: `T1* operator -> ()`
- alt. volání: `(x.operator -> ()) -> met`
- funkce: -
- alt. volání: -

Volání funkce

- zápis operace: `x (x1, x2, ..., xn)`
- signatura: `T (T1, T2, ..., Tn) -> Tx`
- metoda v `T`: `Tx operator () (T1, T2, ..., Tn)`
- alt. volání: `(x.operator () (x1, x2, ..., xn))`
- funkce: -
- alt. volání: -

I. Dědičnost a polymorfismus

Polymorfismus

Přetížené operátory

Dědičnost

Statická a dynamická vazba

- Problém – vývoj třídy reprezentující čítač (čítající celá čísla).
- Čítač má čítat s určitým modulem m .
- Už dříve jsme vyvinuli celočíselný čítač – třídu `Counter`.
- Nová třída jen rozšíří (vylepší) funkci této existující třídy.
- Novou třídu odvodíme z existující třídy prostřednictvím dědičnosti:
 - zdědí se všechny existující položky (členské proměnné),
 - lze přidat nové položky,
 - existující metody lze zdědit nebo je přepsat (override),
 - lze přidat nové metody.

Dědičnost – příklad 1/4

```
class Counter
{
protected:
    int value;
    int load;
public:
    void increment ( ) { value++; }
    void decrement ( ) { value--; }
    void reset ( ) { value = load; }
    int get ( ) const { return value; }
    Counter (int val):value(val) { reset(); }
};
```

Dědičnost – příklad 2/4

```
class CounterMod : public Counter
{
protected:
    int mod;
public:
    void increment ();
    void decrement ();
    CounterMod ( int val, int modulus );
};
```

Dědičnost – příklad 3/4

```
CounterMod::CounterMod ( int v, int m ) :  
    // volání konstrukturu předka  
    Counter ( v % m ) { mod = m; }  
  
void CounterMod::increment () {  
    Counter::increment ();  
    // volání metody increment () předka  
    val = val % mod;  
}  
  
void CounterMod::decrement () {  
    Counter::decrement ();  
    val = val % mod;  
}
```

Dědičnost – příklad 4/4

```
int menu () { ... }
// ret: 0 = konec, 1,2,3 = manipulace z čítačem
int main () {
    CounterMod cnt ( 0, 5 );
    for (;;) {
        cout << "Val = " << cnt . get ( ) << endl;
        switch ( menu ( ) ) {
            case 1: cnt . increment ( ); break;
            case 2: cnt . decrement ( ); break;
            case 3: cnt . reset ( ); break;
            case 0: return 0;
        }
    }
}
```

Dědičnost v C++

- v C++ je možno dědit od více předků,
- nelze však přímo dědit dvojmo od jedné třídy,
- předek musí být plně deklarován,
- dědí se metody a datové položky,
- nedědí se konstruktory, destruktory a přetížený operátor =,
- při dědění lze měnit přístupová práva k datovým složkám i metodám.

Odvozené třídy

- Odvozené třídy mají tuto syntaxi

```
class T1 : public T
{
    // deklarace nových členských proměnných
    // deklarace nových členských funkcí (metod)
    // deklarace přepsaných metod
};
```

- Členské proměnné v odvozené třídě:
 - existující proměnné jsou vždy zděděny,
 - existující proměnné nemohou být odebrány,
 - datové typy existujících proměnných nemohou být změněny,
 - mohou být přidány nové proměnné.
- Metody v odvozených třídách:
 - existující metody jsou zděděny,
 - existující metody mohou být přepsány (stejná signatura, jiná implementace).
 - mohou být přidány nové metody.

Odvozené třídy – viditelnost

- Zachování viditelnosti:

```
class T1 : public T { ... };
```

- členské proměnné a funkce (metody) zděděné z `T` mají stejnou viditelnost v `T1`.
- Změna viditelnosti všech zděděných členů na `private`:

```
class T1 : private T { ... };
```

- zděděné členy nejsou mimo `T1` viditelné,
 - efektem je dosažení dědičnosti členů, ale potlačení polymorfismu.
- Když není specifikována viditelnost:

```
class T1 : T { ... };  
// class T1 : private T ... ;  
struct T1 : T { ... };  
// struct T1 : public T ... ;
```

Odvozené třídy a operátor přiřazení

- Instance odvozené třídy (potomka) může být přiřazena předkovi.
- Instance báze třídy (předka) nemůže být přiřazena potomkovi.

```
class T { ... };  
class T1 : public T { ... };  
class T2 : public T { ... };
```

```
T x; T1 x1; T2 x2;  
T *p; T1 *p1; T2 *p2;
```

```
x = x1; x = x2; // obě přiřazení ok  
x1 = x; x1 = x2; // obě přiřazení chybně  
p = p1; p = p2; // obě přiřazení ok  
p1 = p; p1 = p2; // obě přiřazení chybně
```

Konstruktory a destruktory

- konstruktory předků se volají před vstupem do těla konstruktoru potomka,
- je-li více předků, jejich konstruktory se volají v pořadí, v jakém byly napsány,
- destruktory předka je volán až po dokončení těla destrukturu potomka,
- je-li předkem odvozená třída, aplikují se tato pravidla rekurzivně.

Virtuální dědění

- pokud se shodují názvy složek od různých předků případně název složky předka a potomka, lze je odlišit `::`,
- problém nastává např. při opakovaném dědění:

```
class A {int a};  
class B:A {int b};  
class C:A {int c};  
class D:B, C {int d};
```

- neexistenci dvou výskytů třídy A v třídě D lze zařídit pomocí **virtuálního dědění**:

- stačí předka A definovat ve třídách B a C jako virtuálního:

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

- je-li třída děděna virtuálně i nevirtuálně, pak je v potomkovi obsažena jednou za všechna virtuální dědění a jednou za každé nevirtuální,
- v konstruktoru se volají nejprve konstruktory virtuálních předků.

I. Dědičnost a polymorfismus

Polymorfismus

Přetížené operátory

Dědičnost

Statická a dynamická vazba

Statická a dynamická vazba

```
Counter c (0);  
CounterMod cm (0, 5);  
...  
c.increment (); // Counter::increment()  
cm.increment (); // CounterMod::increment()  
c = cm;  
c.increment ();
```

- Operace je určena datovým typem proměnné stojící vlevo od operátoru `.` nebo `->`, zde tedy typem proměnné `c`.
- Metoda je vybrána v době kompilace.
- Statická vazba metod.

Statická a dynamická vazba

```
Counter * p = new Counter (0);
CounterMod * pm = new CounterMod (0, 5);
...
p->increment (); // Counter::increment()
pm->increment (); // CounterMod::increment()
delete p;

p = pm;
p->increment (); // Counter::increment()
```

- Operace je opět určena datovým typem proměnné `p`.
- Metoda je vybrána v době kompilace.
- Opět statická vazba metod.

Statická a dynamická vazba

- Statická vazba je rychlejší, ale neumožňuje polymorfismus:
 - volající musí rozlišit objekty a jejich typy,
 - volající musí mít znalost všech možných odvozených tříd,
 - volající (vyšší úroveň abstrakce) se musí starat o implementační detaily objektů, které používá.
- Dynamická vazba určuje volanou metodu až v době výpočtu, na základě objektů, které jsou zpracovávány:
 - volání je trochu pomalejší,
 - volající zpracovává objekty (např. zde čítače) a provádí nějaké operace. Nemusí rozlišovat typy objektů při provádění operací,
 - existující kód není třeba modifikovat, pokud do programu přidáme nové odvozené třídy.

Statická a dynamická vazba

- Vyvineme funkci `funWithCounter`, která bude provádět logiku čítání (menu, modifikace instance čítače).
- Tato funkce bude schopna pracovat s každým objektem odvozeným z třídy `Counter`.
- Třída `Counter` bude používat dynamickou vazbu.

```
void funWithCounter ( Counter * x )
{
    for (;;) {
        cout << "Val = " << x -> get ( ) << endl;
        switch ( menu ( ) ) {
            case 1: x -> increment ( ); break;
            case 2: x -> decrement ( ); break;
            case 3: x -> reset ( ); break;
            case 0: return;
        }
    }
}
```

Statická a dynamická vazba

```
class Counter
{
protected:
    int value;
    int load;
public:
    virtual void increment ( ) { value++; }
    virtual void decrement ( ) { value--; }
    void reset ( ) { value = load; }
    int get() const { return value; }
    Counter(int val):value(val) {reset ( ); }
};
```

- Klíčové slovo `virtual` indikuje dynamickou vazbu konkrétní metody.

Statická a dynamická vazba

```
class CounterMod : public Counter {
    ...
    virtual void increment () {
        Counter::increment ();
        value %= mod;
    }
    virtual void decrement () {
        CCounter::decrement ();
        value %= mod;
    }
};
```

- Klíčové slovo `virtual` zde není třeba – metody jsou automaticky dynamicky vázány (zděděno po předkovi); může ale zvýšit čitelnost zdrojového kódu.

Statická a dynamická vazba

```
...  
int main()  
{  
    Counter c (0);  
    CounterMod cm (0, 24);  
    cout << "Counter" << endl;  
    funWithCounter (&c);  
    cout << "CounterMod" << endl;  
    funWithCounter (&cm);  
    return 0;  
}
```

Statická a dynamická vazba

- Mohlo by být rozhraní funkce `funWithCounter` změněno takto?

```
void funWithCounter (Counter & x)
```

Ano, program by pracoval správně.

- A co takto?

```
void funWithCounter (Counter x)
```

Nikoli, program bude pracovat s operacemi třídy `Counter` (jako kdyby nebyla dynamická vazba).

Proč?

Statická a dynamická vazba

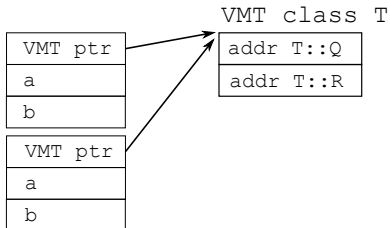
- Když je objekt předán přes ukazatel (nebo referenci), kód ve funkci `funWithCounter` pracuje s originálním objektem:
 - má originální instanci,
 - má originální interface.
- Když je objekt předán jako kopie objektu, pak:
 - je vytvořena instance `Counter` (kopírující konstruktor),
 - nová instance je objekt typu `Counter`,
 - ten má interface `Counter` a metody `Counter`.
- To není překvapující: objekt se chová jako `Counter`, protože to je instance `Counter`.
- Proč se v tomto chování C++ liší od Javy (a dalších jazyků)?
 - Java nemá možnost předávat objekty hodnotou.
 - Java vždy předává pouze referencí.

Jak dynamická vazba pracuje

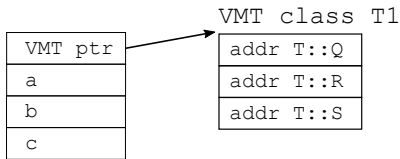
- Při dynamické vazbě je metoda určena v době běhu programu:
 - objekty musí "znát" svoji třídu, aby bylo možno metodu této třídy najít,
 - metoda musí být vybrána velmi rychle.
- C++ používá VMT (Virtual Method Table) k určení metody v konstantním čase:
 - VMT je připravena pro každou třídu s virtuálními funkcemi; tabulka je generována kompilátorem,
 - VMT obsahuje pole adres metod s dynamickou vazbou,
 - každý objekt s dynamickou vazbou metod má ukazatel na svou VMT, tento ukazatel je inicializován konstruktorem,
 - metody jsou ve VMT uspořádány tak, že jméno metody koresponduje s offsetem v tabulce,
 - pro odvozené třídy se zachovává pořadí metod v tabulce.
- Volání virtuální metody znamená index ve VMT (2x dereference) a volání kódu referencovaného tímto ukazatelem.

Jak dynamická vazba pracuje

```
class T {  
    int a, b;  
public:  
    void P ();  
    virtual void Q ();  
    virtual void R ();  
};
```



```
class T1 : public T {  
    int c;  
public:  
    virtual void R ();  
    virtual void S ();  
    void U();  
};
```



Jak dynamická vazba pracuje

- Ve VMT jsou umístěny jen virtuální metody.
- Metoda `T::Q` není přepsána, její adresa je převzata do VMT třídy `T1`.
- Metoda `Q` má přiřazen index `0` v VMT, metoda `R` má index `1`.
- Tyto indexy musí být zachovány i v odvozených třídách (např. v `T1`).
- Nové metody (např. `S`) jsou přidány na konec. Metoda `S` bude mít index `2`, což bude zachováno i ve všech potomcích třídy `T1`.
- Struktura VMT je jednoduchá díky jednoduchému dědění (jeden předek). V případě vícenásobného dědění (C++ vícenásobné dědění umožňuje) se struktura VMT komplikuje.

Abstraktní třídy

- slouží pro implementaci obecného předka dalších tříd,
- některé metody takového předka nemá smysl definovat:
 - takováto metoda je označena jako virtuální a místo těla má řetězec "=0;" ,
 - takováto metoda se nazývá čistě virtuální,
- nesmí být vytvořena instance abstraktní třídy,
- pointer nebo reference však ano.

Část II

Výjimky

II. Výjimky

Výjimky

Výjimky

- V programování se výjimkou rozumí nějaká neočekávaná událost, která může vést i ke zhroutení samotného programu.
- Výjimky v C++
 - Část kódu, ve kterém může nastat výjimka, uzavřeme do bloku `try`;
 - pokud bloku `try` dojde k nějaké nečekané situaci, řízení programu se předá do bloku `catch`; bloků `catch` může být více;
 - pokud k žádné výjimce nedojde, blok `catch` se přeskočí.

dělení nulou, zápis za konec pole, čtení neexistujícího souboru, ...

```
try
{
    // nějaký kód, který může způsobit výjimku
}
catch (typ vyjimky)
{
    // kód, který se provede při vyvolání výjimky
}
```

Příklad

```
int main() {
    double a, b;
    cin >> a;
    cin >> b;

    try {
        if (b == 0)
            throw "Deleni nulou.\n";
        cout << a / b << endl;
    }
    catch (const char* exception) {
        cout << "Byla zachycena vyjimka - " << exception;
    }
    return 0;
}
```

Ošetření více výjimek

- V programu může dojít k více výjimkám
- Je možné napsat více bloků `catch` pro různé výjimky.
- Existuje blok `catch(...)`, který dokáže zachytit všechny výjimky.

```
try
{
    NejakaNebezpecnaFunkce();
}
catch (int)
{
    // zachycení výjimky datového typu int
}
catch (...)
{
    // zachycení všech neošetřených výjimek
}
```


Standardní výjimky v C++

C++ definuje třídu standardních výjimek, včetně interface a datových typů.

- `<exceptions>`
 - `std::exception` – bazová třída
 - `virtual what()` – vrací řetězec s popisem výjimky
 - `terminate()` – ukončuje program, volá `abort()`
- `<stdexcept>`
 - `std::out_of_range` – přístupu mimo rozsah
 - `std::overflow_error` – přetečení
- `<new>`
 - `std::bad_alloc` (memory allocation fails)
 - `std::nothrow`

Co když není výjimka zachycena?

```
void myFunction()
{
    std::cerr << "Nezachycena vyjimka.\n";
    std::cerr << "Program bude ukoncen.\n";
    exit(1);
}

int main()
{
    std::set_terminate(myFunction);
    throw 1;
    return 0;
}
```

Výjimka při alokaci dynamické paměti

```
try {
    while (true) {
        new int[1000000000ul];    // throwing overload
    }
} catch (const std::bad_alloc& e) {
    std::cout << e.what() << '\n';
}

while (true) {
    int* p = new (std::nothrow) int[1000000000ul]; //
        non-throwing overload
    if (p == nullptr) {
        std::cout << "Allocation returned nullptr\n";
        break;
    }
}
```

Vlastní výjimka jako derivace třídy exception

```
class myexception: public exception
{
    virtual const char* what() const throw()
    {
        return "Moje vyjimka";
    }
} myex;

int main () {
    try {
        throw myex;
    }
    catch (exception& e) {
        cout << e.what() << '\n';
    }
    return 0;
}
```

Část III

Ještě ke konstruktorům

III. Ještě ke konstruktorům

Konverzní konstruktor

Kopírující konstruktor

Přetypování

- V C++ lze stejně jako v C použít explicitní přetypování
- K dispozici je ještě alternativní formát zápisu – konverzní konstruktor
 - má právě jeden parametr
 - pokud se při vytváření instance objeví přiřazení jiného datového typu, překladač se pokusí najít odpovídající konstruktor

```
double a = 3.1415;  
// konverze známá z jazyka C  
int i = (int)a;  
// konverzní konstruktor  
int j = int(a);
```

Příklad

```
struct Double
{
    Double (int a) { cout << "int\n"; };
    // zákaz implicitní konverze
    explicit Double (long a) { cout << "long\n"; };
};

int main()
{
    Double a = 10L;
    // co kdyby byla hodnota a třeba 10000000000L?
}
```


III. Ještě ke konstruktorům

Konverzní konstruktor

Kopírující konstruktor

Motivační příklad

- Mějme třídu `Vektor`, která popisuje vektor celých čísel
- Data jsou uložena v dynamické paměti
- Implementace obsahuje přetížený operátor pro přístup k datům
- Přístup k indexům mimo alokaci vyvolá výjimku

```
class Vektor {
    int * data;
    int velikost;
public:
    explicit Vektor (int v):velikost(v) {
        data = new int[velikost];
        for (int i = 0; i < velikost; i++) data[i] = 0;
    }
    ~Vektor () {delete [] data;}
    int operator[] (int idx);
};
```

Motivační příklad

```
int & Vektor::operator [] ( int idx ) {
    if ( idx < 0 || idx >= velikost )
        throw "Index mimo meze";
    return data[idx];
}

int main()
{
    Vektor a(5);
    a[1] = 10;

    for (int i = 0; i < 5; i++)
        std::cout << a[i] << " ";
    std::cout << std::endl;

    return 0;
}
```

Motivační příklad

```
int main()
{
    Vektor a(5), b(5);

    a[4] = 3;
    b = a;    // problém č. 1
    b[3] = 5; // problém č. 2

    for (int i = 0; i < 5; i++)
        std::cout << a[i] << " ";
    std::cout << std::endl;
    // 0 0 0 5 3
}
```

Mělká kopie

- Objekt byl okopírován použitím operátoru `=`.
- Třída `Vektor` nemá operátor `=` přetížený, takže si jej kompilátor vymyslí:
 - objektové členské proměnné kopíruje jejich operátorem `=`,
 - primitivní datové typy (včetně ukazatelů a ukazatelů na objekty) kopíruje po bitech.
- Binární kopie ukazatele na dynamicky alokovaná data referencuje stejnou paměť (tedy stejná data).
- To se nazývá mělká kopie (shallow copy).
- Mělká kopie není problémem, pokud jsme si toho vědomi.
- Destruktor ale problémem je (stejný dynamicky alokovaný blok je uvolněn dvakrát nebo i vícekrát).

Hluboká kopie

- Sémantika operátoru = (deep copy) je jasná:
 - chceme okopírovat zdrojová data (vpravo) na cílová (vlevo),
 - původní data na levé straně budou zřejmě zničena,
 - objekty musí zůstat nezávislé.
- Kopírování adresy je nedostatečné, je třeba kopírovat obsah:
 - musí být připraven nový dynamicky alokovaný blok,
 - obsah pole musí být zkopírován,
 - dříve alokované pole musí být zrušeno.

```
Vektor & Vektor::operator = (const Vektor & src) {  
    if ( &src == this ) return *this;  
    delete [] m_Data;  
    velikost = src.velikost;  
    data = new int [velikost];  
    for ( int i = 0; i < velikost; i ++ )  
        data[i] = src.data[i];  
    return *this;  
}
```

Kopírující konstruktor

- Deklarace

```
Vektor b = a;
```

deklaruje nový objekt **b** a inicializuje jej obsahem objektu **a**.

- Deklarace je identická jako

```
Vektor b ( a );
```

- V obou předchozích případech inicializace se volá kopírující konstruktor :

```
Vector( const CVector & src );
```

- Když kopírující konstruktor neexistuje (programátor ho nenapsal), kompilátor automaticky připraví kopírující konstruktor:
 - objektové členské proměnné jsou kopírovány jejich kopírujícími konstruktory,
 - primitivní datové typy (včetně ukazatelů a ukazatelů na objekty) jsou kopírovány po bitech.

Toto je v podstatě mělká kopie objektu.

Kopírující konstruktor

```
Vektor::Vektor ( const Vektor & src ) {  
    velikost = src.velikost;  
    data = new int [velikost];  
    for ( int i = 0; i < velikost; i++ )  
        data[i] = src.data[i];  
}
```

- Konstruktor je podobný operátoru =, ale
 - chybí podmínka,
 - chybí delete.

Proč?

Kopírující konstruktor

- Kopírující konstruktor a operátor = jsou použity pro vytvoření hluboké kopie objektu. Rozdíl je v cílovém objektu:
 - operátor = má na levé straně plně funkční objekt. Tudíž zdroje alokované pro cílový objekt musí být před kopírováním uvolněny,
 - kopírující konstruktor nemá žádný objekt k modifikaci. Místo toho má cíl již rezervovanou (neinicializovanou) paměť, která musí být inicializována.
- Kopírující konstruktor, operátor = a destruktory jsou těsně svázané. Je-li důvod pro implementaci destruktora, je také důvod pro implementaci kopírujícího konstruktora a operátoru = (pokud kopírování není zakázáno).
- Pravidlo 3 (rule-of-three): buď třída implementuje všechny tři (destruktory, kopírující konstruktor a operátor =), nebo ani jeden z nich.