

PB161 Programování v jazyce C++

Přednáška 9

Právo `friend`
Přetěžování operátorů

Nikola Beneš

16. listopadu 2015

PB173 Tematicky zaměřený vývoj aplikací v jazyce C/C++

- různé tematické skupiny
- možný opakovaný zápis
- v semestru jaro 2016: jedna nebo dvě skupiny zaměřené na C++11/C++14
 - chytré ukazatele
 - paralelní programování
 - metaprogramování, použití šablon podrobně
 - funkcionální prvky, lambdy
- informace z minulého běhu: <http://cecko.eu/public/pb173>

Parametry: brát hodnotou nebo referencí?

- co s parametrem hodláme dělat?
- jde o primitivní typ nebo objekt?
- parametr budeme měnit – referencí
- parametr nebudeme měnit
 - primitivní datové typy – hodnotou
 - objekty – konstantní referencí
 - jednoduché objekty?
- parametr nebudeme měnit, ale budeme určitě vytvářet jeho kopii
 - hodnotou

```
void f(const Type & x) {  
    Type y = x;  
    // ...  
}
```

```
void f(Type y) {  
    // ...  
    // ...  
}
```

Kdy použít `inline`?

- původní záměr `inline`: náповěda pro překladač
 - moderní překladače umí posoudit lépe než programátor
- `inline` při linkování:
 - více stejných funkcí ve více překladových jednotkách

Doporučení

- funkce v hlavičkových souborech: `inline`
- funkce ve zdrojových souborech: bez `inline`
- metody definované uvnitř deklarace třídy jsou automaticky `inline`

Spřátelené třídy a funkce



Princip zapouzdření

- přístupová práva (`private`, `protected`)

Přístup k soukromým atributům a metodám z venku

- operátory vstupu a výstupu `>>`, `<<`
- iterátory

Přístup k soukromým atributům jen pro určité třídy

- multiple dispatch (viz vzorové řešení 5. cvičení)

Klíčové slovo **friend**

- způsob jak „obejít“ přístupová práva
- deklarace spřátelených funkcí a třídy
 - přístup k libovolným členům třídy
- porušení principu zapouzdření?
 - záleží na použití
 - nevhodné použití porušuje zapouzdření
 - vhodné použití jej naopak může posilovat

Deklarace přátel (pokr.)

Použití `friend`

- přístup povoluje ten, k jehož soukromým členům má být přístupováno
- uvnitř deklarace třídy (kdekoliv)

- `friend class` JmenoTridy;
- `friend` typ jmenoFunkce(parametry);
 - funkci je možno zároveň i definovat

[ukázka použití `friend`]

Vlastnosti `friend`

- třída si určuje, kdo je její přítel, ne naopak
- přátelství není reciproční
- přátelství není tranzitivní
- přátelství není dědičné
- pokud třída A označí funkci/třidu X jako přítele, X má částečný přístup i k potomkům A (ke zděděným částem)
 - proč?

[ukázka]

Kdy a jak používat `friend`?

- typické využití: operátory
- přístup k atributům jen pro někoho (bez getterů/setterů)
- používejte opatrně a kromě operátorů spíše výjimečně

Přetěžování operátorů



Přetížené operátory

- téměř v každém jazyce (aritmetické operátory pro `int` vs. `float`)
- C++ umožňuje přetížit operátory pro vlastní datové typy
- už jsme viděli:
 - `operator=` pro kopírovací přiřazení
 - `operator()` pro funkční objekty
 - `operator>>` a `operator<<` pro vstup a výstup
 - `operator+` pro řetězce

Proč přetěžovat operátory?

- zlepšit čitelnost kódu
- usnadnit používání třídy
- snížit chyby při použití třídy

Syntax

- volná funkce nebo metoda třídy
- **operator** a označení operátoru
- konkrétní syntaxe závisí na počtu parametrů

Zdroje

- https://en.wikipedia.org/wiki/Operators_in_C_and_C++
- <http://en.cppreference.com/w/cpp/language/operators>

Operátor jako funkce

- typicky **friend**
- unární operátory – jeden parametr
- binární operátory – dva parametry
- už jsme viděli: operátory vstupu a výstupu

$a + b \implies \text{operator}+(a, b)$

[ukázka použití]

Operátor jako metoda

- levý operand je `*this`
- počet parametrů: o jeden méně než implementace funkcí

$a + b \implies a.\text{operator}+(b)$

[ukázka použití]

Jak přetěžovat operátory?

- některé operátory není možno přetěžovat
 - `., ::, ? :`
- některé operátory musí být implementovány jako metody
 - `=, [], ->, ()`
- ostatní je možno implementovat jako metody nebo jako funkce

Doporučení

- pokud nemáme jak zasáhnout do třídy levého objektu: funkce
- unární operátor: metoda
- binární operátor, který se chová ke svým operandům stejně: funkce
- binární operátor, který se chová k levému operandu jinak: metoda
 - nezapomínejte na `const`

Omezení přetěžování operátorů

- nelze definovat nové operátory
- nelze měnit počet parametrů (kromě operátoru `()`)
- nelze definovat nové operátory pro primitivní typy
 - alespoň jeden z paramterů musí být uživatelsky definovaného typu
- nelze měnit prioritu ani asociativitu operátorů

Přiřazení `operator=`

- musí být metoda
- už jsme viděli v části o kopírování
- nezapomeňte na idiom *copy and swap*

```
X & X::operator=(X other) {  
    swap(other);  
    return *this;  
}
```

Přetěžování operátorů

Vstup a výstup `operator>>`, `operator<<`

- operátory bitového posunu s jiným významem
- musí být funkce
- už jsme viděli v části o vstupně-výstupních proudech
- nezapomeňte, že vrací referenci na zadaný proud

```
std::istream & operator>>(std::istream & in, X & x) {  
    // read data from in and update x  
    return in;  
}  
  
std::ostream & operator<<(std::ostream & out, const X & x) {  
    // write x to out  
    return out;  
}
```

Přetěžování operátorů

Funkční volání `operator()`

- musí být metoda
- může brát libovolný počet parametrů
- už jsme viděli v části o algoritmech a funkčních objektech
- doporučení: funkční objekty by měly být snadno kopírovatelné

```
class Adder { // not the snake
    int val;
public:
    Adder(int v) : val(v) {}
    int operator()(int x) { return x + val; }
};
```

Porovnávání `operator==`, `operator!=`, `operator<`, `operator<=`,
`operator>`, `operator>=`

- měly by vracet `bool`
- neměly by měnit své parametry
- typicky jako funkce
- pokud přetížíte jeden, je vhodné přetížit i všechny ostatní

Přetěžování operátorů – porovnávání

```
bool operator<(const X & l, const X & r) {  
    // do the actual comparison  
    return ...;  
}  
  
bool operator>(const X & l, const X & r) { return r < l; }  
bool operator>=(const X & l, const X & r) { return !(l < r); }  
bool operator<=(const X & l, const X & r) { return !(r < l); }  
bool operator==(const X & l, const X & r) {  
    // do the actual comparison  
    return ...;  
}  
  
bool operator!=(const X & l, const X & r) { return !(l == r); }
```

Aritmetické operátory `operator+`, `operator+=` apod.

- silně doporučené: ke každému operátoru přetížit i kombinaci s přiřazením
- `+` apod. by měly být funkce a měly by vracet hodnotu
- `+=` apod. by měly být metody a měly by vracet referenci

- pokud chceme dělat kopii: jeden z operandů (levý) hodnotou
- pokud chceme interakci i s jinými typy: pamatujte na symetrii

[ukázka]

Přetěžování operátorů

Indexování `operator[]` * musí být metoda * typicky chceme konstantní a nekonstantní přetížení

```
class X {  
    // ...  
public:  
    value_type & operator[] (index_type ix);  
    const value_type & operator[] (index_type ix) const;  
};
```


Inkrement a dekrement `operator++`, `operator--`

- typicky jako metody
- dvě varianty: postfixová varianta má (nepoužívaný) parametr `int`
- prefixová varianta by měla vracet referenci
- postfixová varianta by měla vracet hodnotu (kopii)

[ukázka]

Přetěžování operátorů

Dereference `operator*`, `operator->`

- pro typy, které se chovají jako ukazatele (iterátory, chytré ukazatele)
- typicky jako metody
- `operator*` – vrací referenci
- `operator->` – vrací ukazatel nebo něco jako ukazatel (řetězení `->`)

$a \rightarrow b \implies a.operator\rightarrow() \rightarrow b$

```
class MyIterator {  
    value_type * ptr;  
public:  
    value_type & operator*()          { return *ptr; }  
    const value_type & operator*() const { return *ptr; }  
    value_type * operator->()          { return ptr; }  
    const value_type * operator->() const { return ptr; }  
};
```

Přetěžování operátorů

Přetypování operator novotyp

- speciální operátor pro implicitní přetypování
- musí být metoda (bez parametrů)
- nepíše se návratová hodnota

```
class X {  
    int val;  
public:  
    X(int v) : val(v) {}  
    operator int() const { return val; }  
};  
  
int main() {  
    X x(3);  
    int a = x;  
    return x;  
}
```

Přetěžování operátorů – přetypování

Problémy s implicitním přetypováním

// která funkce se zavolá?

```
void f(int a);
```

```
void f(X & x);
```

```
int main() {  
    f( X(3) );  
}
```

- C++11: klíčové slovo **explicit**

```
class X {
```

```
    // ...
```

```
public:
```

```
    // ...
```

```
    explicit operator int() const { return val; }
```

```
};
```

Přetěžování operátorů – přetypování

Použití `explicit` pro `bool`

- překladač smí použít explicitní operátor `bool` pro přetypování, ale jen na `bool`, ne dál
- použití pro objekty, na jejichž stav se můžeme dotazovat pomocí `if`

```
class X {  
    // ...  
    bool is_ok;  
public:  
    // ...  
    explicit operator bool() const { return is_ok; }  
};  
  
int main() {  
    X x;  
    if (x) { /* ... */ }    // OK  
    int a = x;    // CHYBA  
}
```

Klíčové slovo **friend**

- umožňuje obejít přístupová práva
- přátelství není reciproční, transitivní, ani dědičné

Přetěžování operátorů

- můžeme přetěžovat většinu operátorů v C++
- význam operátorů by měl být intuitivně jasný
- držte se doporučení