

PB161 – 9. přednáška (16. listopadu 2015)

Poznámky na úvod

Brát parametry hodnotou nebo referencí?

- záleží (a) co s parametry hodláme dělat a (b) jestli jsou to primitivní typy nebo objekty a případně jak náročná je jejich kopie
- pokud hodláme parametr měnit
 - bereme vždy referencí
- pokud nehodláme parametr měnit
 - složité typy (objekty) bereme vždy konstantní referencí
 - primitivní datové typy bereme hodnotou
 - zde je drobný prostor k diskusi, co s jednoduchými objekty (ale možná je rozumné zacházet se všemi objekty stejně, tj. referencí, a o případné optimalizace se snažit až později, po profilingu)
- moderní C++: pokud nehodláme parametr měnit, ale budeme uvnitř funkce určitě vytvářet jeho kopii
 - bereme hodnotou (tím se vytvoří kopie)
 - to umožňuje optimalizace na straně překladače

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

// vs.

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

*Kdy mám označit funkci jako **inline**?*

- původní záměr **inline**: sloužit jako nápověda pro překladač, kterou funkci má inlinovat (rozbalit její tělo v kódu)
- ale: moderní překladače umí posoudit, kdy inlinovat, mnohem lépe než programátor
- **inline** má ještě jeden význam: při linkování může existovat více stejných **inline** funkcí ve více překladových jednotkách (musí být úplně stejné, jinak dojde k nedefinovanému chování)
- doporučení: pište **inline** k funkcím definovaným v hlavičkových souborech (tam je to nutné, pokud bude hlavičkový soubor použit ve více překladových jednotkách); nemusíte psát **inline** do zdrojových souborů (tam je to nejspíš zbytečné a může to být nebezpečné)

```

// example.h

void fun1(int x); // ok, jen deklarace

inline int fun2(int x) {
    // musí být inline, jinak dojde k chybě při linkování,
    // pokud by více zdrojových souborů používalo tento hlavičkový soubor
    return x + 7;
}

// example.cpp

// definice funkce z hlavičkového souboru
void fun1(int x) {
    // ...
}

// zde může a nemusí být inline
// a překladači to je nejspíš úplně jedno
// navíc, pokud bude v jiné překladové jednotce jiná funkce fun3(int),
// linker to odhalí jako chybu jen když nebude inline
void fun3(int x) {
    // ...
}

```

- pamatujte, že metody definované uvnitř deklarace třídy jsou automaticky **inline**

Spřátelené třídy a funkce

Motivace

- v rámci zapouzdření mají přístup k soukromým atributům jen metody třídy, jsou ale situace, kdy by se hodilo mít přístup k soukromým atributům i z venku
- příklad: operátory vstupu a výstupu >>, <<
 - nemohou být metodami tříd (více v části o přetěžování operátorů)
- jiný příklad: iterátory
- někdy bychom chtěli povolit přístup k některým metodám jen určitým třídám
- příklad: vzorové řešení 5. cvičení (multiple dispatch)

Klíčové slovo **friend**

- způsob jak „obejít“ přístupová práva (**private**, **protected**)
- třída může deklarovat spřátelené funkce a spřátelené třídy
 - mohou přistupovat k libovolným členům (atributům, metodám) třídy
- porušení principu zapouzdření?

- záleží na použití
- nevhodné použití porušuje zapouzdření
- vhodné použití jej naopak může posilovat

Použití *friend*

- přístup povoluje ten, k jehož soukromým členům má být přístupováno
- použití uvnitř deklarace třídy (kdekoliv)
- použití pro třídy
 - *friend class* jmeno_tridy;
- použití pro funkce
 - *friend* typ jmeno_funkce(parametry);
 - musí přesně odpovídat hlavičce funkce
 - přetížené funkce: třeba povolit přístup každé verzi
 - možno použít i pro metody
- funkci je možno zároveň i definovat
 - funkce je pak automaticky *inline*

```
#include <iostream>

class Example {
    int value;
public:
    Example(int val) : value(val) {}
private:
    int privateGet() const { return value; }

    friend void directAccess(const Example &);

    friend class MyFriend;
};

void directAccess(const Example & ex) {
    // přístup k soukromému atributu
    std::cout << "value = " << ex.value << std::endl;
    // přístup k soukromé metodě
    std::cout << "privateGet() result: " << ex.privateGet() << std::endl;
}

class MyFriend {
public:
    void myMethod(Example & ex) {
        ex.value += 7;
    }
};
```

Vlastnosti *friend*

- třída sama si určuje, kdo je její přítel (a komu je tedy povolen přístup)
 - ne naopak: není možno se prohlásit sám za někoho přítele
- přátelství není reciproční
 - to, že někoho deklaruji jako přítele (a povolím mu přístup ke svým soukromým věcem), ještě neznamená, že i on mě má za přítele (a povolí mi přístup)
- přátelství není tranzitivní
 - přátelé mých přátel nemusí nutně být i mí přátelé
- přátelství není dědičné
 - přátelé mých dětí nemusí být mí přátelé
 - mí přátelé nemusí být přátelé mých dětí
- pokud třída A označí funkci/třidu X jako přítele, pak X má částečný přístup i k potomkům A, konkrétně k těm částem, které jsou zděděné z A
 - proč?
 - princip „potomek může zastoupit předka“

```
#include <iostream>

using namespace std;

class Base {
    int v;
    void method1() { cout << "B::method1()\n"; }
    virtual void method2() { cout << "B::method2()\n"; }
    void method3() { cout << "B::method3()\n"; }
    virtual void method4() { cout << "B::method4()\n"; }
    friend void check();
};

class Derived : public Base {
    void method1() { cout << "D::method1()\n"; }
    void method2() override { cout << "D::method2()\n"; }
};

void check() {
    Derived d;
    d.v = 0;
    // d.method1(); // CHYBA
    // d.method2(); // CHYBA
    d.method3();
    d.method4();
    Base & rb = d;
    rb.v = 0;
    rb.method1();
    rb.method2();
    rb.method3();
}
```

```

        rb.method4();
    }

    int main() {
        check();
    }

```

*Kdy a jak používat **friend**?*

- typické využití: operátory (viz níže)
- tam, kde je potřeba, aby měl někdo přístup k soukromým atributům vaší třídy, ale zároveň nechcete zveřejňovat gettery/settery všem
- obecná rada (jako obvykle): používejte opatrně a kromě operátorů spíše výjimečně

Přetěžování operátorů

Motivace

- téměř všechny jazyky obsahují přetížené operátory
 - velmi často: aritmetické operace (+ * - /) pro `int` vs. `float`
 - často: přetížení nějakého operátoru pro řetězce
- C++ nám dává možnost, jak přetížit operátory pro vlastní datové typy
 - ne úplně všechny operátory, jak uvidíme dále
- už jsme viděli:
 - přetížený operátor = pro kopírovací přiřazení
 - přetížený operátor () pro funkční objekty
 - přetížené operátory >> a << pro vstup a výstup
 - přetížený operátor + pro řetězce
- k čemu je to dobré?
 - zlepšit čitelnost kódu – co dělá `a.add(b)`? má to význam `a += b` nebo `a + b`?
 - usnadnit používání naší třídy
 - snížit chyby při použití naší třídy
- samozřejmě, všechno záleží na tom, jak operátory implementujeme
 - operátory by měly fungovat intuitivně správně

Syntaxe přetížených operátorů

- funkce nebo metody, které mají místo jména klíčové slovo **operator** a označení operátoru (může mezi nimi být i bílé místo)
 - např. `operator+`, `operator()`, `operator <<`
- konkrétní syntaxe závisí na
 - počtu parametrů
 - tom, jestli jde o funkci nebo metodu
- https://en.wikipedia.org/wiki/Operators_in_C_and_C++ má přehled všech operátorů v C++ včetně jejich typů

- dobrý zdroj je, jako obvykle, i <http://en.cppreference.com/w/cpp/language/operators>

Operátor jako funkce (nečlenský operátor)

- implementace jako samostatná funkce
 - často používáno spolu s deklarací **friend**
- počet parametrů podle arity operátoru
 - unární operátory – jeden parametr (s možnou výjimkou ++ a --, viz dále)
 - binární operátory – dva parametry
- použití operátoru se přeloží na volání funkce
 - místo `a + b` se provede `operator+(a, b)`

```
class Vector3D {
    float x, y, z;
public:
    // ...
    friend Vector3D operator+(const Vector3D &,
                             const Vector3D &);
};

Vector3D operator+(const Vector3D & first,
                   const Vector3D & second) {
    Vector3D result;
    result.x = first.x + second.x;
    result.y = first.y + second.y;
    result.z = first.z + second.z;
    return result;
}
```

- už jsme viděli použití pro operátory vstupu a výstupu

Operátor jako metoda (členský operátor)

- implementace jako metoda třídy
 - levý operand je vždy objekt ***this**
- počet parametrů: o jeden méně než implementace funkcí
- použití operátoru se přeloží na volání metody na objektu daném levým parametrem
 - místo `a + b` se provede `a.operator+(b)`

```
class Vector3D {
    float x, y, z;
public:
    //...
    Vector3D & operator+=(const Vector3D & other) {
        x += other.x;
    }
}
```

```

        y += other.y;
        z += other.z;
        return *this;
    }
};

```

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:
 - použít funkci, pokud nemáme jak zasáhnout do třídy levého objektu (př.: operátory vstupu a výstupu)
 - implementovat unární operátory jako metody
 - pokud se binární operátor chová ke svým operandům stejně (typicky tak, že je nemění), pak jej implementujte pomocí funkce
 - pokud se binární operátor chová k levému operandu jinak (typicky tak, že jej změní), pak jej implementujte pomocí metody
- jako vždy, jsou případy, kdy je vhodné tato doporučení porušit, ale mělo by vás to alespoň vést k zamyšlení, proč je hodláte porušit
- pokud operátor-metoda nemění stav objektu, je vhodné jej deklarovat se specifikátorem `const` (ostatně jako jakoukoli jinou metodu, která nemění stav objektu)

Omezení přetěžování operátorů

- nelze definovat nové operátory
- nelze měnit počet parametrů (kromě operátoru `()`)
- nelze definovat nové operátory pro primitivní typy
 - alespoň jeden z parametrů musí být uživatelsky definovaného typu
- nelze měnit prioritu ani asociativitu operátorů
 - `a + b * c * d` se bude vždy vyhodnocovat jako `a + ((b * c) * d)`

Přetěžování operátorů – přiřazení

- `operator=`
 - musí to 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>>` a `operator<<`
 - jsou to ve skutečnosti operátory bitového posunu
 - pro použití při vstupu a výstupu to musí být funkce (nemáme jak zasáhnout dovnitř I/O tříd)
- už jsme viděli v části o vstupně-výstupních proudech
- nezapomenejte, 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 to být metoda
- už jsme viděli v části o algoritmech a funkčních objektech
- může brát libovolný počet parametrů
- doporučení: funkční objekty by měly být snadno kopírovatelné (předpokládá to standardní knihovna)

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

Přetěžování operátorů – porovnávání

- `operator==`, `operator!=`, `operator<`, `operator<=`, `operator>`, `operator>=`
 - měly by vracet `bool`
 - neměly by měnit své parametry
 - typicky implementované jako funkce
- pokud přetížíte jeden, je vhodné přetížit i všechny ostatní
 - standardní knihovna používá jen operátor `<` (`set`, `map`, třídící a jiné algoritmy) nebo operátor `==` (`unordered_set` apod.)
 - ale uživatelé vaší třídy budou možná chtít používat i `>=` a jiné


```

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

```

- jsou způsoby, jak toto udělat (polo)automaticky, ale to je nad rámec tohoto předmětu

Přetěžování operátorů – aritmetické operátory

- **operator+** apod.
- patří k nim i **operator+=** apod.
 - silně doporučené: ke každému operátoru přetížít i kombinaci s přiřazením
- doporučení:
 - + apod. by měly být funkce a měly by vracet hodnotu
 - += apod. by měly být metody a měly by vracet referenci
 - většinou rozumné: + apod. by měly brát levý operand hodnotou (stejně bychom dělali kopii; srovnajte s idiomem *copy and swap*)

```

class X {
    int val;
public:
    X(int v) : val(v) {}
    X & operator+=(const X & rhs) {
        // do the actual addition
        val += rhs.val;
        return *this;
    }
};

// use inline if we are in a header file
inline X operator+(X lhs, const X & rhs) {
    lhs += rhs;
    return lhs;
}

```

- podobná doporučení platí i pro bitové operátory (&, | apod.), ale ty se nepřetěžují zdaleka tak často (kromě operátorů >> a << pro I/O)

Poznámka: pokud chcete přetěžovat operátory i pro interakci s jinými typy (třeba sčítání `X` a `int`), pamatujte na symetrii – je třeba mít obě verze:

```
X operator+(X lhs, int rhs);
X operator+(int lhs, X rhs);
```

Přetěžování operátorů – indexování

- `operator[]`
 - musí být metoda
- typicky chceme dvě přetížení: konstantní a nekonstantní

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

Přetěžování operátorů – inkrement, dekrement

- `operator++`, `operator--`
 - doporučeno implementovat jako metody
- mají dvě varianty: pro odlišení bere postfixová varianta ještě jeden (nepoužívaný) parametr typu `int`
 - prefixová varianta by měla vracet referenci, postfixová hodnotu (kopii)

```
class X {
    int val;
public:
    X & operator++() {
        // do the actual increment
        ++val;
        return *this;
    }
    X operator++(int) {
        X tmp(*this); // copy
        ++*this;      // call the prefix operator
        return tmp;
    }
};
```

- všimněte si, že postfixová varianta musí vytvářet kopii
 - to je důvod, proč je lepší se naučit psát `++i` místo `i++`, pokud tu kopii explicitně nepotřebujete

Přetěžování operátorů – dereference

- operátory pro typy, které se chovají jako ukazatele (iterátory, chytré ukazatele)
- **operator*** (unární)
 - vrací referenci
- **operator->** je sice binární, ale jeho typ je jako by byl unární
 - vrací ukazatel nebo něco, co se chová jako ukazatel
 - pokud **a** není typ ukazatel, pak **a->b** zavolá **a.operator->()->b** (řešení operátorů **->**)
- podobně jako u **[]** se typicky píšou dvě varianty (konstantní a nekonstantní)
 - u konstantních iterátorů se nekonstantní varianta vynechá

```
class MyIterator {
    value_type * ptr;
public:
    value_type & operator*()      { return *ptr; }
    const value_type & operator*() const { return *ptr; }
    // aby mělo -> smysl, musí být value_type třída nebo struct
    value_type * operator->()      { return ptr; }
    const value_type * operator->() const { return ptr; }
};
```

Přetěžování operátorů – ostatní

- několik operátorů jsme schválně vynechali, protože jejich přetěžování (ačkoli možné) je většinou nevhodné
- občas i nečekané důsledky – např. přetížené **&&** a **||** ztratí svou schopnost zkráceného vyhodnocování!

Přetěžování operátorů – přetypování

- můžeme pro třídu zavést i speciální operátor pro implicitní přetypování
- syntax: **operator novy_typ ()**;
 - nepíše se návratová hodnota, je stejná jako **novy_typ**

```
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;
}

```

- to může mít jisté problémy:

```

// která funkce se zavolá?
void f(int a);
void f(X & x);

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

```

- C++11 zavádí explicitní konverzní operátory, které provádějí konverzi jen při explicitním přetypování (např. pomocí `static_cast`)

```

class X {
// ...
public:
    // ...
    explicit operator int() const { return val; }
};

```

- speciální použití: překladač smí použít explicitní operátor `bool` pro přetypování, ale jen na `bool`; už z něj pak dál nepřetypuje (na `int` apod.)
 - použití pro objekty, na jejichž stav se můžeme dotazovat pomocí `if` (např. I/O proudy)

```

class X {
// ...
    bool is_ok;
public:
    // ...
    explicit operator bool() const { return is_ok; }
};

int main() {
    X x;
    if (x) { /* ... */ }    // OK
    int a = x;    // CHYBA
    // toto by prošlo, kdyby operator bool nebyl explicit
}

```