

PB161 – Programování v jazyce C++

Objektově Orientované Programování

Manipulátory, friend, přetěžování operátorů

STL Algoritmy

STL zavádí nové koncepty

1. Kontejnery

- objekty, které uchovávají jiné objekty bez ohledu na typ
- kontejnery různě optimalizovány pro různé typy úloh
- např. `std::string` (uchovává pole znaků)
- např. `std::list` (zřetězený seznam)

2. Iterátory

- způsob (omezeného) přístupu k prvkům kontejneru
- např. `std::string.begin()`
- přetížené operátory `++` pro přesun na další prvek atd.

3. Algoritmy

- běžné operace vykonané nad celými kontejnery
- např. `sort(str.begin(), str.end())`

STL Algoritmy

- Standardní metody pracující nad kontejnery
- Obsahuje často používané operace (hledání, třídění...)
- Mohou kontejner číst nebo i měnit
- Často využívají (jako argument) iterátory

Algoritmy - dokumentace

- Funkce dostupné v <algorithm>
- <http://www.cplusplus.com/reference/algorithm/>
- Ukázka syntaxe na for_each

for_each

```
template <class InputIterator, class Function>  
Function for_each (InputIterator first, InputIterator last, Function f);
```

Apply function to range

Applies function *f* to each of the elements in the range [first,last).

The behavior of this template function is equivalent to:

```
1 template<class InputIterator, class Function>  
2   Function for_each(InputIterator first, InputIterator last, Function f)  
3   {  
4       for ( ; first!=last; ++first ) f(*first);  
5       return f;  
6   }
```

Základní dostupné STL algoritmy

- Vyhledávání, statistika (nemodifikují)
 - `find()`, `search()`, `count()`...
- Modifikující kontejnery
 - `copy()`, `remove()`, `replace()`, `transform()`
- Aplikace uživatelské funkce
 - `for_each()` – nemodifikuje původní kontejner
 - `transform()` – modifikuje původní kontejner
- Řadící
 - `sort()`
 - vhodný řadící algoritmus automaticky vybrán dle typu kontejneru
- Spojování rozsahů, Minimum, maximum...
- A spousta dalších
 - <http://www.cplusplus.com/reference/algorithm/>

STL algorithmy - find

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>
using std::cout;
using std::endl;
int main() {
    std::list<int> myList;
    myList.push_back(1);myList.push_back(2);myList.push_back(3);
    myList.push_back(4);myList.push_back(5);
    // 1, 2, 3, 4, 5

    std::list<int>::iterator iter;
    // Find item with value 4
    if ((iter = std::find(myList.begin(), myList.end(), 4)) != myList.end()) {
        cout << *iter << endl;
    }
    // Try to find item with value 10
    if ((iter = std::find(myList.begin(), myList.end(), 10)) != myList.end()) {
        cout << *iter << endl;
    }
    else cout << "10 not found" << endl;
    return 0;
}
```

STL algorithmy – for_each a transform

```
#include <iostream>
#include <list>
#include <algorithm>
using std::cout;
using std::endl;
int increase10(int value) {
    return value + 10;
}
void print(int value) {
    cout << value << endl;
}
```

```
int main() {
    std::list<int> myList;
    // ... fill something into myList
    // Apply function to range (non-modifying)
    std::for_each(myList.begin(), myList.end(), print);
    // Apply function to range (will work only for integers) (modifying)
    std::transform(myList.begin(), myList.end(), myList.begin(), increase10);
    return 0;
}
```


STL algoritmy – callback funkce

- Některé algoritmy berou jako parametr funkci
 - aplikují ji na prvky kontejneru

```
std::for_each(myList.begin(), myList.end(), print);
```

```
template<class InputIterator, class Function>  
Function for_each(InputIterator first, InputIterator last, Function f)  
{  
    for ( ; first!=last; ++first ) f(*first);  
    return f;  
}
```

- Může být klasická C funkce (např. `print()`)
- Může být static metoda objektu (viz. dále)
- Může být objekt s přetíženým `operátorem()`
 - tzv. functor (pozdější přednáška)

STL algorithmy – sort

```
// sort algorithm example from http://www.cplusplus.com/reference/algorithm/sort/
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i,int j) { return (i<j); }

int main () {
    int myints[] = {32,71,12,45,26,80,53,33};
    vector<int> myvector (myints, myints+8);           // 32 71 12 45 26 80 53 33
    vector<int>::iterator it;

    // using default comparison (operator <):
    sort (myvector.begin(), myvector.begin()+4);       //(12 32 45 71)26 80 53 33

    // using function as comp
    sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

    cout << endl;

    return 0;
}
```

Přístupové právo friend

Spřátelené metody/objekty

- Způsob jak „obejít“ přístupová práva
 - přístup k `private` a `protected` atributům/metodám jiné třídy
 - není používáno zcela běžně
 - nevhodné použití porušuje zapouzdření, vhodné naopak posiluje
- Využíváno hlavně pro implementaci operátorů
 - operátor typicky potřebuje nízkoúrovňové funkce
 - tyto by se musely zavádět nebo volat (pomalé, porušuje abstrakci)
- Klíčové slovo **friend**
 - **friend** funkce/metody
 - **friend** třídy

Friend - syntaxe

1. friend funkce/metoda

- uvádí se ve třídě, která přístup povoluje
- **friend** typ jméno_funkce (parametry);
- musí přesně odpovídat hlavičce povolované funkce
 - návratová hodnota i typ a počet parametrů
 - tj. při povolování např. přetížených funkcí musí uvést všechny, kterým chceme povolit přístup

2. friend třída

- uvádí se ve třídě, která přístup povoluje
- **friend class** jméno_třídy;
- Deklarace friend metod/tříd může být uvedena kdekoli v rámci třídy

```

class CTest {
private:
    int m_value;
public:
    CTest(int value) : m_value(value) {}

private:
    int privateMethod() const { return m_value; }

    friend void directAccessFnc(const CTest& test);

    friend class CMyFriend;
};

```

povol přístup pro
directAccessFnc()

povol přístup pro
třidu CMyFriend

```

void directAccessFnc(const CTest& test) {
    // Direct access to private attribute
    cout << "CTest::m_value = " << test.m_value << endl;
    // Access to private method
    cout << "CTest::m_value = " << test.privateMethod() << endl;
}

class CMyFriend {
public:
    void directAccessMethod(const CTest& test) {
        // Direct access to private attribute
        cout << "CTest::m_value = " << test.m_value << endl;
        // Access to private method
        cout << "CTest::m_value = " << test.privateMethod() << endl;
    }
};

```

Vlastnosti práva friend

- Třída definuje spřátelené třídy/funkce
 - ne naopak (funkce/třída se nemůže “prohlásit” za friend)
- Není dědičné, transitivní ani reciproční (vzhledem k příjemci)
 - právo se nepřenáší na potomky přítele
 - právo se nepřenáší na přátele mých přátel
 - nejsem automaticky přítelem toho, koho já označím za svého přítele
- Pokud třída A označí funkci/třidu X jako friend, tak X má přístup i k potomkům A
 - proč?
 - (nefungovala by substituce potomka za předka)
 - např. IPrinter dává friend operátoru <<



Friend - vhodnost použití

- Typické využití pro operátory
 - vyžadují přístup k private atributům/metodám
 - zároveň nechceme zveřejňovat všem setter/getter
- Vhodné použití podporuje, nevhodné škodí zapouzdření
 - nemusíme dělat veřejné getter/setter (to je dobře)
 - čtení hodnot atributů méně problematické
 - lze samozřejmě i měnit hodnoty atributů (opatrně)
- Potenciálně rychlejší přístup k atributům
 - není třeba funkční volání

Friend - ukázka

- friendDemo.cpp
- deklarace friend pro funkci a třídu
- nefunkčnost dědění práva
- nefunkčnost transitivity
- nefunkčnost reciprocity

Psaní dobrého kódu

- Používejte friend spíše výjimečně (operátory)
 - nevhodné použití narušuje hierarchii a zapouzdření
- Speciálně nepoužívejte jenom proto, že vám to jinak nejde přeložit ☺

Přetížení operátorů

Přetížení operátorů - motivace

- „Přetížení“ operátorů znáte
 - stejný operátor se chová různě pro různé datové typy
 - / se chová rozdílně při dělení int a dělení float
 - + se chová různě pro výraz $5 + 5$ a výraz s ukazatelovou aritmetikou
 - chování pro standardní typy je definováno standardem
- V C++ můžeme deklarovat vlastní datové typy
 - třídy, struktury, typedef...
- Můžeme definovat operátory pro tyto nové typy?

Uživatelsky definované operátory

- C++ poskytuje možnost vytvoření/přetížení operátorů pro nové datové typy (typicky pro naši třídu)
- Cílem přetěžování operátorů je
 - usnadnit uživateli naší třídy její intuitivní použití
 - snížit chyby při použití třídy (my víme, jak to správně udělat)
 - např. chceme sčítat prostým $C = A + B$;
 - např. chceme vypsát prostým `cout << A`;
 - operátor by se měl chovat intuitivně správně!
- Operátor můžeme implementovat jako samostatnou funkci nebo jako metodu třídy

Přetížení operátorů - syntaxe

- Funkce/metoda, která má namísto jména
 - klíčové slovo `operator`,
 - po něm následuje označení operátoru (např. `operator +`)
- Celková syntaxe závisí na konkrétním operátoru
 - unární, binární...
 - jeho datových typech a očekávanému výsledku
 - na způsobu jeho implementace (funkce nebo metoda)
- Využijte například
 - http://en.wikipedia.org/wiki/Operators_in_C_and_C++
 - namísto `T` doplňte svůj datový typ

```
T T::operator -(const T& b) const;
```

Operator name	Syntax	Overloadable	Included in C	Prototype examples (T is any type)	
				As member of T	Outside class definitions
Basic assignment	<code>a = b</code>	Yes	Yes	<code>R T1::operator =(T2);</code>	N/A
Addition	<code>a + b</code>	Yes	Yes	<code>T T::operator +(const T& b) const;</code>	<code>T operator +(const T& a, const T& b);</code>
Subtraction	<code>a - b</code>	Yes	Yes	<code>T T::operator -(const T& b) const;</code>	<code>T operator -(const T& a, const T& b);</code>
Unary plus (NOP, but overloadable)	<code>+a</code>	Yes	Yes	<code>T T::operator +() const;</code>	<code>T operator +(const T& a);</code>

1. Operátor jako funkce

- Tzv. nečlenský operátor
- Operátor je implementován jako **samostatná funkce**
 - $A + B$, **operator**+(A, B)
 - všechny operandy musí být uvedeny v hlavičce funkce

```
T operator +(const T& first, const T& second) {  
    // Implement operator behavior  
    // Store result of addition into 'result' and return it  
    return result;  
}
```

- Použijte vždy u I/O operátorů
 - jinak nastane "obrácená" syntaxe
 - např. **operator**<< (dej na „výstup“, např. cout << a;)
 - promenna << cout namísto cout << promenna;

protože v případě *členského* operátoru je
třída první argument

Operátor jako funkce - ukázka

```
class CComplexNumber {  
    float m_realPart;  
    float m_imagPart;  
public:  
    // ...  
  
    // Make I/O operators my friends  
    friend CComplexNumber operator +(const CComplexNumber& first, const CComplexNumber& second);  
};  
  
/**  
    Addition operator as function  
    */  
CComplexNumber operator +(const CComplexNumber& first, const CComplexNumber& second) {  
    CComplexNumber result(first.m_realPart + second.m_realPart, first.m_imagPart + second.m_imagPart);  
    return result;  
}
```

povol přístup k
interním atributům

definuj operátor
jako samostatnou
binární funkci

2. Operátor jako metoda třídy

- Tzv. členský operátor
- Operátor implementován jako **metoda cílové třídy**
 - $A = B$, $A.\text{operator}=(B)$
 - první operand je automaticky `this`
- Typické pro operátory měnící vnitřní stav třídy
 - navíc operátory `->`, `=`, `()` a `[]` **musí** být jako metody třídy
 - jinak syntaktická chyba

```
T T::operator=(const T& second) {  
    // Implement operator behavior, first is this  
    // Store result of assignment into 'result' and return it  
    return result;  
}
```



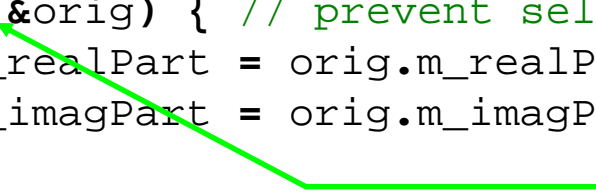
Proč vracíme výsledek?
(aby bylo možné `a=b=c;`)

Operátor jako metoda třídy - ukázka

```
class CComplexNumber {
    float m_realPart;
    float m_imagPart;
public:
    // ...

    // Operators
    CComplexNumber& operator =(const CComplexNumber& orig);
};

CComplexNumber& CComplexNumber::operator =(const CComplexNumber& orig) {
    if (this != &orig) { // prevent self-assignment
        this->m_realPart = orig.m_realPart;
        this->m_imagPart = orig.m_imagPart;
    }
    return *this;
}
```



tato část je specifická pro operátor = a
nesouvisí s operátory obecně

Který způsob kdy zvolit?

- Pokud operátor mění vnitřní stav třídy, tak většinou implementováno jako metody této třídy
 - např. operátor přiřazení **operator =**
 - např. unární operátor inkrement **operator ++**
- Pokud u operátoru nezáleží na pořadí (symetrický), většinou jako samostatné funkce
 - A+B stejně jako B+A
 - aritmetické, porovnávací...
 - např. operátor součtu **operator +**
- Pokud operátor nemůže mít jako první parametr naši třídu, pak musí být samostatná funkce
 - např. **operator <<** (např. `cout << "Hello world"`)
- Některé operátory musí být naopak metoda
 - **operator ->**, **=**, **()** a **[]**
- Zkrácené zápisy operátorů jako metoda
 - **operator +=**, **-=**, ***=** ...

Přetížení operátorů - vhodnost použití

- Motivací je snazší použití pro uživatele třídy
- Nepřehánět komplexitu operátorů
 - intuitivně očekávaná funkčnost
 - rozumná shoda funkčnosti se předdefinovanými typy
- Přetěžovat jen opravdu požadované operátory
- Raději nepřetěžovat operátory se speciálním významem
 - `" , " , "& " , "&& " , " | | "`
 - pokud přetížíme, přestane být dostupná jejich původní funkčnost
 - např. zkrácené vyhodnocování logických podmínek

Omezení pro přetěžování operátorů

- Nelze definovat nové operátory (jejich symboly)
 - jen nové implementace standardních
 - http://en.wikipedia.org/wiki/Operators_in_C_and_C++
- Ne všechny lze přetěžovat
 - nepřetížitelné operátory `::`, `.*`, `..`, `?`
- Nelze měnit počet parametrů operátoru
 - pokud je binární, přetížený také musí být binární
 - nelze `int operator+(int a);`

Omezení pro přetěžování operátorů (2)

- Nelze definovat nový význam operátorů pro předdefinované typy
 - např. nelze nové sčítání pro typ `int`
- Alespoň jeden typ musí být uživatelsky definovaný typ
 - typicky naše třída nebo struktura
 - nelze `int operator+(int a, int b);`
- Nelze měnit prioritu ani asociativitu operátorů
 - není jak

Na co myslet u přetěžování operátorů

- Přetěžujte pro všechny kombinace argumentů
 - co funguje pro `int`, to by mělo fungovat pro vaši třídu
- Prefixové vs. postfixové verze operátorů
 - `T operator++(int)` – postfix (`a++` ;)
 - `T& operator++()` – prefix (`++a` ;)
 - C++ používá „nadbytečný“ nepojmenovaný parametr typu `int` pro rozlišení prefixu() vs. postfixu(int)
- Typově konverzní operátory
 - změni typ argumentu na jiný
 - `T1::operator T2() const;`
- Pokud přetěžujete vstupní operátor, musíte sami ošetřit nečekaný výskyt konce vstupu
 - např. konec souboru souboru

Na co myslet u přetěžování operátorů (2)

- Pokud třída poskytuje aritmetický operátor a přiřazení, tak poskytněte i zkrácené operátory
 - např. poskytněte i `*=`, pokud přetížíte `*`
 - viz <http://www.cplusplus.com/reference/std/complex/complex/>
- Pokud přetěžujete relační operátory, tak všechny
 - je nepříjemné, když `==` neodpovídá `!=`
- Přetížený binární operátor není automaticky symetrický
 - záleží na pořadí argumentů
 - pokud `operator+(int, T)` tak i `operator+(T, int)`

Operátory - ukázka

- `operatorDemo.cpp`
- přetížení operátoru výstupu pomocí funkce
- přetížení operátoru + pomocí funkce
- přetížení operátoru = pomocí metody třídy

Typový systém

Typový systém obecně

- Co je typový systém
 - každá hodnota je na nejnižší úrovni reprezentována jako sekvence bitů
 - každá hodnota během výpočtu má přiřazen svůj typ
 - typ hodnoty dává sekvenci bitů význam – jak se má interpretovat
- Jsou definována pravidla
 - jak se mohou měnit typy hodnot
 - které typy mohou být použity danou operací

Typový systém v C++

- Silnější typový systém než C
- C++ je především **staticky typovaný systém**
 - typ kontrolován během překladu (static_cast)
 - umožňuje ale zjistit typ i za běhu (RTTI)
- Možnost tvorby uživatelské typové hierarchie
 - co lze a jak přetypovat (implicitně i explicitně)
 - v C je definována hierarchie pro základní datové typy (short -> int)
 - pomocí dědičnosti tříd – potomka lze přetypovat na předka

Typový systém zajišťuje

- Aby nebylo nutné přemýšlet na úrovni bitů - podpora abstrakce
- Aby se neprováděla operace nad neočekávaným typem hodnoty
- Typ proměnných může být kontrolován
 - překladačem během kompilace – staticky typovaný systém, konzervativnější
 - běhovým prostředím – dynamicky typovaný systém
- Aby bylo možné lépe optimalizovat

static_cast

- `static_cast<nový_typ>(výraz_se_starým_typem)`
 - změní typ aktuálního výrazu na jiná typ
 - `float a = static_cast<float>(10) / 3;`
- Možnost typové konverze se kontroluje při překladu
 - objekt typu A může být přetypován na B jen když je A je předek B

dynamic_cast

- `dynamic_cast<nový_typ>(výraz_se_starým_typem)`
- Typová kontrola za běhu programu
 - pokud se nepodaří, vrátí 0 (NULL)
- Využití např. pro zpětné získání typu objektu, který byl přetypován na svého předka
 - to ale často značí problém s navrženou OO hierarchií
- Run Time Type Identification (RTTI)
 - `#include <typeinfo>`
 - `typeid()`

Statické metody

- Klíčové slovo `static`
- Metodu lze volat, aniž by existovala instance třídy
 - `A::metodaStatic()`
- Static metoda nemá jako první parametr `this`
- Static metodu nelze udělat virtuální
- Static metoda nemůže přistupovat k atributům třídy ani k jiným nestatickým metodám
 - protože objekt nemusí existovat
 - všechny vstupní data musí být jako parametry

Ukázky přetypování

- Třidu A a B budeme používat v dalších ukázkách

```
class A {  
protected:  
    int m_value;  
public:  
    void setValue(int value) {  
        m_value = value;  
        cout << "A::setValue() called" << endl;  
    }  
  
    virtual int getValue() const {  
        cout << "A::getValue() called" << endl;  
        return m_value;  
    }  
  
    static void printValue(int value) {  
        cout << "Value = " << value << endl;  
        cout << "A::printValue() called" << endl;  
    }  
};
```

```
class B : public A {  
public:  
    void setValue(int value) {  
        m_value = value;  
        cout << "B::setValue() called" << endl;  
    }  
  
    virtual int getValue() const {  
        cout << "B::getValue() called" << endl;  
        return m_value;  
    }  
  
    static void printValue(int value) {  
        cout << "Value = " << value << endl;  
        cout << "B::printValue() called" << endl;  
    }  
};
```

Ukázky přetypování - reference

```
int main() {
    A objectA;
    B objectB;

    // Class A methods
    objectA.setValue(10);
    objectA.getValue();
    objectA.printValue(15);
    A::printValue(16); // Can be called even when no object A exists

    // Class B methods
    objectB.setValue(10);
    objectB.getValue();
    B::printValue(16);

    // Retype B to A via reference
    A& refB = objectB;
    refB.setValue(10); // from A
    refB.getValue();   // from B (virtual)
    refB.printValue(15); // from A (static)
    return 0;
}
```

Ukázky přetypování - ukazatele

```
// Retype B to A via pointers (compile time)
A* pObject = new B;
pObject->setValue(10); // from A
pObject->getValue();    // from B (virtual)
pObject->printValue(15); // from A (static)

// Retype pObject (type A) to type B during runtime
B* pObjectB = dynamic_cast<B*> (pObject);
pObjectB->setValue(10); // from B
pObjectB->getValue();    // from B (virtual)
pObjectB->printValue(15); // from B (static)

// Try to retype pObject (type A) to type C during runtime
// Will return NULL during runtime as retype A to C is not allowed
C* pObjectC = dynamic_cast<C*> (pObject);
if (pObjectC) pObjectC->setValue(10);
else cout << "Retype A to C not allowed" << endl;
// Warning: SIGSEV, if you will not test pObjectC
```

Obcházení typového systému

- Céčkové přetypování umožňuje obcházet typový systém
 - přetypuj pole `uchar` na pole `int` a proved' xor
- C++ také umožňuje obcházet
 - céčkové přetypování a `reinterpret_cast()`
 - z důvodů rychlosti, nepoužívejte (pokud nemusíte)
- Implicitní vs. explicitní typová konverze
 - implicitní konverze dělá automaticky překladač
 - `void foo(int a); short x; foo(x);`
 - explicitní specifikuje programátor
 - `float a = (float) 10 / 3; // verze z C`
 - `float a = static_cast<float>(10) / 3;`

reinterpret_cast

- **reinterpret_cast<nový_typ>(výraz_starý_typ)**

- změni datový typ bez ohledu na typové omezení
- funguje analogicky jako union – přetypování na bitové úrovni

```
// Computation speed up with reinterpret_cast
const int ARRAY_LEN = 80;
unsigned char* byteArray = new unsigned char[ARRAY_LEN];
for (int i = 0; i < ARRAY_LEN; i++) byteArray[i] = i;
// xor array with 0x55 (01010101 binary)
// 80 iterations required
for (int i = 0; i < ARRAY_LEN; i++) byteArray[i] ^= 0x55;
// retype to unsigned integers
unsigned int* intArray = reinterpret_cast<unsigned int*>(byteArray);
// only 20 iterations required (x86 version - unsigned int is 4 bytes)
for (unsigned int i = 0; i < ARRAY_LEN / sizeof(unsigned int); i++)
    intArray[i] ^= 0x55555555;
// only 10 iterations required (x64 version - unsigned int is 8 bytes)
// NOTE: xor value must be expanded accordingly (e.g., not 0x55 but 0x55555555 for x86)
```

využijeme celé
šířky architektury
x86 je 32bit

const_cast

- “Odstraní” modifikátor `const` z datového typu
 - můžeme následně měnit data a volat `ne-const` metody objektu
- `const_cast<nový_typ>(výraz_starý_typ)`
 - `výraz_starý_typ` musí být ukazatel nebo reference

```
const A* pConstObjectA = new A;  
//pConstObjectA->setValue(10); // error: no const method available  
A* pNonConstObjectA = const_cast<A*> (pConstObjectA);  
pNonConstObjectA->setValue(10); // now we can call non-const
```

- Nepoužívá se zcela běžně!
 - používá se, pokud cizí kód (který nemůžeme modifikovat) neposkytuje `const` metody a my máme `const` objekt

Shrnutí

- Kontejnery + iterátory + algoritmy
 - ušetří hodně práce
- Právo friend – používat opatrně
- Přetěžování operátorů – mocné, ale používat přiměřeně
- Typový systém
 - přetypování – snažte se používat statickou kontrolu