

PB161 – Programování v jazyce C++

Objektově Orientované Programování

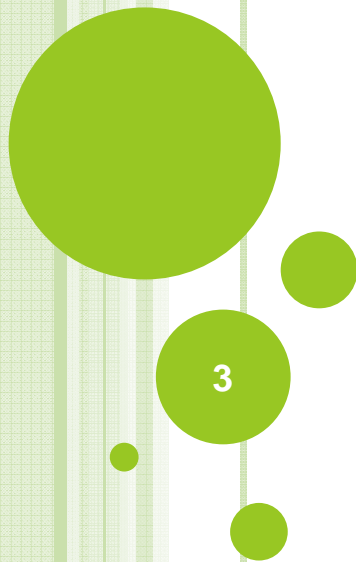
Unit testing, STL algoritmy, friend, přetěžování operátorů

UNIT TESTING – ONDRA BOUDA



2

STL ALGORITHM



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 ALGORITHMY

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

Non-modifying sequence operations:

for_each	Apply function to range (function template)
find	Find value in range (function template)
find_if	Find element in range (function template)
find_end	Find last subsequence in range (function template)
find_first_of	Find element from set in range (function template)
adjacent_find	Find equal adjacent elements in range (function template)
count	Count appearances of value in range (function template)
count_if	Return number of elements in range satisfying condition (function template)
mismatch	Return first position where two ranges differ (function template)
equal	Test whether the elements in two ranges are equal (function template)
search	Find subsequence in range (function template)
search_n	Find succession of equal values in range (function template)

Modifying sequence operations:

copy	Copy range of elements (function template)
copy_backward	Copy range of elements backwards (function template)
swap	Exchange values of two objects (function template)
swap_ranges	Exchange values of two ranges (function template)
iter_swap	Exchange values of objects pointed by two iterators (function template)
transform	Apply function to range (function template)
replace	Replace value in range (function template)
replace_if	Replace values in range (function template)
replace_copy	Copy range replacing value (function template)
replace_copy_if	Copy range replacing value (function template)
fill	Fill range with value (function template)
fill_n	Fill sequence with value (function template)
generate	Generate values for range with function (function template)
generate_n	Generate values for sequence with function (function template)
remove	Remove value from range (function template)
remove_if	Remove elements from range (function template)
remove_copy	Copy range removing value (function template)
remove_copy_if	Copy range removing values (function template)
unique	Remove consecutive duplicates in range (function template)
unique_copy	Copy range removing duplicates (function template)
reverse	Reverse range (function template)
reverse_copy	Copy range reversed (function template)
rotate	Rotate elements in range (function template)
rotate_copy	Copy rotated range (function template)
random_shuffle	Rearrange elements in range randomly (function template)
partition	Partition range in two (function template)
stable_partition	Partition range in two - stable ordering (function template)

Převzato z<http://www.cplusplus.com/reference/algorithm>

||

Sorting:

sort	Sort elements in range (function template)
stable_sort	Sort elements preserving order of equivalents (function template)
partial_sort	Partially Sort elements in range (function template)
partial_sort_copy	Copy and partially sort range (function template)
nth_element	Sort element in range (function template)

Binary search (operating on sorted ranges):

lower_bound	Return iterator to lower bound (function template)
upper_bound	Return iterator to upper bound (function template)
equal_range	Get subrange of equal elements (function template)
binary_search	Test if value exists in sorted array (function template)

Merge (operating on sorted ranges):

merge	Merge sorted ranges (function template)
inplace_merge	Merge consecutive sorted ranges (function template)
includes	Test whether sorted range includes another sorted range (function template)
set_union	Union of two sorted ranges (function template)
set_intersection	Intersection of two sorted ranges (function template)
set_difference	Difference of two sorted ranges (function template)
set_symmetric_difference	Symmetric difference of two sorted ranges (function template)

Heap:

push_heap	Push element into heap range (function template)
pop_heap	Pop element from heap range (function template)
make_heap	Make heap from range (function template)
sort_heap	Sort elements of heap (function template)

Min/max:

min	Return the lesser of two arguments (function template)
max	Return the greater of two arguments (function template)
min_element	Return smallest element in range (function template)
max_element	Return largest element in range (function template)
lexicographical_compare	Lexicographical less-than comparison (function template)
next_permutation	Transform range to next permutation (function template)
prev_permutation	Transform range to previous permutation (function template)

||

Převzato z

<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;
```

```
}
```

function template

std::find

<algorithm>

```
template <class InputIterator, class T>
    InputIterator find ( InputIterator first, InputIterator last, const T& value );
```

Find value in range

Returns an iterator to the first element in the range [first,last) that compares equal to *value*, or *last* if not found.

The behavior of this function template is equivalent to:

```
1 template<class InputIterator, class T>
2     InputIterator find ( InputIterator first, InputIterator last, const T& value )
3     {
4         for ( ;first!=last; first++) if ( *first==value ) break;
5         return first;
6     }
```

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;
}
```

for_each může také modifikovat kontejner
Pokud je hodnota předávána referencí
print(int& value)
Pro modifikaci používejte ale raději **transform**

```
int main() {
    std::list<int> myList;
    // ... fill something into myList
    // Apply function to range (typically 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)
- Může být lambda (**C++1**, pozdější přednáška)

STL ALGORITHM – 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

14

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

- Přístup povoluje ten, k němuž bude přístupováno

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, tranzitivní 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/třídu 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
 - ale tohle typicky optimalizuje překladač automaticky pomocí inline funkcí (vlození těla funkce namísto jejího volání)

PSANÍ DOBRÉHO KÓDU

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

FRIEND - UKÁZKA

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

PŘETÍŽENÍ OPERÁTORŮ

22

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
 - preferujte variantu se samostatnou funkcí, viz. dále

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 implemetace (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 operator my friend  
    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**
 - $A = B$, $A.operator = (B)$
 - první operand je automaticky `this`
- Typické pro operátory měnící vnitřní stav třídy
 - navíc operátory `->`, `=`, `()`, `(typ)` 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;  
}
```



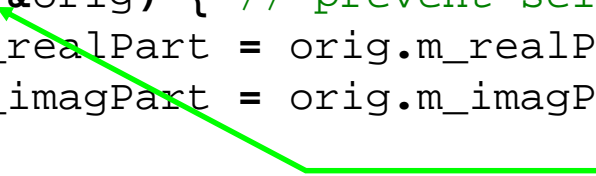
Proč vrátíme výsledek?
(aby bylo možné `a=b=c;`)

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

KTERÝ ZPŮSOB KDY ZVOLIT? (2)

- Většinu operátorů se vyplatí implementovat jako samostatné funkce
- Získáme následující výhodu
 - operátor ?? je přetížený pro třídu A.
 - třída B se umí implicitně přetypovat na A (potomek)
 - při zavolání ??b (b je typu B) dojde k přetypování b na typ A
 - využije se implementace operátoru pro třídu A
- Při implementaci operátoru ?? jako členské metody třídy A není toto chování možné

PŘETÍŽENÍ OPERÁTORŮ - VHODNOST POUŽITÍ

- Motivací je snažší použití pro uživatele třídy
- Nepřehánět komplexitu operátorů
 - intuitivně očekávaná funkčnost
 - rozumná shoda funkčnosti se předefinovanými 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
 - např. pokud je binární, přetížený také musí být binární
 - např. nelze `int operator+(int a);`
 - výjimkou je operátor funkčního volání (různý počet argumentů)

OMEZENÍ PRO PŘETĚŽOVÁNÍ OPERÁTORŮ (2)

- Nelze definovat nový význam operátorů pro předdefinované typy
 - např. nelze nové sčítání pro typ `int`
 - je již definováno, přepisování existujících funkcí by způsobilo zmatek
- 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 (protože je to funkce)
 - pokud `operator+(int, T)` tak i `operator+(T, int)`
- Pravidla pro přetěžování operátorů
 - <http://www.comp.dit.ie/bduggan/Courses/OOP/cpp-ops.html>

PŘETÍŽENÍ OPERÁTORU << - UKÁZKA

```
#include <iostream>
using std::cout;
using std::endl;
using std::ostream;

class CComplexNumber {
    float m_realPart;
    float m_imagPart;
public:
    // ...
    // Make some operators my friends
    friend ostream& operator <<(ostream& out, const CComplexNumber& complex);
};

/**
    Output operator as friend function
    */
ostream& operator <<(ostream& out, const CComplexNumber& complex) {
    out << "[" << complex.m_realPart << ", " << complex.m_imagPart << "];"
    return out;
}

int main() {
    CComplexNumber    value1(10, 20);
    cout << value1 << endl;
    return 0;
}
```

PŘETÍŽENÍ OPERÁTORU >> - UKÁZKA

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
using std::istream;

class CComplexNumber {
    float m_realPart;
    float m_imagPart;
public:
    // ...
    // Make some operators my friends
    friend istream& operator >>(istream& in, CComplexNumber& complex);
};

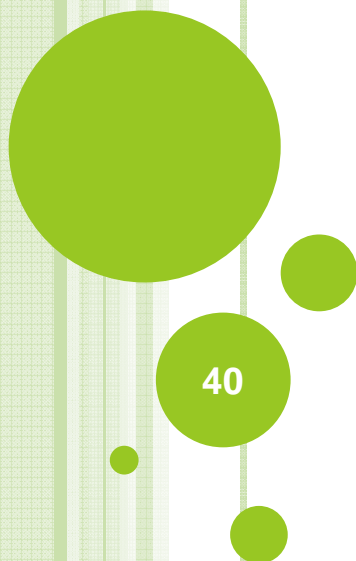
/** Input operator as friend function */
istream& operator >>(istream& in, CComplexNumber& complex) {
    if (in.good()) {
        in >> complex.m_realPart;
        in >> complex.m_imagPart;
    }
    return in;
}
```

```
int main() {
    CComplexNumber    value1(10, 20);
    cout << value1 << endl;
    cin >> value1;
    cout << value1 << endl;
    return 0;
}
```

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

- Analogie Céčkového $A = (\text{nový_typ}) B$; pro přetypování při překladu
- `static_cast<nový_typ>(výraz_se_starým_typem)`
 - změni 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
- Používejte namísto céčkového (`nový_typ`)
 - Céčkové přetypování nerozlišuje mezi přetypování během kompilace, za běhu...
 - lze snadno hledat všechny přetypování v kódu ("`static_cast<`")
 - <http://stackoverflow.com/questions/103512/in-c-why-use-static-castintx-instead-of-intx>

DYNAMIC_CAST

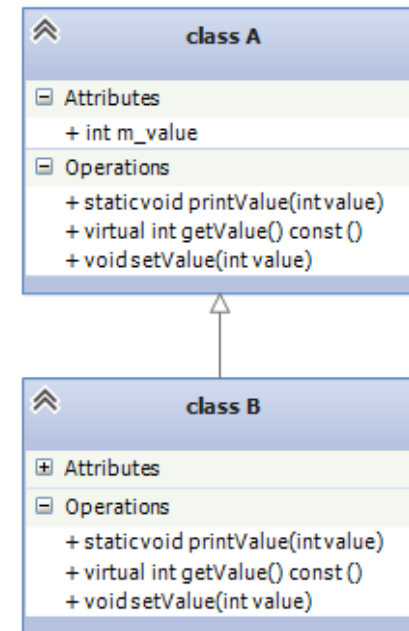
- `dynamic_cast<nový_typ>(výraz_se_starým_typem)`
- Typová kontrola za běhu programu
 - pokud se nepodaří, vrátí 0 (NULL)
- Využití např. pro zpětné získání typu objektu, který byl přetypován na svého předka
 - to ale často značí problém s navrženou OO hierarchií
- Run Time Type Identification (RTTI)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

- 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;
    }
};
```

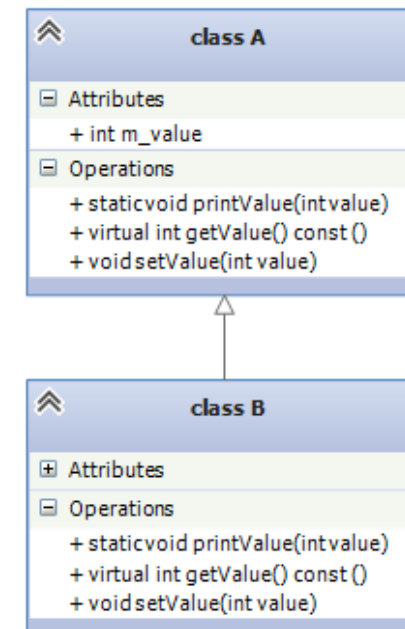
UKÁZKY PŘETÝPOVÁNÍ – REFERENCE

```
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;
}
```

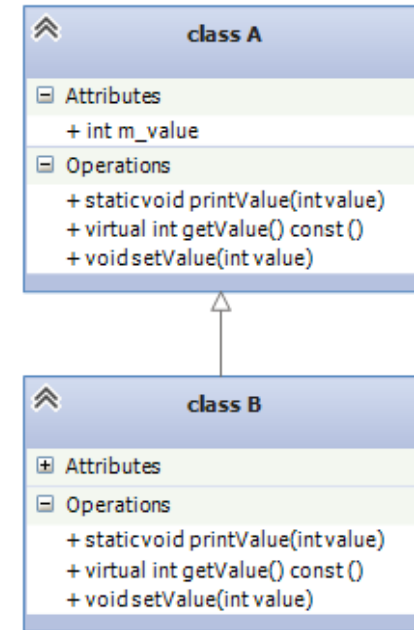


UKÁZKY PŘETÝPOVÁNÍ – UKAZATELE

```
// 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

```
// 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 `non-const` metody objektu
- `const_cast<nový_typ>(výraz_starý_typ)`

```
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

ŠABLONY

FUNKČNÍ OBJEKTY

NÁVRHOVÉ PRINCIPY

NÁVRHOVÉ VZORY

FUNKČNÍ OBJEKT - MOTIVACE

- Z C/C++ jsou známy tzv. callback funkce

- pomocí funkčního ukazatele je možné předat jako parametr ukazatel na funkci, která je později zavolána
- používá se například u STL algoritmů

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

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

```
template < class InputIterator, class OutputIterator, class UnaryOperator >
OutputIterator transform ( InputIterator first1, InputIterator last1,
                           OutputIterator result, UnaryOperator op )
{
    while (first1 != last1)
        *result++ = op(*first1++); // or: *result++=binary_op(*first1++,*first2++);
    return result;
}
```

- Co když chceme přičítat pomocí `std::transform` číslo 10?

- vytvoříme funkci `int add10(int value){return value+10;}`
- `std::transform(l.begin(), l.end(), l.begin(), add10);`

- Co když chceme přičítat libovolné číslo?



FUNKČNÍ OBJEKT – FUNCTOR

- Objekt, který může zavolán, jako by to byla běžná funkce
 - dosáhneme pomocí přetíženého operátoru `()`
 - `A prom; prom(5);`
- Operátor `()` musíme přetížit pro všechny potřebné typy a počty argumentů
 - tj. pro každou funkci, kterou chceme functorem nahradit
- Pokud se pokusíme použít objekt jako functor a neexistuje odpovídající přetížený operátor `()`
 - `error: no match for call to '(typ_objektu) (typ_argumentů)'`
 - *`stl_algo.h:4688:2: error: no match for call to '(A) (int&)'`*

FUNCTOR - UKÁZKA

```
#include <iostream>
#include <list>
#include <algorithm>

class CAddX {
    int m_whatAdd;
public:
    CAddX(int whatAdd) : m_whatAdd(whatAdd) {}
    void set(int whatAdd) { m_whatAdd = whatAdd; }
    int operator () (int value) {
        return value + m_whatAdd;
    }
};
```

```
int main() {
    std::list<int> myList;

    myList.push_back(1);  myList.push_back(2);
    myList.push_back(3);  myList.push_back(4);

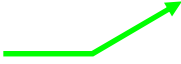
    CAddX variableAdder(3);
    // variableAdder is functor object, now adding number 3
    transform(myList.begin(), myList.end(), myList.begin(), variableAdder);
    // change added value
    variableAdder.set(101);
    transform(myList.begin(), myList.end(), myList.begin(), variableAdder);

    return 0;
}
```

Nyní přičítač trojky



Nyní přičítač stojedničky



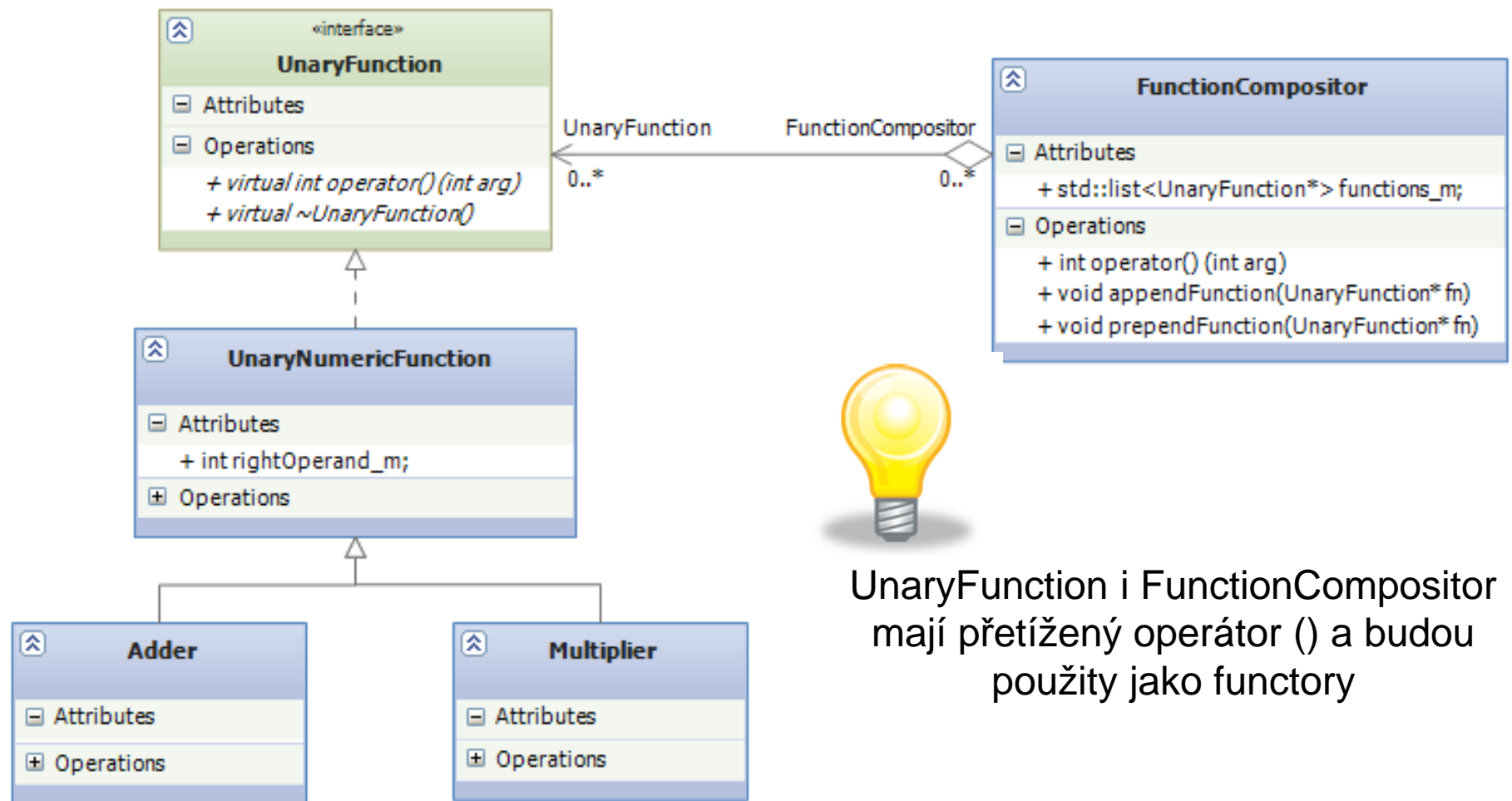
FUNCTORY – VHODNOST POUŽITÍ

- Výhodou je možnost uchovávat (a měnit) stav functoru
 - u funkcí bychom museli použít globální proměnné
- Functor může být výrazně rychlejší než funkce
 - překladač může vložit kód functoru namísto volání
- u funkčních ukazatelů (callback) není možné
- Využívá myšlenek z funkcionálním programování
 - současné imperativní jazyky mají podporu pro některé funkční prvky
 - functor v C++, lambda v C++11, delegates v C#...

FUNCTOR A TESTOVACÍ ZÁPOČTOVÝ PŘÍKLAD

- Testovací zápočtový příklad
 - http://cecko.eu/public/pb161_cviceni#prednaskacviceni_tyden_21-27112011
- Aplikace série funkcí na zadaný argument
- Definice vhodného rozhraní a využití dědičnosti
- (autor původního kódu Petr Pilař)

HIERARCHIE



```

class UnaryFunction {
public:
    virtual ~UnaryFunction() {}
    virtual int operator() (int arg) const = 0;
};

```

Přetížíme operátor () pro aplikaci unární funkce

```

class UnaryNumericFunction : public UnaryFunction {
public:
    UnaryNumericFunction(int rightOperand) : rightOperand_m(rightOperand) {}

    virtual int rightOperand() const { return rightOperand_m; }
    virtual void setRightOperand(int r) { rightOperand_m = r; }

```

```

private:
    int rightOperand_m;
};

```

Numerická unární funkce je functor s parametrizací v rightOperand_m

```

class Adder : public UnaryNumericFunction {
public:
    Adder(int r) : UnaryNumericFunction(r) {}
    int operator() (int arg) const { return arg + rightOperand(); }
};

```

Parametrizace functoru

```

class Multiplier : public UnaryNumericFunction {
public:
    Multiplier(int r) : UnaryNumericFunction(r) {}
    int operator() (int arg) const { return arg * rightOperand(); }
};

```

V přetíženém operátoru () přičítáme

```
// Function for deallocation
void delete_ptr(UnaryFunction* ptr) { delete ptr; }

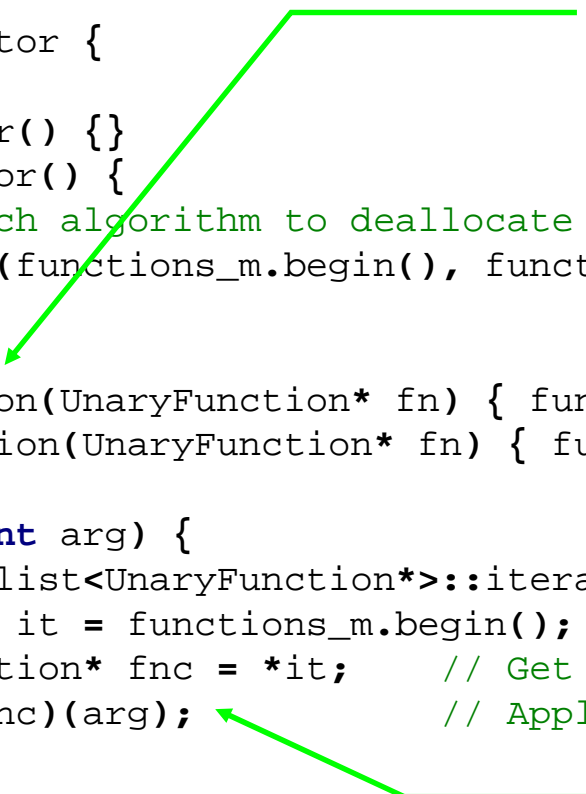
class FunctionCompositor {
public:
    FunctionCompositor() {}
    ~FunctionCompositor() {
        // Use for_each algorithm to deallocate separate functions
        std::for_each(functions_m.begin(), functions_m.end(), delete_ptr);
    }

    void appendFunction(UnaryFunction* fn) { functions_m.push_back(fn); }
    void prependFunction(UnaryFunction* fn) { functions_m.push_front(fn); }

    int operator() (int arg) {
        typedef std::list<UnaryFunction*>::iterator iterator;
        for (iterator it = functions_m.begin(); it != functions_m.end(); ++it) {
            UnaryFunction* fnc = *it;    // Get particular function
            arg = (*fnc)(arg);           // Apply it on argument
        }

        return arg;
    }

private:
    std::list<UnaryFunction*> functions_m;
};
```



**Parametrizace functoru
probíhá přidáváním
unárních funkcí**

**Aplikací functoru pomocí
operátoru () postupně
zavoláme všechny
obsažené funkce**

POUŽITÍ FUNCTORŮ

```
#include <iostream>
#include <algorithm>
#include <list>
```

```
int main() {
    FunctionCompositor comp;
    comp.prependFunction(new Adder(10));
    comp.prependFunction(new Multiplier(2));
    comp.prependFunction(new Adder(-5));
    comp.prependFunction(new Adder(2));

    int number = 0;
    std::cin >> number;
    std::cout << comp(number) << std::endl;
}
```

Vytváříme functor comp

Vytváříme functor typu
Adder (a další) a
vkládáme do comp

Aplikujeme functor comp

SHRNUTÍ

- Kontejnery + iterátory + algoritmy
 - ušetří hodně práce
 - méně kódu, čitelnější zápis, pravděpodobně rychlejší
- Právo friend – používat opatrně
- Přetěžování operátorů – mocné, ale používat přiměřeně
- Typový systém
 - snažte se používat statickou typovou kontrolu