

PB161 – 6. přednáška (26. října 2015)

Standardní knihovna C++

- obsahuje řadu částí, některé už jsme viděli (strings, I/O streams)
- mimo jiné obsahuje i knihovní funkce ze standardní knihovny jazyka C
 - jiné hlavičkové soubory – místo `<xxx.h>` máme `<cxxx>`, např. `<cstdlib>` místo `<stdlib.h>`
 - mnohé věci se dají v C++ dělat jinak a lépe (obvykle lepší typová kontrola, někdy snazší použití) – např. `<iostream>` místo `<cstdio>`
- velká část standardní knihovny používá generické programování (šablony)
 - o šablonách si podrobně řekneme jindy
 - dnes uvidíme syntax stylu: `container<type>`, což reprezentuje typ generického kontejneru `container`, který obsahuje objekty typu `type`
 - možná už jste si při práci s řetězci všimli (např. z chybových hlášek), že `string` je ve skutečnosti zkratka za `basic_string<char>`
- dnes se budeme bavit o částech standardní knihovny, které definují:
 - kontejnery (generické objekty, které mohou obsahovat jiné objekty)
 - iterátory (chytré ukazatele, pomocí kterých umíme kontejnery procházet)
 - algoritmy (generické operace, které je možno provádět s kontejnery nebo jejich částmi)
- standardní knihovna je rozsáhlá; používejte dokumentaci
 - <http://cppreference.com>
 - <http://cplusplus.com/reference>
- nezapomeňte, že standardní knihovna používá jmenný prostor `std`
 - v následujícím textu to vynecháváme, ale mějte na paměti, že když píšeme `vector`, `pair`, `map`, máme ve skutečnosti na mysli `std::vector`, `std::pair`, `std::map` apod.

Kontejnery & iterátory

Motivace pro kontejnery

- data je většinou třeba uchovávat ve složitějších datových strukturách
- různé druhy dat potřebují podporovat různé druhy operací (vkládání na konec, vkládání doprostřed, vyhledávání, ...)
- není obecně jedna ideální datová struktura, která by podporovala všechno a s rozumnou efektivitou
- na druhou stranu, uspořádání datové struktury je často nezávislé na konkrétním typu dat (pole čísel a pole ukazatelů mají podobný způsob uložení v paměti) a základní operace jsou shodné
- proto dává smysl používat generické kontejnery, které fungují nezávisle na tom, co se do nich ukládá

Typy kontejnerů

- sekvenční: reprezentují posloupnosti dat, umožňují sekvenční procházení
 - `array` (C++11) – klasické pole ve stylu C (souvislý blok paměti, pevná velikost), ale s výhodami kontejneru v C++ (stejně operace jako pro ostatní kontejnery, přístup s kontrolou mezí apod.)
 - `vector` – tzv. dynamické pole, stále jde o souvislý blok paměti, ale má možnost se za běhu zvětšovat
 - `deque` – obousměrná fronta, umožňuje rychlé přidávání/odebírání prvků na obou koncích
 - `forward_list` (C++11), `list` – jednostranně/oboustranně zřetěžený seznam
- asociativní: umožňují rychlé vyhledávání
 - `set` – množina (uspořádaná)
 - `map` – tzv. asociativní pole nebo slovník, množina záznamů ve tvaru (klíč, hodnota), uspořádaná podle klíče
 - `multiset`, `multimap` – umožňují opakování klíčů
 - `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap` – neuspořádané verze (typicky implementované pomocí hashovacích tabulek), zavedeny v C++11
- adaptéry: různá rozhraní pro sekvenční kontejnery
 - `stack` (zásobník), `queue` (fronta), `priority_queue` (prioritní fronta)
- datové struktury pro dvojice, trojice, ... typů
 - nepatří mezi kontejnery, mají jiné operace
 - `pair` – drží dva objekty různých (nebo stejných) typů
 - `tuple` (C++11) – drží fixní počet objektů různých (nebo stejných) typů
- řetězece `string`
 - nejsou to generické kontejnery, ale funguje pro ně řada konceptů stejně jako pro kontejnery

Dvojice

- `pair` má dva šablonové parametry (typ prvního objektu, typ druhého objektu)
- přístup k prvnímu objektu pomocí atributu `first`, k druhému pomocí atributu `second`
- pomocná funkce `make_pair`

```
pair<int, char> p1;
p1.first = 17;
p1.second = 'a';
pair<int, char> p2(p1);           // kopírovací konstruktor
pair<int, float> p3(42, 3.14);   // inicializační konstruktor

p3 = p1; // kopírovací přiřazení, navíc zde dochází k implicitní konverzi
         // char na float
```

```
pair<string, int> psi;
psi = make_pair("James Bond", 7);
// je totéž jako
psi = std::pair<const char *, int>("James Bond", 7);
```

- objekty typu `pair` se dají porovnávat pomocí `==`, `<` apod.
 - pokud to podporují objekty uvnitř
 - porovnávání je lexikografické

Sekvenční kontejner `vector`

- dynamické pole, tj. pole, které se umí zvětšovat
- prvky ve `vectoru` uloženy v souvislém úseku paměti
- rychlost přístupu stejná jako běžné pole
- rychlé přidávání na konec (rezervovaná kapacita)
 - při zaplnění vyhrazené paměti se alokuje větší kus paměti a objekty se do nového pole zkopírují (zamyslete se nad tím)
 - typicky zvětšování na dvojnásobek (jsou i jiné možnosti)
 - konstantní *amortizovaná* složitost
- podporuje i vkládání na jiná místa
 - pomalejší, vyžaduje přesouvání prvků
 - stále dostatečně rychlé pro malé až střední vektory
- vhodný pro velkou řadu použití
 - měla by to být vaše první volba, pokud nemáte nějaký zvláštní důvod pro použití `deque` nebo zřetěženého seznamu (pozor, zřetěžený seznam je velmi, velmi pomalý)

Syntaxe: `vector<typ>`

```
vector<int> vec;    // dynamické pole celých čísel
vec.push_back(17); // vložení prvku na konec
vec.push_back(21);
vec.push_back(0);

for (int i = 0; i < vec.size(); ++i) { // zjištění aktuální velikosti
    cout << vec[i] << endl; // přístup pomocí operátoru []
}

cout << vec[9] << endl; // zřejmě dojde k nějaké chybě, nehlídaný přístup
cout << vec.at(9) << endl; // hlídaný přístup, vyhodí výjimku
                        // std::out_of_range

vec.pop_back(); // odstranění prvku z konce

if (vec.empty()) { /* ... */ } // test prázdnosti
// toto je lepší než vec.size() == 0, proč?
```

- používání metody `at()` je dražší (pomalejší) než použití operátoru `[]`

- nepoužívejte, když kontrola mezí nedává smysl

```
for (int i = 0; i < vec.size(); ++i) {
    cout << vec.at(i) << endl; // zbytečné! víme, že i je v mezích
}
```

- rezervování kapacity, zvětšování vektoru

```
vector<int> vec;
vec.reserve(100); // vec má rezervovanou paměť pro 100 intů
                  // vec stále neobsahuje žádné prvky

vec.push_back(17);
vec.push_back(20);

// vec.size() je 2, vec.capacity() je stále 100

vec.resize(99);

// vec.size() je 99, vec.capacity() je 100
// vec obsahuje prvky 17, 20, 0, 0, ..., 0 (celkem 99 prvků)

vec.push_back(-7);
vec.push_back(9);

// vec.size() je 101, vec.capacity() je víc než 100 (typicky 200)
// vec obsahuje prvky 17, 20, 0, 0, ..., 0, -7, 9 (celkem 101 prvků)
```

- vektory (stejně jako jiné kontejnery) lze kopírovat
 - volá se kopírovací konstruktor jednotlivých prvků

Motivace pro iterátory

- základní myšlenka: po klasickém poli se můžeme pohybovat jednak pomocí indexů, jednak pomocí ukazatelů
- iterátory jsou „inteligentní ukazatele“
- různé druhy iterátorů podle typu kontejneru
 - typicky podporují sekvenční procházení alespoň v jednom směru
 - různé další operace
- různé metody kontejnerů berou nebo vrací iterátory
 - minimálně `begin()` a `end()`
 - `begin()` vrací iterátor na začátek kontejneru
 - `end()` vrací iterátor za konec kontejneru

Iterátory pro *vector*

- můžeme si představovat jako ukazatele do pole

Syntax:

`vector<typ>::iterator` (pro čtení i zápis)

`vector<typ>::const_iterator` (jen pro čtení)

```
vector<int> vec;
```

```
// naplnění vektoru nějakými daty
```

```
vec.push_back(10);
```

```
vec.push_back(40);
```

```
vec.push_back(20);
```

```
vec.push_back(30);
```

```
vector<int>::iterator vecIt = vec.begin();
```

```
// vecIt je teď iterátor, který ukazuje na první prvek vektoru vec
```

```
// dereference jako u ukazatelů
```

```
cout << *vecIt << endl; // vypíše 10
```

```
++vecIt; // posunutí iterátoru na další prvek
```

```
cout << *vecIt << endl; // vypíše 40
```

```
--vecIt; // posunutí iterátoru na předchozí prvek
```

```
// použití pro procházení vektoru
```

```
for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
```

```
    cout << *it << endl;
```

```
    *it = 17; // můžeme prvky vektoru měnit
```

```
}
```

```
// vec.end() ukazuje ZA poslední prvek vektoru
```

- v případě, že se vektor zvětší, přestávají iterátory být platné

Odbočka – užitečné zkratky pro iterování v C++11

- v C++11 můžeme nechat kompilátor sám dedukovat typ objektu z inicializace
 - typový specifikátor **auto**
 - používat opatrně, nadměrné používání zhoršuje čitelnost kódu
 - hodí se pro iterátory, ušetří nám to psaní dlouhého typu

```
for (auto it = vec.begin(); it != vec.end(); ++it) {
```

```
    cout << *it << endl;
```

```
}
```

- v C++11 máme novou formu cyklu **for**

– tzv. range-based for

```
for (int element : vec) {
    cout << element << endl;
}

// přeloží se nějak jako (nepřesně; podrobnosti viz dokumentace)
for (auto it = vec.begin(); it != vec.end(); ++it) {
    int element = *it;
    cout << element << endl;
}
```

- pokud chceme prvky kontejneru měnit, musíme brát prvek referencí

```
for (int & element : vec) {
    cout << element << endl;
    element += 17;
}

// přeloží se nějak jako (nepřesně; podrobnosti viz dokumentace)
for (auto it = vec.begin(); it != vec.end(); ++it) {
    int & element = *it;
    cout << element << endl;
    element += 17;
}
```

Asociativní kontejner *set*

- reprezentuje uspořádanou množinu
 - prvky se neopakují
 - prvky se dají uspořádat
- typická implementace pomocí nějakého druhu vyvážených vyhledávacích stromů (často red-black trees)
- rychlé ($O(\log n)$) vkládání, mazání, vyhledávání
- iterátory jsou vždy konstantní (neměnitelné)

```
set<int> s;
s.insert(17); // vložení prvku do množiny
s.insert(25);
s.insert(9);
s.insert(-4);

s.erase(17); // odebere prvek z množiny
s.erase(11); // nic se nestane
```

```
if (s.empty()) { /* ... */ } // test prázdnosti
```

```
cout << s.size() << endl; // vypíše 3
```

- vyhledávání vrací iterátor
 - iterátor na prvek, pokud existuje
 - iterátor `end()`, pokud ne

```
set<int>::iterator it = s.find(17);
```

```
if (it != s.end()) {  
    // množina obsahuje 17  
    // můžeme používat *it, ale jen ke čtení, ne k zápisu  
} else {  
    // množina neobsahuje 17  
}
```

- `erase` rovněž může brát iterátor jako vstup, vrací pak iterátor na další prvek

```
// smazání všech lichých prvků z množiny  
for (auto it = s.begin(); it != s.end(); ) {  
    if ( *it % 2 == 1 ) {  
        it = s.erase(it);  
    } else {  
        ++it;  
    }  
}
```

- `insert` vrací dvojici (`pair`) iterátor na vložený (nebo už existující) prvek a hodnotu typu `bool` určující, jestli vkládání skutečně proběhlo (`false` znamená, že prvek už v množině byl)

```
pair< set<int>::iterator, bool > result = s.insert(-4);
```

```
if (result.second) {  
    // vložení proběhlo  
} else {  
    // prvek -4 už v množině byl  
}
```

Asociativní kontejner *map*

- (asociativní pole, slovník)
- reprezentuje mapování klíčů (keys) na hodnoty
- uspořádaný podle klíčů

- klíče se nemohou opakovat
- dá se představit jako množina jejíž prvky jsou dvojice (klíč, hodnota), přičemž porovnávání a uspořádání se dívá pouze na klíč

Příklad: mapování UČO → jméno (vektor by zde byl neefektivní, chceme-li reprezentovat jen pár lidí, bylo by zbytečné alokovat paměť pro milion záznamů)

```
map<int, string> cppTeachers;

cppTeachers.insert(make_pair(72525, "Nikola Benes")); // přidání záznamu
cppTeachers.insert(make_pair(374154, "Jiri Weiser"));

cppTeachers.erase(374154); // odebrání záznamu s klíčem 374154
cppTeachers.erase(4085);   // nic se nestane
```

- v C++11 je místo `make_pair` možno použít i novou inicializační syntax:

```
cppTeachers.insert( {72525, "Nikola Benes"} );
```

- vyhledávání vrací iterátor
 - iterátor ukazuje na záznam, tj. dvojici (klíč, hodnota)
 - záznam je typu `pair<key, value>`
 - klíč se nesmí modifikovat, hodnota ano

```
map<int, string>::iterator it = cppTeachers.find(72525);
```

```
if (it != cppTeachers.end()) {
    // cppTeachers obsahuje záznam s klíčem 72525
    cout << it->first << ": " << it->second << endl;
    // vypíše 72525: Nikola Benes
    it->second += ", Ph.D.";
    // it->first nelze použít k zápisu
} else {
    // cppTeachers neobsahuje záznam s klíčem 72525
}
```

- speciální syntax používá přetížený operátor `[]`
 - uvnitř operátoru `[]` je klíč
 - pokud záznam neexistuje, automaticky se vytvoří
 - používejte opatrně (dávejte přednost spíš `insert` a `find`)
 - narozdíl od `insertu` umožňuje měnit hodnotu už existujícího klíče

```
map<string, int> namesToUCO;

namesToUCO["Nikola Benes"] = 72525;
```



```

// přístup k neexistujícímu záznamu jej vytvoří
if (namesToUCO["James Bond"] == 7) { /* ... */ }

map<string, int>::iterator it = namesToUCO.find("James Bond");

if (it != namesToUCO.end()) {
    cout << it->second << endl; // vypíše 0
}

```

Algoritmy

Představení knihovny algoritmů

- máme různé druhy kontejnerů
- máme iterátory – ty představují jednotný způsob, jak zacházet s objekty uvnitř kontejnerů
- knihovna algoritmů využívá iterátorů k implementaci různých užitečných algoritmů
 - algoritmy typicky operují nad rozsahem (range), který je zadán dvojicí iterátorů (iterátor na první prvek rozsahu, iterátor za poslední prvek rozsahu)
- algoritmy je možno používat i s klasickými poli (ukazatele do polí fungují jako iterátory)
- používání algoritmů představuje funkcionální styl programování v C++

Řazení

- algoritmus sort

```

int arr[8] = {27, 8, 6, 4, 5, 2, 3, 0};

sort(arr, arr + 8);
// arr nyní obsahuje {0, 2, 3, 4, 5, 6, 8, 27}

// v C++11 můžeme používat pole i takto:
array<int,8> arr2 = {27, 8, 6, 4, 5, 2, 3, 0};
sort(arr2.begin(), arr2.end());

vector<int> vec;
vec.push_back(9);
vec.push_back(6);
vec.push_back(17);
vec.push_back(-3);

sort(vec.begin(), vec.end());
// vec nyní obsahuje -3, 6, 9, 17

```

- implicitně se při řazení používá porovnávání pomocí operátoru <
- toto chování můžeme změnit – `sort` může brát jako parametr porovnávací funkci (nebo funkční objekt, viz níže), kterou se nahradí volání operátoru <

```
bool pred(int x, int y) { return y < x; }
```

```
sort(vec.begin(), vec.end(), pred);
// vec nyní obsahuje 17, 9, 6, -3
```

- některé porovnávací funkční objekty už máme v knihovně algoritmů připravené

// předchozí příklad je možno přepsat takto:

```
sort(vec.begin(), vec.end(), greater<int>());
```

- k zamyšlení: proč bere funkce `sort` dva iterátory a ne přímo kontejner?
- Poznámka: `sort` nemusí být stabilní, proto standardní knihovna obsahuje ještě algoritmus `stable_sort`

Kopírování

- algoritmy `copy`, `copy_if` (C++11) a další
- `copy` bere tři parametry, první dva jsou iterátory, které specifikují zdrojový rozsah, třetí je iterátor, který ukazuje na místo, kam se má kopírovat
 - je třeba zajistit, aby v cílovém kontejneru bylo dost místa

```
set<int> s;
s.insert(15);
s.insert(6);
s.insert(-7);
s.insert(20);
```

```
vector<int> vec;
vec.resize(7);
// vec obsahuje {0, 0, 0, 0, 0, 0, 0}
```

```
copy(s.begin(), s.end(), vec.begin() + 2);
```

// vec nyní obsahuje {0, 0, -7, 6, 15, 20, 0}

- `copy_if` má navíc jako parametr predikát (funkce/funkční objekt vracející `bool`); kopírují se jen objekty splňující daný predikát

```
bool isOdd(int num) {
    return (num % 2) != 0;
}
```

```

}

array<int, 5> arr = {1, 2, 3, 4, 5};

copy_if(s.begin(), s.end(), arr.begin(), isOdd);

// arr nyní obsahuje {-7, 15, 3, 4, 5};

```

Kopírování s modifikací

- algoritmus **transform**
- dvě varianty
- první varianta bere jako parametry dva iterátory pro zdrojový rozsah, jeden iterátor ukazující na cíl a unární funkci, kterou postupně aplikuje na objekty ze zdrojového rozsahu a výsledky zapisuje do cíle
- funkcionální prvek C++: **transform** je vlastně obdoba funkce, známé v některých funkcionálních jazycích jako *map*

```

set<int> s;
s.insert(15);
s.insert(6);
s.insert(-7);
s.insert(20);

vector<double> vec;
vec.resize(7);
// vec obsahuje sedmkrát 0.0

double half(int num) {
    result num / 2.0;
}

transform(s.begin(), s.end(), vec.begin() + 1, half);

// vec nyní obsahuje 0.0, -3.5, 3.0, 7.5, 10.0

```

- druhá varianta bere dva zdroje pomocí tří iterátorů (začátek prvního rozsahu, konec prvního rozsahu, začátek druhého rozsahu), iterátor na cíl a binární funkci
 - předpokládá se, že oba zdrojové rozsahy jsou stejně dlouhé (proto zde není iterátor na konec druhého rozsahu)
 - obdoba *zipWith* z funkcionálních jazyků

```

vector<int> vec;
vec.resize(7);
vec[3] = 17;

```

```

array<char, 7> arr = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};

string combine(int num, char c) {
    return to_string(num) + ":" + c;
}

array<string, 7> strArr;

transform(vec.begin(), vec.end(), arr.begin(), strArr.begin(), combine);

// strArr nyní obsahuje "0:a", "0:b", "0:c", "17:d", "0:e", "0:f", "0:g"

```

Akumulace hodnoty

- algoritmus `accumulate`
- v základní verzi sečte všechny objekty v zadaném rozsahu pomocí operátoru `+`
- ve druhé verzi umožňuje dodat vlastní funkci (funkční objekt) místo `+`
- obě verze mají jako parametr kromě rozsahu taky počáteční hodnotu
- akumulace probíhá zleva
 - obdoba funkcionálního *foldl*

```

array<int, 6> arr = {1, 7, 8, 3, 2, 3};

int s = accumulate(arr.begin(), arr.end(), 100);

// s je 124

string producer(const string & s, int num) {
    return s.empty() ? to_string(num) : s + ", " + to_string(num);
}

string str = accumulate(arr.begin(), arr.end(), string(), producer);

// str nyní obsahuje řetězec "1, 7, 8, 3, 2, 3"

```

Funkční objekty

- některé algoritmy berou jako parametry funkce
- místo funkcí můžeme předávat obecnější funkční objekty
- funkční objekt je objekt třídy, která má definovaný operátor funkčního volání `()`
 - takové objekty je možno volat jako funkce
 - výhoda oproti funkcím: objekty mohou držet vnitřní stav
- v některé literatuře se funkční objekty nazývají funktory (functors)

– pozor! neplést s pojmem funktoru ve funkcionálním programování,
jde o jiný pojem

```
class LessThan {
    int threshold;
public:
    LessThan(int t) : threshold(t) {}
    bool operator()(int num) {
        return num < threshold;
    }
};

array<int, 7> arr = {5, 7, 8, 0, 2, -3, 99};

// count_if je algoritmus, který počítá, kolik prvků v zadaném rozsahu
// splňuje zadaný predikát

cout << count_if(arr.begin(), arr.end(), LessThan(7)) << endl;
// vypíše 4
cout << count_if(arr.begin(), arr.end(), LessThan(0)) << endl;
// vypíše 1
cout << count_if(arr.begin(), arr.end(), LessThan(100)) << endl;
// vypíše 7

class Tagger {
    int id;
public:
    Tagger() : id(0) {}
    string operator()(int num) {
        str = to_string(id) + ": " + to_string(num);
        ++id;
        return str;
    }
    int getId() const { return id; }
};

vector<string> vec;
vec.resize(7);

Tagger tagger;
transform(arr.begin(), arr.end(), vec.begin(), tagger);

// vec obsahuje řetězce "0: 5", "1: 7", "2: 8", "3: 0",
// "4: 2", "5: -3", "6: 99"
```

```
cout << tagger.getId() << endl;  
// vypíše 7
```

```
// funkční objekt můžeme volat jako funkci  
cout << tagger(-17) << endl;  
// vypíše 7: -17
```

```
// ekvivalentní zápis (nepoužívejte, jen pro ilustraci):  
cout << tagger.operator()(-17) << endl;
```

- řada předdefinovaných funkčních objektů v hlavičkovém souboru `<functional>`
 - obalují běžné operátory

```
vector<int> intVec;  
intVec.resize(7);  
transform(arr.begin(), arr.end(), vec.begin(), negate<int>());  
// co bude v intVec nyní?
```