

# PB161 Programování v jazyce C++

## Přednáška 6

Kontejnery  
Iterátory  
Algoritmy

Nikola Beneš

26. října 2015

## Několik částí

- řetězce
- I/O proudy
- funkce ze standardní knihovny jazyka C (<cstdlib> apod.)
- *dnes*: kontejnery, iterátory, algoritmy
- a mnohé další...

## Generické programování

- `syntax container<object_type>`

## Jmenný prostor `std`

## Kontejnery

- generické objekty, které mohou obsahovat jiné objekty

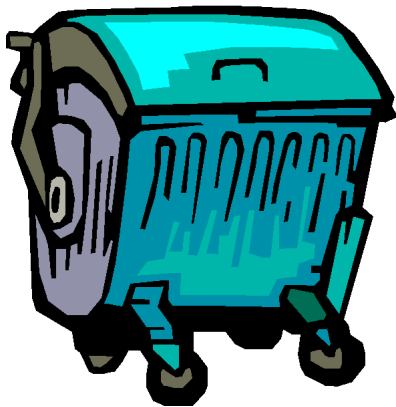
## Iterátory

- inteligentní ukazatele, pomocí kterých umíme kontejnery procházet

## Algoritmy

- generické operace nad kontejnery nebo jejich částmi
- využívají iterátory

## Kontejnery & iterátory



- složitější datové struktury
  - různé druhy operací
  - není jedna ideální datová struktura
- 
- uspořádání datové struktury je často nezávislé na konkrétním typu dat
  - základní operace jsou shodné
  - *proto*: generické kontejnery

# Typy kontejnerů

## Sekvenční

- `array` (C++11) – klasické pole
- `vector` – dynamické pole, může se zvětšovat
- `deque` – obousměrná fronta, rychlé přidávání/odebírání prvků
- `forward_list` (C++11), `list` – zřetězený seznam

## Asociativní

- `set` – uspořádaná množina
- `map` – asociativní pole (slovník), uspořádané dle klíče
- `multiset`, `multimap` – umožňují opakování klíčů
- `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap` – neuspořádané verze (hash tabulky), C++11

## Adaptéry

- `stack` (zásobník), `queue` (fronta), `priority_queue` (prioritní fronta)

## Dvojice a ntice

- `pair` – drží dva objekty různých (nebo stejných) typů
- `tuple` (C++11) – drží fixní počet objektů různých (nebo stejných) typů

## Řetězce

- `string` – fungují podobně jako kontejnery

## Šablonová třída pair

```
pair<int, char> p1;  
p1.first = 17;  
p1.second = 'a';  
pair<int, char> p2(p1);  
pair<int, float> p3(42, 3.14);  
  
p3 = p1;  
  
pair<string, int> psi;  
psi = make_pair("James Bond", 7);  
// je totéž jako  
psi = std::pair<const char *, int>("James Bond", 7);
```



## Šablonová třída `vector`

- pole, které se umí zvětšovat
- souvislý úsek paměti
- rychlost přístupu jako pole
- rychlé přidávání na konec
  - alokace větší paměti + překopírování
  - konstantní *amortizovaná* složitost
- podporuje i jiné operace
- vhodný pro mnohá použití

Syntaxe: vector<typ>

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

```
vec.push_back(17);
```

```
vec.push_back(21);
```

```
vec.push_back(0);
```

```
for (int i = 0; i < vec.size(); ++i) {  
    cout << vec[i] << endl;  
}
```

```
cout << vec[9] << endl; // nehlídaný přístup
```

```
cout << vec.at(9) << endl; // hlídaný přístup, vyhodí výjimku
```

```
vec.pop_back();
```

```
if (vec.empty()) { /* ... */ }
```

at() je pomalejší než operátor []

- nepoužívejte, když kontrola mezí nedává smysl!

```
for (int i = 0; i < vec.size(); ++i) {  
    cout << vec.at(i) << endl; // zbytečné!  
}
```

- rezervování kapacity, zvětšování vektoru

[ukázka]

- vektory lze kopírovat

- základní myšlenka: „inteligentní ukazatele“
- různé druhy podle kontejneru
  - sekvenční procházení
  - různé další operace
- různé metody kontejnerů používají iterátory
  - minimálně `begin()` a `end()`
  - `begin()` vrací iterátor na začátek kontejneru
  - `end()` vrací iterátor *za konec kontejneru*

# Iterátory pro vector

- 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(20);  
  
vector<int>::iterator vecIt = vec.begin();  
cout << *vecIt << endl;  
++vecIt; // posunutí na další prvek  
cout << *vecIt << endl; // vypíše 20  
--vecIt; // posunutí na předchozí prvek
```

# Iterátory pro vector (pokr.)

## Procházení vektoru

```
for (vector<int>::iterator it = vec.begin();  
    it != vec.end(); ++i) {  
    cout << *it << endl;  
    *it = 17;  
}  
  
// vec.end() ukazuje ZA poslední prvek vektoru
```

## Invalidace iterátorů

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

# Užitečné zkratky pro iterování v C++11

## Automatická dedukce typu

- typový specifikátor `auto`
- používat opatrně
- hodí se pro iterátory

```
for (auto it = vec.begin(); it != vec.end(); ++i) {  
    cout << *it << endl;  
}
```

## Nová syntax cyklů `for`

- range-based `for`

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

- funguje i pro klasická pole

## Užitečné zkratky pro iterování v C++11 (pokr.)

- pokud chceme prvky kontejneru měnit, musíme brát prvek referencí

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

- jak to funguje?

```
for (typ objekt : kontejner) { ... }
```

*// znamená něco jako: (ve skutečnosti trochu složitější)*

```
for (auto it = kontejner.begin();  
     it != kontejner.end(); ++it) {  
    typ objekt = *it;  
    ...  
}
```



## Šablonová třída set

- prvky se neopakují
- prvky se dají uspořádat
- typická implementace: stromy (často *red-black trees*)
- rychlé vkládání, mazání, vyhledávání
- iterátory vždy konstantní (neměnitelné)

```
set<int> s;  
s.insert(17);  
s.insert(25);  
s.insert(9);  
s.insert(-4);  
  
s.erase(17);  
s.erase(11);  
  
if (s.empty()) { /* ... */ }  
  
cout << s.size() << endl; // vypíše 3
```

## Vyhledávání

- iterátor na prvek, pokud existuje
- 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  
}
```

## Iterátory při vkládání/mazání

```
for (auto it = s.begin(); it != s.end(); ) {  
    if ( *it % 2 == 1 ) {  
        it = s.erase(it);  
    } else {  
        ++it;  
    }  
}
```

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

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

## Šablonová třída map

- asociativní pole, slovník
- mapování klíčů na hodnoty
- uspořádání podle klíčů
- klíče se nemohou opakovat
- funguje podobně jako set

```
map<int, string> cppTeachers;  
cppTeachers.insert(make_pair(72525, "Nikola Benes"));  
cppTeachers.insert(make_pair(374154, "Jiri Weiser"));  
cppTeachers.erase(374154);  
cppTeachers.erase(4085);
```

```
// C++11
```

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

## Vyhledávání

- iterátor na záznam, tj. dvojici (klíč, hodnota)
- 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  
}
```

## Operátor []

- uvnitř operátoru [] je klíč
- pokud záznam neexistuje, automaticky se vytvoří
- použijte opatrně (dávejte přednost spíš insert a find)

```
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



- různé užitečné algoritmy
- funkcionální styl programování v C++
- využívá iterátorů – jednotný způsob, jak zacházet s objekty uvnitř kontejnerů
- rozsah (range) – dvojice iterátorů
  - iterátor na první prvek rozsahu
  - iterátor za poslední prvek rozsahu
- algoritmy fungují i s klasickými poli
  - ukazatele fungují jako iterátory

## Algoritmus sort

```
int arr[8] = {27, 8, 6, 4, 5, 2, 3, 0};  
sort(arr, arr + 8);
```

```
// C++11  
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());
```

## Porovnávání

- implicitně: operátor <
- můžeme dodat vlastní funkci

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

```
sort(vec.begin(), vec.end(), pred);
```

- některé porovnávací funkční objekty už máme v knihovně algoritmů připravené

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

*Poznámka:* sort nemusí být stabilní, proto standardní knihovna obsahuje ještě algoritmus `stable_sort`

# Kopírování

## Algoritmus copy

- zdrojový rozsah, cílový iterátor
- 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}
```

# Kopírování (pokr.)

## Algoritmus `copy_if` (C++11)

- kopírují se jen objekty splňující daný predikát
  - funkce vracející `bool`

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

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

```
// s je množina { -7, 6, 15, 20 }
```

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

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

# Transformace

## Algoritmus transform

- kopírování s modifikací
- dvě varianty
- první varianta: unární funkce jako parametr
- funkcionální prvek C++: podobné *map* v Haskellu

```
// s je množina { -7, 6, 15, 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
```

# Transformace

- druhá varianta: dva zdroje, binární funkce
- podobné *zipWith* v Haskellu

```
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"
```

## Algoritmus accumulate

- sečte všechny objekty v zadaném rozsahu pomocí operátoru +
- umožňuje dodat vlastní funkci
- počáteční hodnotu
- akumulace probíhá zleva
- podobné *foldl* v Haskellu

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

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

```
// s je 124
```



## Akumulace (pokr.)

```
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

## Funkční objekt

- objekt třídy s operátorem ()
- možno volat jako funkce
- výhoda: mohou držet vnitřní stav

```
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};  
cout << count_if(arr.begin(), arr.end(), LessThan(7)) << endl;
```

## Funkční objekty (pokr.)

```
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);
```

# Funkční objekty (pokr.)

- předdefinované funkční objekty <functional>
- obalují běžné operátory

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

**Kontejnery**

**Iterátory**

**Algoritmy**

Používejte dokumentaci:

- <http://cppreference.com>
- <http://cplusplus.com/reference>