PB161 Programování v jazyce C++
Přednáška 3

Kontejnery
Iterátory
Algoritmy

Nikola Beneš

4. října 2016

Standardní knihovna C++

Už jsme viděli

- `std::string`
- `std::vector`
- jednoduchý vstup/výstup

Kontejnery

- objekty, které mohou obsahovat jiné objekty

Iterátory

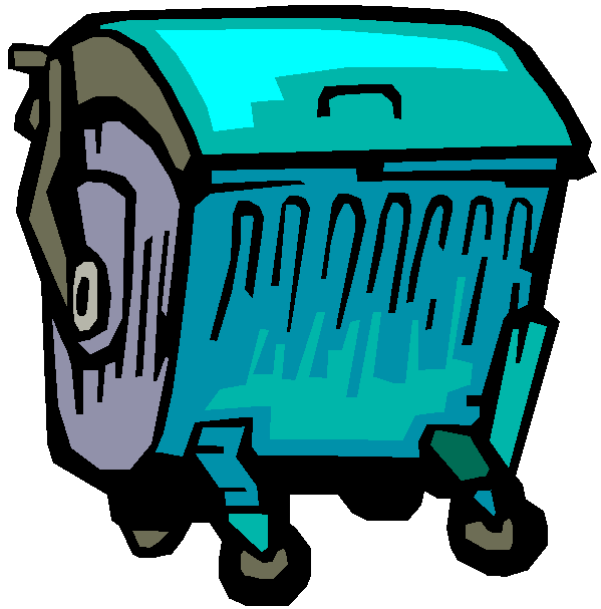
- inteligentní ukazatele dovnitř kontejnerů

Algoritmy

- operace nad kontejnery nebo jejich částmi

Princip **generického programování** – kontejnery smí obsahovat objekty libovolných typů; algoritmy jsou nezávislé na konkrétním typu.

Kontejnery & iterátory



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)

Související datové struktury

Dvojice a ntice

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

Řetězce

- `string` – fungují podobně jako kontejnery

Šablonové třídy `pair` **a** `tuple`

- deklarace

```
std::pair<int, char> p;
std::tuple<int, char, std::string> t;
std::pair<int, char> p1(42, 'x');
std::tuple<int, char, std::string> t1{13, 'z', "Kva"};
```

- přístup k prvkům

```
p.first = 17;
p.second = 'a';
std::get<1>(t) = 'x';
std::get<2>(t) = "www";
```

Dvojice a ntice (pokr.) <utility>

Užitečné pomocné funkce

- automatická dedukce typů

```
auto p = std::make_pair("James Bond", 007); // jaký je typ?
auto t = std::make_tuple(1, 1l, 1.0, 1.f); // jaký je typ?
```

- někdy není potřeba, stačí inicializace složenými závorkami

```
void f(std::tuple<int, double, std::string>);
f({3, 3.14, "Pi"});
```

- rozbalení ntice (ntice referencí)

```
std::tuple<int, char, double> getSomething();
int x;
char c;
double d;
std::tie(x, c, d) = getSomething();
```

Sekvenční kontejnery

`std::array<typ, počet>`

- pevný počet prvků; klasické pole ve stylu C s jinou syntaxí

`std::vector<typ>`

- dynamické pole; rychlé přidávání/odebírání prvků na jednom konci
- použitelné ve většině případů, kdy chceme uchovávat seznam objektů

`std::deque<typ>`

- rychlé přidávání/odebírání prvků z obou konců

`std::forward_list<typ>`, `std::list<typ>`

- zřetězené seznamy (jednosměrné, obousměrné)
- používejte jen pokud je skutečně potřebujete (typicky mnohem pomalejší než `vector`)

Sekvenční kontejnery (pokr.)

Typické operace

- inicializace, přiřazení, porovnávání (operátory ==, < apod.)
- indexování (operátor `[]`, metoda `at()`)
- `empty()`, `size()`
- `swap()` – prohození obsahu s jiným kontejnerem stejného typu

Vkládání

- `push_back()`, `push_front()`, `insert()` apod.
- varianty s `emplace`
  - místo vkládaného prvku berou parametry pro konstruktor
  - vytvoří nový prvek a vloží jej
  - může být efektivnější

Práce s iterátory

- `begin()`, `end()`

Iterátory

- základní myšlenka: „inteligentní ukazatele“
  - pro `vector` a `string` typická implementace: ukazatele
  - jiné kontejnery ale mohou mít složitější iterátory
- 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

Syntax:

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

```
std::vector<int> vec{10, 20, 30};

std::vector<int>::iterator vecIt = vec.begin();
std::cout << *vecIt << '\n'; // vypíše 10
++vecIt; // posunutí na další prvek
std::cout << *vecIt << '\n'; // vypíše 20
--vecIt; // posunutí na předchozí prvek
```

[ukázka insert]

Iterátory pro vector (pokr.)

Procházení vektoru (aneb život před C++11)

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

[ukázka: range-for]

Invalidace iterátorů

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

Uspořádaná množina <set>

Šablonová třída `set`

- prvky se neopakují
- prvky se dají uspořádat (operátorem <)
- typická implementace: stromy (často *red-black trees*)
- rychlé ($\mathcal{O}(\log n)$) vkládání, mazání, vyhledávání
- iterátory vždy konstantní (neměnitelné)

Operace

- `insert` vrací dvojici `std::pair<iterator, bool>`
  - iterátor na vložený (nebo už existující) prvek
  - `true`, pokud byl prvek vložen
- `erase` může brát hodnotu prvku ...
- ... nebo bere iterátor, příp. rozsah iterátorů
  - pak vrací iterátor na další prvek (od C++11)
- `find` vrací iterátor nebo `end()`
- a další (`empty`, `size`, ...)

[ukázka]

Asociativní kontejner <map>

Šablonová třída `map`

- asociativní pole, slovník
- mapování klíčů na hodnoty
- uspořádání podle klíčů
- klíče se nemohou opakovat
- funguje podobně jako `set`
- dvojice (klíč, hodnota): klíč se nesmí modifikovat, hodnota ano

[ukázka: `insert`, `erase`, `find`]

Asociativní kontejner (pokr.) <map>

Operátor []

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

```
std::map<std::string, int> namesToUCO;

namesToUCO["Nikola Benes"] = 72525;

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

auto iter = namesToUCO.find("James Bond");

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

Algoritmy

Knihovna algoritmů `<algorithm>`

- 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
  - ale možná je lepší preferovat `std::array` (C++11)

Řazení

Algoritmus `sort`

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

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

std::vector<int> vec = {9, 6, 17, -3};
std::sort(vec.begin(), vec.end());

std::vector<int> vec2 = {9, 6, 17, -3, 0, 1};
std::sort(vec.begin() + 2, vec.end() - 1); // co se stane?
```

Řazení (pokr.)

Porovnávání

- implicitně: operátor <
- můžeme dodat vlastní funkci

```
bool pred(int x, int y) { return y < x; }
std::sort(vec.begin(), vec.end(), pred);
```

- některé porovnávací funkční objekty už máme v knihovně algoritmů připravené

```
std::sort(vec.begin(), vec.end(), std::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

```
std::set<int> s = {15, 6, -7, 20};

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

std::copy(s.begin(), s.end(), vec.begin() + 2);
// vec nyní obsahuje {0, 0, -7, 6, 15, 20, 0}
```

- užitečné speciální iterátory `std::inserter` (pro `set`; volá `insert`), `std::back_inserter` (pro `vector`; volá `push_back`)

[ukázka]

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;
}

std::array<int, 5> arr = {1, 2, 3, 4, 5};
std::set<int> s = {-7, 6, 15, 20}

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

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

Další užitečné algoritmy

Algoritmus `transform`

- kopírování s modifikací
- dvě varianty
- první varianta: unární funkce jako parametr (*map*)
- druhá varianta: dva zdroje, binární funkce (*zipWith*)

Algoritmus `accumulate`

- sečte všechny objekty v zadaném rozsahu pomocí operátoru +
- umožňuje dodat vlastní funkci místo +
- počáteční hodnota
- akumulace probíhá zleva (*foldl*)

... a mnohé další

- `find`, `find_if`, ...
- `fill`, `generate`, `iota`, ...
- atd.

Funkční objekty

- algoritmy často berou jako (volitelný) parametr funkci
- ve skutečnosti to může být komplikovanější objekt, tzv. *funkční objekt*
  - objekt třídy s přetíženým operátorem ()
  - o tom později

Lambda funkce (od C++11)

```
std::vector<int> v = {10, 7, 5, 4, 2, 3};
int oddCount = std::count_if(v.begin(), v.end(),
    [](int x){ return x % 2 == 1; });
```

- trochu nad rámec tohoto předmětu, ale občas se hodí

Lambda funkce

Syntax

`[zachytávání](parametry){ tělo funkce }`

- zachytávání:
  - `[]` nic
  - `[x]` proměnnou `x` hodnotou
  - `[&x]` proměnnou `x` referencí
  - `[x,&y]` proměnnou `x` hodnotou a proměnnou `y` referencí
  - `[=]` všechny proměnné vyskytující se uvnitř těla funkce hodnotou
  - `[&]` všechny proměnné vyskytující se uvnitř těla funkce referencí

[ukázka]

Závěrečný kvíz

https://kahoot.it