2. Standardní knihovna – kontejnery, iterátory a algoritmy

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Kontejnery a iterátory

  Iterátory

  Sekvenční kontejnery

  Asociativní kontejnery

- Část 2 – Algoritmy

Standardní knihovna C++

Už jsme viděli na minulé přednášce

- `std::string`, `std::vector`
- jednoduchý vstup a výstup

Kontejnery

- objekty, které mohou obsahovat jiné objekty

Iterátory

- inteligentní ukazatele dovnitř kontejnerů

Algoritmy

- operace nad kontejnery nebo jejich částmi

Část I

Kontejnery a iterátory

STL kontejnery

- Kontejner (kolekce) je abstraktní datový typ určený na organizované skladování prvků konkrétního typu podle určitých pravidel.

  Ve standardní knihovně je připravena celá řada šablon užitečných kontejnerů.

- Kontejnery se od sebe významně liší způsobem přístupu k prvkům, možnostmi vkládání a rušení prvků a také časovou složitostí jednotlivých operací.
- Mezi kontejnery můžeme počítat i typ `std::string` pro zpracování řetězců znaků.
- Všechny STL kontejnery mají public kopírující konstruktor a operátor přiřazení (`=`).

  Více na některé z příštích přednášek.

- Základní dělení:
  - **Sekvenční kontejnery** – sekvenční (postupný) přístup k prvkům
  - **Asociativní kontejnery** – libovolný (náhodný) přístup k prvkům

Typy kontejnerů

Sekvenční

- `std::array` – klasické pole
- `std::vector` – dynamické pole, může měnit velikost
- `std::deque` – obousměrná fronta, rychlé přidávání/odebírání prvků na obou koncích
- `std::list` – zřetězený seznam

Asociativní

- `std::set` – uspořádaná množina
- `std::map` – asociativní pole (slovník), uspořádané dle klíče
- `std::multiset`, `std::multimap` – umožňují opakování klíčů

Adaptéry

- `std::stack` (zásobník), `std::queue` (fronta), ...

Související datové struktury

Dvojice a ntice

- `std::pair` – dva objekty různých (nebo stejných) typů
- `std::tuple` – fixní počet objektů různých (nebo stejných) typů

Řetězce

- `std::string` – fungují podobně jako kontejnery

Dvojice a ntice

<utility>

- 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" };
```

- Rozbalování

```
int a = p.first;
char b = p.second;
auto str = std::get<2>(t1);
```

- Přiřazování prvků

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

Řetězce <string>

- Inicializace

```cpp
std::string s1("Ahoj");
std::string s2(8, 'x');
std::string mesic = "Leden";
// implicitní volání konstruktoru
std::string mesic("Leden");
```

lec03/01-string-init.cpp

- Spíše nesprávná inicializace

```cpp
// neinicializovaný objekt
string foo = new string;
// anonymní objekt, který se zkopíruje a pak zničí
// jako funguje to, no...
string foo = string("Ahoj");
```

Řetězce – obecné vlastnosti

- Není nezbytně zakončen terminačním znakem
- Na rozdíl od **C** není identifikátor ukazatelem
- Lze využít operátor `[]` pro přístup k jednotlivým znakům řetězce
- Znaky jsou indexovány od `0` do `délka - 1`
- Délka je dostupná pomocí metody `length()`

```
1  std::string s1("ahoj");
2  std::cout << s1[1];
```

- Přiřazení pomocí operátoru `=`

```
1  s2 = s1;  // vytvoření separátní kopie
```

- Přístup pomocí `[]` je plnohodnotný, znaky řetězce lze i měnit

```
1  s1[1] = s2[0];
```

- Délka

```
s1.length();
```

- Přiřazení – http://www.cplusplus.com/reference/string/string/assign.html

```
s2.assign(s1);  // odpovídá s2 = s1
// zkopíruje N znaků od indexu start
s2.assign(s1, start, N);
```

- Přístup ke znakům – provádí kontrolu délky, může vyvolat výjimku out_of_range

```
s2.at(0) = s3.at(2);
```

lec03/02-string-elements.cpp

- Spojování

```
s3.append("pet");
s3 += "pet";
s3.append(s1, start, N);
```

Řetězce – členské metody

- Porovnávání

```
// přetížené operátory ==, !=, <, >, <=, >=
// vrací 0 v případě stejných řetězců, porovnává znak po znaku
s1.compare(s2)
// porovná části řetězců
s1.compare(start1, length1, s2, start2, length2);
// porovná část s1 s celým s2
s1.compare(start1, length1, s2)
```

- Části řetězců

```
// vrátí část řezezce od indexu start o délce N
s1.substr(start, N);
```

- Prohození

```
s1.swap(s2);
```

Řetězce – členské metody

- Charakteristiky

```
// počet znaků v řetězci
s1.size()
s1.length()
// počet znaků, které mohou být vloženy bez realokace
s1.capacity()
// maximální možná délka řetězce
// při překročení se vyvolá výjimka length_errorexception
s1.max_size()
// vrací true, pokud je řetezec prázdný
s1.empty()
// změní velikost na novou newlength
s1.resize(newlength)
```

Řetězce – členské metody

- Hledání – nalezení: vrácen index, nenalezení: `string::npos`

```
s1.find(s2) // vrací pozici s2 v s1
// vrací pozici s2 v s1 při hledání v od pos
s1.find(s2, pos)
// hledá zprava doleva
s1.rfind(s2)
s1.find_first_of(s2)
// vrací index prvního 'a', 'b' nebo 'c'
s1.find_first_of("abc")
s1.find_last_of(s2)
// první výskyt není v s2
s1.find_first_not_of(s2)
// poslední výskyt znaku který není v s2
s1.find_last_not_of(s2)
```

lec03/03-string-find.cpp

- Nahrazení

```
// begin:  index s1, kde začne funkce nahrazovat
// N: počet znaků s1, které budou změněny
// s2:  nahrazující řetězec
s1.replace(begin, N, s2)
// index:  prvek s2, kde začíná náhrada
// num:  počet prvků s2, které budou použity k nahrazení
s1.replace(begin, N, s2, index, num)
```

- Vkládání

```
// vloží s2 na index, nepřepisuje znaky s1
s1.insert(index, s2)
// vloží část řetězce s2 na pozici index1 v s1
s1.insert(index1, s2, index2, N);
```

- Konverze

```
// zkopíruje N znaků do pole char* ptr
// začíná na pozici index řetězce s1
// ukončuje řetězec NULL
s1.copy(ptr, N, index)

// vrací const char*
// ukončuje řetězec NULL
s1.c_str()

// vrací const char*
// NEukončuje řetězec NULL
s1.data()
```

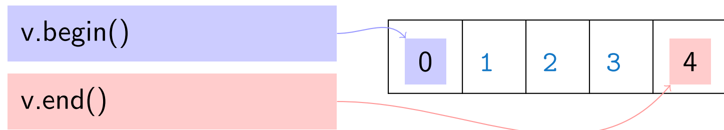
I. Kontejnery a iterátory

Iterátory

Sekvenční kontejnery

Asociativní kontejnery

Iterátory 1/2




```
auto iter = v.begin();
++iter; // posun na další prvek
```

- základní myšlenka: inteligentní ukazatele
  - pro `std::vector` a `std::string` typická implementace: (něco jako) ukazatele
  - jiné kontejnery ale mohou mít složitější iterátory
- různé druhy podle kontejneru
  - sekvenční procházení, různé další operace
- různé metody kontejnerů používají iterátory
- minimálně `begin()` (iterátor na začátek kontejneru) a `end()`, ale často i jiné

Iterátory 2/2

- Iterátor je datový typ definovaný v STL, můžeme si ale vytvořit i vlastní
- Odkazuje na nějaká data či pozici
- Iterátory jsou používány v algoritmech
- Data mohou být v nějaké kolekci (`vector`, `set`, ...), nebo např. v souboru,
- Iterátor např. umožní čtení celých čísel z I/O proudu (`std::cin`),
- Iterátor je typicky generická třída,
- Rozhraní iterátoru umožňuje čtení (dereference), posun vpřed (`++`) a porovnávání dvojice iterátorů (`==`, `!=`),
- Některé iterátory mají ještě další rozhraní

Příklad – součet čísel na standardním vstupu

```cpp
#include <iostream>
using namespace std;
int main ()
{
  int sum = 0, x;
  while ( cin >> x )
    sum += x;
  cout << sum << endl;
  return 0;
}
```

```cpp
#include <iostream>
#include <numeric>
#include <iterator>
using namespace std;
int main ()
{
  cout << accumulate (
    istream_iterator<int> (cin),
    istream_iterator<int> (), 0);
  cout << endl;
}
```

lec03/04-accumulate.cpp

- `istream_iterator<int> (cin)` – při dereferenci čte data typu integer ze std. vstupu
- `istream_iterator<int> ()` – znamená konec vstupu (EOF).

I. Kontejnery a iterátory

Iterátory

Sekvenční kontejnery

Asociativní kontejnery

Sekvenční kontejnery 1/2

- `std::array<typ, počet> (std::array<int, 42>)`
  - 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, který se má za běhu programu dynamicky měnit
- `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 2/2

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.

Vkládání

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

Pole fixní velikosti `std::array`

- Lze kontrolovat indexy, snadné kopírování.

```
1  #include <array>
2  #include <iterator>
3  // ..
4  array<int, 10> x;
5  x.fill (-1); // inicializace pole konstantou
6  x[3] = 100;  // nekontrolovaný přístup
8  for (array<int,10>::size_type i = 0; i < x.size(); i++)
9    x[i] += 100;
11 copy (x.begin(), x.end(), ostream_iterator<int> (cout, "\n"));
13 array<int, 10> y(x); // kopie pole
```

lec03/05-std-array.cpp

Pole proměnné velikosti `std::vector`

- Lze kontrolovat indexy, snadné kopírování, rozšiřování.

```
1   vector<int> x;
3   x.resize (10); // 0 0 0 ...  0
4   x.push_back (100);
5   x.insert (x.begin() + 5, 200); // vložení čísla
7   x[3] = 120; // změna prvku, bez kontroly
8   x.at (4) = 150; // změna prvku, kontrola
10  copy (x.begin(), x.end(), ostream_iterator<int> (cout, "\n"));
11  vector<int> y (x.begin()+1, x.begin()+9); // kopírování v rozsahu
12  sort (y.begin(), y.end ());
```

lec03/06-std-vector.cpp

Oboustranná fronta `std::deque`

- Dokáže nahradit zásobník i frontu
- Lze přidávat i odebírat z obou konců, přístup přes index.

```
deque<int> x;

x.push_back (100);
x.push_front (200);
x.insert (x.begin() + 1, 500);
// copy, read, display & pop
for (deque<int> y (x); !y.empty (); y.pop_front())
  cout << y.front () << endl;
x.erase (x.begin () + 1, x.begin () + 3 );
// iterate in reverse direction
deque<int>::reverse_iterator it;
for (it = x.rbegin(); it != x.rend (); ++it)
  cout << *it << endl;
```

lec03/07-std-deque.cpp

Obousměrný spojový seznam `std::list`

- Vkládání/odebírání prvku z libovolné z pozice (začátek, konec, iterátorem).

```
1  list<int> x;
3  x.push_back (100);
4  x.push_back (200);
5  x.push_front (300);
6  list<int>::iterator pos = x.begin ();
7  pos++;
8  x.insert (pos, 400);
9  pos++;
10 x.erase (pos);
11 // iterate forward
12 list<int>::iterator it;
13 for (it = x . begin(); it != x.end (); ++it)
14   cout << *it << endl;
```

lec03/08-std-list.cpp

I. Kontejnery a iterátory

Iterátory

Sekvenční kontejnery

Asociativní kontejnery

Množina prvků – std::set

- Uspořádaná množina, prvky se neopakují (prvek buď je nebo není obsažen).
- Vkládání/mazání/testování přítomnosti prvku.
- Iterátory jsou vždy konstatní

```
set<int> x;

x.insert (20);
x.insert (100);
x.insert (1000);

cout << (x.count (20) == 1 ? "present" : "not present" ) << endl;
set<int>::iterator pos = x.find (1000);
if (pos != x.end ()) x.erase (pos);
set<int>::iterator it;
for (it = x . begin(); it != x.end (); ++it)
  cout << *it << endl;
```

lec03/09-std-set.cpp

Asociativní pole – std::map

- Tabulka (klíč – hodnota), klíče se nemohou opakovat
- Vkládání/mazání/čtení prvku

```
1  map<string,int> x;
3  x.insert (make_pair ("test", 10));
4  x["key"] = 20;
5  x["testkey"] = x["test"] + x["key"];
7  map<string,int>::const_iterator it;
8  for (it = x . begin(); it != x . end (); ++it )
9    cout << it -> first << "->" << it -> second << endl;
11 map<string,int>::iterator pos = x.find ("test");
12 cout << (pos != x.end () ? "present" : "not present") << endl;
13 x.erase (pos);
```

lec03/10-std-map.cpp

Další kontejnery

- `forward_list`, C++ 11
  - jednosměrný spojový seznam,
  - obdoba `list` s nižší paměťovou režií, ale s omezením operací,

    Např. nelze snadno vkládat před pozici iterátoru.
  - sekvenční kontejner, ForwardIterator.
- `stack`
  - zásobník,
  - implementován jako wrapper na deque (příp. list nebo vector),
  - omezení rozhraní na LIFO,
  - sekvenční kontejner, nemá iterátor.
- `queue`
  - fronta, implementována jako wrapper na deque příp. list,
  - omezení rozhraní na FIFO,
  - sekvenční kontejner, nemá iterátor.

Další STL kontejnery

- `priority_queue`
  - fronta s prioritami,
  - prioritní ukládání je dosaženo tím, že se data ukládají do datové struktury halda,
  - sekvenční kontejner, nemá iterátor.
- `multiset`, `multimap`
  - třídy odpovídají `set` a `map`,
  - stejná hodnota klíče může být uložena vícekrát,
  - `#include <map>`, `#include <set>`
  - asociativní kontejnery, ForwardIterator.
- `bitset`
  - pole `bool` hodnot zadané velikosti,
  - kompaktní uložení (po jednotlivých bitech),
  - nemá iterátor.

Část II

Algoritmy

Knihovna algoritmů

`<algorithm>`

- různé užitečné algoritmy
- procedurální styl programování v C++
- využívá iterátory – 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` nebo `std::vector`

Základní algoritmy

`find(st, en, x)`

- hledá první výskyt prvku `x` v rozsahu `st` až `en`, vrací iterátor

```
1 vector<char> T = {'a', 'b', 'c', 'd', 'b', 'e'};
2 cout << find(T.begin(), T.end(), 'b') - T.begin();
```

`copy(st,en,dst)`

- kopíruje prvky z rozsahu `st` až `en` do cíle `dst` (a dále)
- volbou iterátoru `dst` lze použít i pro vstup / výstup dat, přesunutí či přidání

```
1 copy (T.begin(), T.end(), ostream_iterator<char>(cout, ", "));
```

`sort(st, en, fn)`

- seřadí prvky v rozsahu `st` až `en`, kritériem řazení může být `fn`

```
1 sort(T.begin(), T.end());
```

lec03/11-functions.cpp

Příklad – řazení

```
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);
```

Porovnávání

- implicitně: operátor <, můžeme dodat vlastní funkci (nebo lambdu)

```
std::sort(v.begin(), v.end(), [](int x, int y) {return y < x;});
```

Příklad – kopírování

- Algoritmus copy – zdrojový rozsah, cílový iterátor
- Je třeba zajistit, aby v cílovém kontejneru bylo dost místa

```
1  std::set<int> s = { 15, 6, -7, 20 };
3  std::vector<int> vec;
4  // vec bude obsahovat  0, 0, 0, 0, 0, 0, 0
5  vec.resize(7);
6  // vec bude obsahovat  0, 0, -7, 6, 15, 20, 0
7  std::copy(s.begin(), s.end(), vec.begin() + 2); 4
8
```

- Kopírování do kontejnerů je lépe řešit metodou insert pokud to jde:

```
1  std::set s = { 15, 6, -7, 20 };
2  std::vector vec = { 0, 1 };
3  // vkládá před danou pozici daný rozsah
4  vec.insert(vec.end(), s.begin(), s.end());
```

Další algoritmy

Další běžné úlohy a podpůrné STL funkce:

- binární vyhledávání: lower bound, upper bound, binary search,
- slučování (merge) a operace založené na slučování (průnik, sjednocení seřazených polí): `merge`, `set_union`, `set_intersection`,
- práce s datovou strukturou halda (heap): `make_heap`, `push_heap`, `pop_heap`,
- minimum a maximum: `min_element`, `max_element`,
- agregace: `accumulate`,
- test vlastnosti pro všechny prvky: `all_of`, `any_of`, `none_of`.