

# STL: standardní knihovna šablon

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

© Karel Richta, Martin Hořeňovský, Aleš Hrabalík, 2021

Programování v C++, B6B36PJC

11/2021, Lekce 10b

<https://cw.fel.cvut.cz/wiki/courses/b6b36pjc/start>



# STL je...

- Standard template library. Knihovna šablon, která je součástí standardní knihovny C++.
- Skládá se především z *kolekcí* a *algoritmů*.

*standard template library*  
standardní knihovna šablon

*containers*  
kolekce

*algorithms*  
algoritmy

*<iterator>*  
iterátory

*<functional>*  
funkční objekty

# Kolekce

- `std::vector` je příklad kolekce z STL.
- STL obsahuje celkem 13 druhů kolekcí v 9 hlavičkových souborech.

```
<vector>  
std::vector
```

```
<deque>  
std::deque
```

```
<array>  
std::array
```

C++11

```
<list>  
std::list
```

```
<map>  
std::map  
std::multimap
```

```
<set>  
std::set  
std::multiset
```

```
<forward_list>  
std::forward_list
```

C++11

```
<unordered_set>  
std::unordered_set  
std::unordered_multiset
```

C++11

```
<unordered_map>  
std::unordered_map  
std::unordered_multimap
```

C++11

# std::vector

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> container;
    container.push_back(1);
    container.push_back(2);
    container.push_back(3);

    std::cout << "c[1]: " << container[1] << "\n";
    std::cout << "size: " << container.size() << "\n";

    for (const auto& elem : container) {
        std::cout << "elem: " << elem << "\n";
    }
}
```

```
c[1]: 2
size: 3
elem: 1
elem: 2
elem: 3
```

# std::deque

```
#include <iostream>
#include <deque>

int main() {
    std::deque<int> container;
    container.push_back(2);
    container.push_back(3);
    container.push_front(1);

    std::cout << "c[1]: " << container[1] << "\n";
    std::cout << "size: " << container.size() << "\n";

    for (const auto& elem : container) {
        std::cout << "elem: " << elem << "\n";
    }
}
```

```
c[1]: 2
size: 3
elem: 1
elem: 2
elem: 3
```

# std::array

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 3> container;
    container[0] = 1;
    container[1] = 2;
    container[2] = 3;

    std::cout << "c[1]: " << container[1] << "\n";
    std::cout << "size: " << container.size() << "\n";

    for (const auto& elem : container) {
        std::cout << "elem: " << elem << "\n";
    }
}
```

```
c[1]: 2
size: 3
elem: 1
elem: 2
elem: 3
```

# std::list

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

```
int main() {
    std::list<int> container;
    container.push_back(2);
    container.push_back(3);
    container.push_front(1);

std::cout << "c[1]: " << container[1] << "\n";
    std::cout << "size: " << container.size() << "\n";

    for (const auto& elem : container) {
        std::cout << "elem: " << elem << "\n";
    }
}
```

```
c[1]: 2
size: 3
elem: 1
elem: 2
elem: 3
```

# std::forward\_list

C++11

```
#include <iostream>
#include <forward_list>

int main() {
    std::forward_list<int> container;
    container.push_front(3);
    container.push_front(2);
    container.push_front(1);

std::cout << "c[1]: " << container[1] << "\n";
std::cout << "size: " << container.size() << "\n";

    for (const auto& elem : container) {
        std::cout << "elem: " << elem << "\n";
    }
}
```

```
c[1]: 2
size: 3
elem: 1
elem: 2
elem: 3
```



# Sekvenční kolekce (*sequence containers*)

- `std::vector` podporuje změnu velikosti na konci, tj. metody `push_back` a `pop_back`, a operaci `[]`.
- `std::deque` navíc podporuje změnu velikosti na začátku, tj. metody `push_front` a `pop_front`.
- `std::array` má pevně určenou velikost, proto jsou zakázány operace `push` a `pop`, lze ale používat operaci `[]`. (C++11)
- `std::list` je obousměrně zřetězený spojový seznam, je zakázána operace `[]`.
- `std::forward_list` je jednosměrně zřetězený spojový seznam. Je zakázána operace `[]`, metoda `push_back`, dokonce i metoda `size`. (C++11)
- Pokud jsou k dispozici, asymptotická složitost operací `[]`, `push`, `pop` a `size` je  $O(1)$ .

# std::set

```
#include <iostream>
```

```
#include <set>
```

```
int main() {
```

```
    std::set<int> container;
```

```
    container.insert(3);
```

```
    container.insert(1);
```

```
    container.insert(2);
```

```
    container.insert(3);
```

```
    std::cout << "size: " << container.size() << "\n";
```

```
    std::cout << "cnt3: " << container.count(3) << "\n";
```

```
    for (const auto& elem : container) {
```

```
        std::cout << "elem: " << elem << "\n";
```

```
    }
```

```
}
```

```
size: 3  
cnt3: 1  
elem: 1  
elem: 2  
elem: 3
```

# std::multiset

```
#include <iostream>
#include <set>
```

```
int main() {
    std::multiset<int> container;
    container.insert(3);
    container.insert(1);
    container.insert(2);
    container.insert(3);
```

```
    std::cout << "size: " << container.size() << "\n";
    std::cout << "cnt3: " << container.count(3) << "\n";
```

```
    for (const auto& elem : container) {
        std::cout << "elem: " << elem << "\n";
    }
```

```
}
```

<b>size: 4</b>
<b>cnt3: 2</b>
<b>elem: 1</b>
<b>elem: 2</b>
<b>elem: 3</b>
<b>elem: 3</b>

# std::map

```
#include <iostream>
#include <map>
```

```
int main() {
    std::map<int, std::string> container;
    container.insert(std::make_pair(3, "abc"));
    container.insert(std::make_pair(1, "def"));
    container.insert(std::make_pair(2, "ghi"));
    container.insert(std::make_pair(3, "jkl"));
```

```
    std::cout << "c[3]: " << container[3] << "\n";
    std::cout << "size: " << container.size() << "\n";
    std::cout << "cnt3: " << container.count(3) << "\n";
```

```
    for (const auto& elem : container) {
        std::cout << "elem: " << elem.first;
        std::cout << " " << elem.second << "\n";
    }
```

```
}
```

```
// namespace std
template<typename U, typename V>
struct pair { U first; V second; };
```

```
c[3]: abc
size: 3
cnt3: 1
elem: 1 def
elem: 2 ghi
elem: 3 abc
```

# std::map

```
#include <iostream>
#include <map>
```

```
int main() {
    std::map<int, std::string> container;
    container[3] = "abc";
    container[1] = "def";
    container[2] = "ghi";
    container[3] = "jkl";
```

```
    std::cout << "c[3]: " << container[3] << "\n";
    std::cout << "size: " << container.size() << "\n";
    std::cout << "cnt3: " << container.count(3) << "\n";
```

```
    for (const auto& elem : container) {
        std::cout << "elem: " << elem.first;
        std::cout << " " << elem.second << "\n";
    }
```

```
}
```

```
// namespace std
template<typename U, typename V>
struct pair { U first; V second; };
```

```
c[3]: jkl
size: 3
cnt3: 1
elem: 1 def
elem: 2 ghi
elem: 3 jkl
```

# std::multimap

```
#include <iostream>
#include <map>
```

```
int main() {
    std::multimap<int, std::string> container;
    container.insert(std::make_pair(3, "abc"));
    container.insert(std::make_pair(1, "def"));
    container.insert(std::make_pair(2, "ghi"));
    container.insert(std::make_pair(3, "jkl"));
```

```
std::cout << "c[3]: " << container[3] << "\n";
std::cout << "size: " << container.size() << "\n";
std::cout << "cnt3: " << container.count(3) << "\n";
```

```
for (const auto& elem : container) {
    std::cout << "elem: " << elem.first;
    std::cout << " " << elem.second << "\n";
}
```

```
}
```

```
// namespace std
template<typename U, typename V>
struct pair { U first; V second; };
```

```
c[3]:
size: 4
cnt3: 2
elem: 1 def
elem: 2 ghi
elem: 3 abc
elem: 3 jkl
```

# Asociativní kolekce (*associative containers*)

- `std::set<T>` reprezentuje množinu s prvky typu `T`.
- `std::map<K, V>` ukládá hodnoty typu `V` s klíči typu `K`. Klíč s hodnotou je reprezentován objektem typu `std::pair<K, V>`.
- `std::multiset` umožňuje, aby se v kolekci opakovaly prvky.
- `std::multimap` umožňuje, aby se v kolekci opakovaly klíče.
- Metoda `count` určí, kolikrát se v kolekci vyskytuje daný prvek nebo klíč.
- Metoda `insert` vloží prvek nebo klíč s hodnotou do kolekce.
- Metoda `find` nalezne daný prvek nebo klíč.
- Prvky v `set`, `multiset` a klíče v `map`, `multimap` musí podporovat operaci `<`.
- Metody `count`, `insert` a `find` mají asymptotickou složitost  $O(\log n)$ , kde  $n$  je momentální velikost kolekce.

# Procházení kolekcí

- Pracujeme-li postupně s každým prvkem, říkáme, že kolekci *procházíme*, nebo že přes kolekci *iterujeme*.

```
elem: 1  
elem: 2  
elem: 3
```

```
for (auto& elem : container) { // C++11  
    std::cout << "elem: " << elem << "\n";  
}
```

```
for (auto i = begin(container); i != end(container); ++i) { // C++11  
    std::cout << "elem: " << *i << "\n";  
}
```

```
for (auto i = container.begin(); i != container.end(); ++i) { // C++11  
    std::cout << "elem: " << *i << "\n";  
}
```

```
for (std::vector<int>::iterator i = container.begin(); // C++03  
     i != container.end(); ++i) {  
    std::cout << "elem: " << *i << "\n";  
}
```



# Procházení kolekcí

- Pokud prvky potřebujeme jenom číst, ne měnit, můžeme použít `const`, `cbegin` a `cend`, `const_iterator`.

```
elem: 1
elem: 2
elem: 3
```

```
for (const auto& elem : container) { // C++11
    std::cout << "elem: " << elem << "\n";
}
```

```
for (auto i = cbegin(container); i != cend(container); ++i) { // C++14
    std::cout << "elem: " << *i << "\n";
}
```

```
for (auto i = container.cbegin(); i != container.cend(); ++i) { // C++11
    std::cout << "elem: " << *i << "\n";
}
```

```
for (std::vector<int>::const_iterator i = container.begin(); // C++03
     i != container.end(); ++i) {
    std::cout << "elem: " << *i << "\n";
}
```

# Procházení kolekcí

- Pokud prvky potřebujeme v opačném pořadí, použijeme `rbegin` a `rend`, případně `reverse_iterator`.

<code>elem: 3</code>
<code>elem: 2</code>
<code>elem: 1</code>

```
for (auto i = rbegin(container); i != rend(container); ++i) { // C++14
    std::cout << "elem: " << *i << "\n";
}
```

```
for (auto i = container.rbegin(); i != container.rend(); ++i) { // C++11
    std::cout << "elem: " << *i << "\n";
}
```

```
for (std::vector<int>::reverse_iterator i = container.rbegin(); // C++03
     i != container.rend(); ++i) {
    std::cout << "elem: " << *i << "\n";
}
```

# Algoritmy

- Algoritmy v STL pracují s daty prostřednictvím *iterátorů*.
- Jak získat iterátory?
  - Z kolekcí, jako návratovou hodnotu metod `begin`, `end`, `cbegin`, `cend`, `rbegin`, `rend`, `crbegin`, `crend`, `find`, `insert`, `lower_bound`, `upper_bound`, `equal_range`
  - Jinak, viz. `std::back_inserter`, `std::front_inserter`, `std::inserter`, `std::istream_iterator`, `std::ostream_iterator`
- Sekvence dat se reprezentuje pomocí dvou iterátorů, počátečního a konečného.
  - Počáteční iterátor ukazuje **na první** prvek.
  - Konečný iterátor ukazuje **za poslední** prvek.



# std::sort (std::stable\_sort)

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v ({
        6, 5, 3, 9, 7, 4, 0, 2, 8, 1
    });

    std::sort(begin(v), end(v));

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

# std::accumulate

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v ({
        6, 5, 3, 9, 7, 4, 0, 2, 8, 1
    });

    auto r = std::accumulate(begin(v), end(v), 0);

    std::cout << "soucet: " << r << "\n";
}
```

soucet: 45
------------

# std::rotate

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v ({
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    });

    std::rotate(begin(v), begin(v) + 5, end(v));

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

5	6	7	8	9	0	1	2	3	4
---	---	---	---	---	---	---	---	---	---

# std::unique

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v ({
        1, 5, 3, 3, 4, 0, 2, 1, 2, 4
    });

    std::sort(begin(v), end(v));           // 0 1 1 2 2 3 3 4 4 5
    auto u = std::unique(begin(v), end(v)); // 0 1 2 3 4 5 3 4 4 5
    v.erase(u, end(v));                   // 0 1 2 3 4 5

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

0	1	2	3	4	5
---	---	---	---	---	---

# std::shuffle

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>

int main() {
    std::vector<int> v ({
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    });

    std::random_device rd;
    std::mt19937 re(rd());
    std::shuffle(begin(v), end(v), re);

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

6 5 3 9 7 4 0 2 8 1
---------------------



# Úprava funkcionality `std::sort`

- Algoritmus `std::sort` určuje správné pořadí prvků pomocí operace `<`. Prvky kolekce jsou seřazeny v *neklesajícím* pořadí.
- Co když je ale chceme naopak v *nerostoucím* pořadí? Potřebujeme, aby `std::sort` používal místo operace `<` operaci `>`.
- Pomocí třetího, nepovinného parametru můžeme poskytnout vlastní operaci, kterou `std::sort` použije.
- Operaci můžeme předat čtyřmi způsoby:
  - Předáme ukazatel na funkci.
  - Předáme funkční objekt.
  - Předáme jiné iterátory.
  - Předáme lambda funkci.

- Jak seřadit prvky v nerostoucím pořadí?
- Možnost 1: Předáme *ukazatel na funkci comp*.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
bool comp(int a, int b) {
    return a > b;
}
```

```
int main() {
    std::vector<int> v ({
        6, 5, 3, 9, 7, 4, 0, 2, 8, 1
    });

    std::sort(begin(v), end(v), &comp);

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

9	8	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---	---

- Jak seřadit prvky v nerostoucím pořadí?
- Možnost 2a: Předáme *instanci funkčního objektu* Comp.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
struct Comp {
    bool operator()(int a, int b) { return a > b; }
};
```

```
int main() {
    std::vector<int> v ({
        6, 5, 3, 9, 7, 4, 0, 2, 8, 1
    });

    std::sort(begin(v), end(v), Comp());

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

9	8	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---	---

- Jak seřadit prvky v nerostoucím pořadí?
- Možnost 2b: Předáme *instanci funkčního objektu* `std::greater`.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
```

```
int main() {
    std::vector<int> v ({
        6, 5, 3, 9, 7, 4, 0, 2, 8, 1
    });

    std::sort(begin(v), end(v), std::greater<int>());

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

9	8	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---	---

- Jak seřadit prvky v nerostoucím pořadí?
- Možnost 3: Předáme jiné iterátory.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
int main() {
    std::vector<int> v ({
        6, 5, 3, 9, 7, 4, 0, 2, 8, 1
    });

    std::sort(rbegin(v), rend(v));

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

9	8	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---	---

- Jak seřadit prvky v nerostoucím pořadí?
- Možnost 4: Předáme *lambda funkci*.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
int main() {
    std::vector<int> v ({
        6, 5, 3, 9, 7, 4, 0, 2, 8, 1
    });

    std::sort(begin(v), end(v), [](int a, int b){ return a > b; });

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

9	8	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---	---

- Jiný příklad: Uspořádej kolekci tak, aby se v ní nacházely nejdříve liché, potom sudé prvky.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
int main() {
    std::vector<int> v ({
        6, 5, 3, 9, 7, 4, 0, 2, 8, 1
    });

    std::sort(begin(v), end(v), [](int a, int b){
        return (a & 1) > (b & 1);
    });

    for (const auto& elem : v) {
        std::cout << elem << " ";
    }
}
```

5 3 9 7 1 6 4 0 2 8
---------------------

# Lambda funkce

- Speciální syntaxe, která umožňuje vytvořit funkční objekt.
- Do lambda funkce vstupují data dvěma způsoby:
  - Standardními parametry. Uvádíme je klasicky v kulatých závorkách a s typem.
  - Proměnné, které jsou k dispozici v místě vytvoření lambda funkce, můžeme použít uvnitř těla. Mechanismus, který to umožňuje, se nazývá zachycení (capture). Zachycené proměnné uvádíme v hranatých závorkách a bez typu.

```
int dniVMesici = 31;
```

```
auto prictiMesic = [dniVMesici] (int cisloDne) {  
    return cisloDne + dniVMesici;  
};
```

```
dniVMesici = 40;
```

```
nejakeCisloDne = prictiMesic(nejakeCisloDne); // přičte 31
```



# Zachycení referencí

- Výchozí způsob zachycení je hodnotou; zachycená proměnná je zkopírována.
  - Takováto proměnná se také nesmí uvnitř lambdy měnit, pokud neurčíme, že chceme lambda s proměnným stavem (`mutable`).
- Pokud chceme změnit hodnotu zachycené proměnné, musíme ji *zachytit referencí*. Před její název přidáme symbol `&`.

```
int i = 0; int j = 0;
```

```
auto foo = [i, &j](int add) mutable {  
    i += add; // neovlivní vnější i, toto i je kopie  
    j += add; // ovlivní vnější j, toto j je reference  
};
```

```
i = 2; j = 2;
```

```
foo(3);
```

```
std::cout << i << ", " << j << "\n"; // 2, 5
```

# Úprava funkcionality `std::accumulate`

- Algoritmus `std::accumulate` provádí nad prvky součet (operaci `+`).
- Tuto operaci můžeme změnit čtvrtým, nepovinným parametrem. Nabízí se např. upravit funkcionalitu tak, aby se provedl součin, příp. zřetězení kolekce řetězců.
- Nesmíme ale zapomenout na třetí parametr, který slouží jako první mezivýsledek. Když mezivýsledek nemáme, musíme použít tzv. neutrální hodnotu pro danou operaci.
  - Neutrální hodnota pro součet je  $0$ .
  - Neutrální hodnota pro součin je  $1$ .
  - Neutrální hodnota pro zřetězení je `""`.

- Příklad: Zjisti součin prvků v kolekci.

```
#include <iostream>
#include <vector>
#include <algorithm>
```

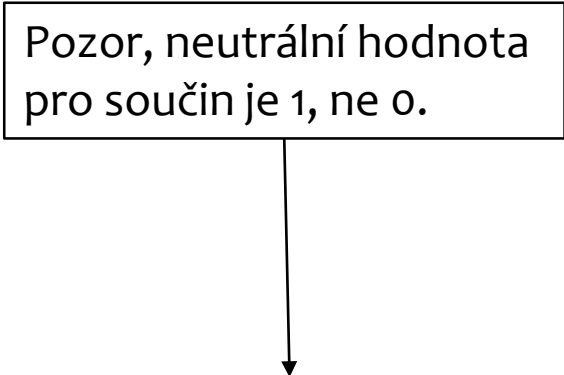
```
int main() {
    std::vector<int> v ({
        9, 8, 7, 6, 5, 4, 3, 2, 1
    });
```

```
    auto r = std::accumulate(begin(v), end(v), 1, [](int a, int b){
        return a * b;
    });
```

```
    std::cout << "soucin: " << r << "\n";
```

```
}
```

Pozor, neutrální hodnota  
pro součin je 1, ne 0.



soucin: 362880



# Další algoritmy

<code>std::count</code>	spočti počet výskytů
<code>std::mismatch</code>	najdi první prvek, kde se dvě posloupnosti liší
<code>std::equal</code>	zjistí, zda mají dvě posloupnosti shodné prvky
<code>std::find</code>	najdi prvek
<code>std::copy</code>	zkopíruj posloupnost
<code>std::fill</code>	vyplň posloupnost daným prvkem
<code>std::transform</code>	proved' operaci nad posloupností
<code>std::generate</code>	ulož výsledky opakovaného volání funkce do posloupnosti
<code>std::reverse</code>	obrat' posloupnost
<code>std::nth_element</code>	na n-tou pozici umístí prvek, který by tam byl po seřazení
<code>std::lower_bound*</code>	najdi první prvek, který je větší, nebo roven hodnotě
<code>std::upper_bound*</code>	najdi první prvek, který je větší, než hodnota
<code>std::equal_range*</code>	najdi podposloupnost, jejíž prvky jsou rovny hodnotě

více informací: <http://en.cppreference.com/w/cpp/algorithm> \*pole musí být seřazené

Děkuji za pozornost.

# Hashující kolekce (*unordered containers*)

C++11

- Kolekce `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map` a `std::unordered_multimap` používají *hashování*.
- Jejich metody `count`, `insert` a `find` mají průměrnou asymptotickou složitost  $O(1)$ .
- Prvky nebo klíče musí podporovat hashování, tj. musí existovat odpovídající specializace šablony `std::hash`. Specializace šablon je nad rámec této přednášky. Více informací a příklady: <http://en.cppreference.com/w/cpp/utility/hash>
- Vestavěné typy (`int`, `double`, apod.) a typy ze standardní knihovny (např. `std::string`) lze jako prvky nebo klíče používat bez problémů, odpovídající specializace `std::hash` jsou součástí standardní knihovny.

# Co je to iterátor?

- Aby byl nějaký objekt považován za iterátor, musí podporovat jisté operace. Iterátor je *koncept*.
- STL rozeznává více druhů iterátorů. Nejmocnější je *random access iterator*, iterátor s náhodným přístupem.

input iterator

forward iterator

bidirectional iterator

random access iterator

# Co je to iterátor? (podrobněji)

- Aby byl nějaký objekt považován za iterátor, musí podporovat jisté operace. Iterátor je *koncept*.
- STL rozeznává více druhů iterátorů. Používá pojmy *input iterator*, *output iterator*, *forward iterator*, *bidirectional iterator*, *random access iterator*. Pro představu (a velmi zhruba):
  - Objekt *i*, který umí  $++i$  a číst  $*i$ , je *input iterator*.
  - Objekt *i*, který umí  $++i$  a zapsat  $*i$ , je *output iterator*.
  - Input iterator *i*, který také umí  $i++$  a inkrementací nezpůsobí zneplatnění předešlých dat, je *forward iterator*.
  - Forward iterator *i*, který také umí  $--i$  a  $i--$  je *bidirectional iterator*.
  - Bidirectional iterator *i*, který také umí  $i += N$ , je *random access iterator*.
  - Input iterator, který je také output iterator, se nazývá *mutable*.



# Iterátory s náhodným přístupem

- Objekt `i` odpovídá konceptu *iterátor s náhodným přístupem*, pokud:
  - Podporuje operace `++i`, `--i`, `i++`, `i--`, `*i`, `i->`
  - Podporuje operace `i[N]`, `i+=N`, `i-=N`, `i+N`, `i-N`, `N+i`, kde `N` je celé číslo
  - Podporuje operace `==`, `!=`, `<`, `>`, `<=`, `>=`
  - Všechny tyto operace se chovají rozumně, viz.  
<http://en.cppreference.com/w/cpp/iterator>
- Všimněte si vztahu mezi iterátory a ukazateli:
  - Pokud lze nějaká operace provést s ukazatelem, tak lze také provést s iterátorem s náhodným přístupem a bude fungovat dle očekávání.
  - **Iterátor s náhodným přístupem** tedy můžeme považovat za **zobecněný ukazatel**. Můžeme mluvit o tom, že iterátor na něco ukazuje.
  - Navíc, **ukazatele jsou iterátory s náhodným přístupem**. Objekty typu `int*`, `char**`, atd., splňují všechny požadavky.