STL Kontajnery

Dátové štruktúry a algoritmy 25/26 LS
Prednášky, garant: prof. Gabriel Juhás
Cvičenia: Milan Mladoniczky - milan.mladoniczky@paneurouni.com

C++ Standard Template Library (STL)

- **Kontajnery (kolekcie)**
- Iterátory
- Algoritmy

- polia - Array
- vektory - Vector
- fronty - Queue, Deque
- zoznamy - List, Set, Map ...

C++ Standard Template Library (STL)

- Kontajnery (kolekcie)
- **Iterátory**
- Algoritmy

- Všetky kontajnery môžu byť prechádzanie prvok po prvku - iterované.
- Správanie je podobné pointeru.

```
vector<int>::iterator it;

vector<int> cisla = { 1, 2, 3, 4, 5 };
vector<int>::iterator zaciatok = cisla.begin( );
vector<int>::iterator koniec = cisla.end( );
```

C++ Standard Template Library (STL)

- Kontajnery (kolekcie)
- Iterátory
- **Algoritmy**

- zoraďovanie
- vyhľadávanie
- kopírovanie
- spočítanie
- a mnohé ďalšie algoritmy nad kontajnermi

Vector

#include <vector>

- Jednoduchý kontajner obsahujúci elementy rovnakého typu.
- Narozdiel od polí môže dynamicky meniť veľkosť.
- Zmena veľkosti a s tým spojené alokácie a dealokácie sú zabezpečené automaticky.

```
vector<int> num {1,2,3,4,5};

vector<int> zeros(5,0); // {0,0,0,0,0}

num.at(1) == 2;

num.at(2) = 33; // {1,2,33,4,5}

num.push_back(6); // {1,2,33,4,5,6}

num.pop_back(); // {1,2,33,4,5}


for(int i : num) {
     cout << i << ", ";
}
```

Iterátor

- Použitý na efektívne predchádzanie kontajnera
- Vytvorenie cez volanie metód kontajnera
  - **.begin()** - pozícia začiatku kontajnera
  - **.end()** - pozícia konca kontajnera
- Podobný pointeru
- Pozícia sa posúva pripočítaní/odpočítaním integeru

```
vector<int> num {1,2,3,4,5};

vector<int>::iterator iter;

for( iter = num.begin();
     iter != num.end();
     iter++) {

     cout << *iter << ", ";

}
```

List

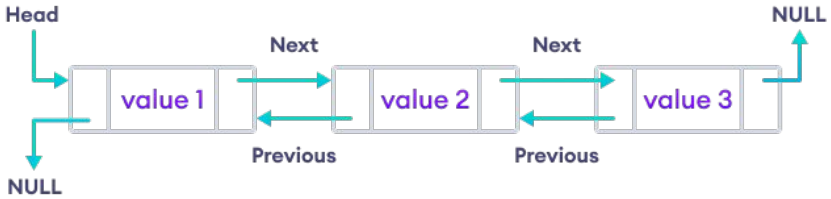
#include <forward_list>

- Sekvenčný kontajner s dynamickou veľkosťou.
- Efektívnejšie vkladanie prvkov na ľubovoľné miesto.
- Rôzne implementácie:
  - **Dopredne zreťazený**
  - Obojstranne zreťazený
  - Cyklický

List

#include <list>

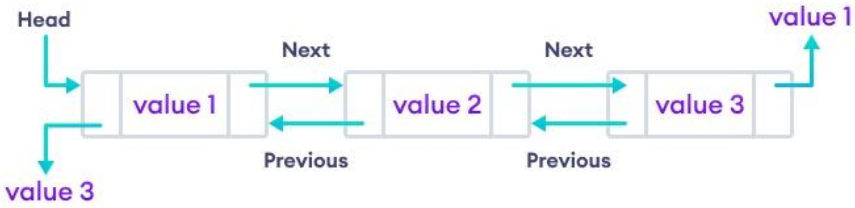
- Sekvenčný kontajner s dynamickou veľkosťou.
- Efektívnejšie vkladanie prvkov na ľubovoľné miesto.
- Rôzne implementácie:
  - Dopredne zreťazený
  - **Obojstranne zreťazený**
  - Cyklický



List

#include <list>

- Sekvenčný kontajner s dynamickou veľkosťou.
- Efektívnejšie vkladanie prvkov na ľubovoľné miesto.
- Rôzne implementácie:
  - Dopredne zreťazený
  - Obojstranne zreťazený
  - **Cyklický**



List

#include <list>

- Sekvenčný kontajner s dynamickou veľkosťou.
- Efektívnejšie vkladanie prvkov na ľubovoľné miesto.
- Rôzne implementácie:
  - Dopredne zreťazený
  - Obojstranne zreťazený
  - Cyklický

```
std::list<Type> list_name = {value1, value2,
...};

list<string> names = {"Milan"};

names.push_front("Martin"); // Martin,Milan
names.front(); // Martin
names.push_back("Janka"); // Martin,Milan,Janka
names.back(); // Janka

names.insert(names.begin()+2,"Fero");
// Martin,Milan,Fero,Janka


!! .at(index) neexistuje !!
!! potrebné použiť iterator !!
```

Set

#include <set>

- Dynamický zoradený kontajner.
- **Prvky** v kontajnery **sú unikátne**.
- Pre prechádzanie je možné použiť iterátor.
- Nie je možné vybrať si pozíciu nové prvku, je vložený podľa ostatných prvkov aby bola množina vždy zoradená.
- **Prístup** k prvkom **nie je cez index**.
- Existuje aj nezoradená varianta - *unordered_set*.

```
std::set<Type> set_name = {value1, value2, ...};

set<string> names = {"Milan"};

names.insert("Jano"); // Jano,Milan
names.insert("Fero"); // Fero,Jano,Milan

set<string>::iterator it = names.find("Milan");
// *it -> Milan

names.erase(it); // Fero,Jano
```

Map

#include <map>

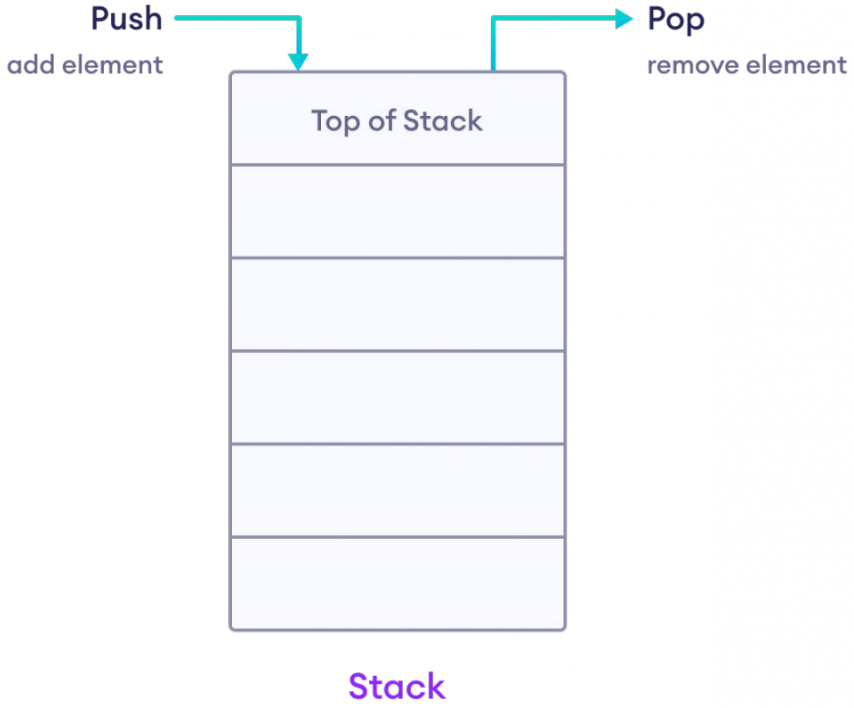
- Dynamický zoradený kontajner.
- Prvky kontajnera sú dvojica **kľúč-hodnota**.
- Kľúče sú vždy unikátne a zoradené.
- **Prístup** k prvkom je **cez kľúče**.
- Pre prechádzanie je možné použiť iterátor.
- Existuje aj nezoradená varianta - *unordered_map*.

```
std::map<Type,Type> map_name = {{key1, value1},
{key2, value2}};

map<int, string> students = {{1,"Milan"}};

students.insert({5,"Jano"}); // Milan,Jano
students.insert({2,"Fero"}); // Milan,Fero,Jano

students.at(5) // Jano
students[2] = "Martin" // Milan,Martin,Jano

students.erase(1); // Martin,Jano
```

Stack

#include <stack>

- Implementuje princíp **LIFO** (Last-in-First-out)
- Jednoduchý na použitie
- Vhodné použiť napríklad pri dávkovom spracovaní.
- **Nie je možné** pristúpiť k jednotlivým prvkom.
- **Nemá iterátor**



Stack

#include <stack>

- Implementuje princíp **LIFO** (Last-in-First-out)
- Jednoduchý na použitie
- Vhodné použiť napríklad pri dávkovom spracovaní.
- **Nie je možné** pristúpiť k jednotlivým prvkom.
- **Nemá iterátor**

```
std::stack<Type> stack;

stack<string> names;

names.push("Milan"); // Milan
names.push("Jano"); // Jano,Milan
names.push("Fero"); // Fero,Jano,Milan

names.pop(); // Fero


names.top(); // Jano
```

Fronta

#include <queue>

- Implementácia princípu **FIFO** (First-in-First-out)
- Jednoduché na použitie
- **Nie je možné** pristúpiť k jednotlivým prvkom.
- **Nemá iterátor**

Fronta

#include <queue>

- Implementácia princípu **FIFO** (First-in-First-out)
- Jednoduché na použitie
- **Nie je možné** pristúpiť k jednotlivým prvkom.
- **Nemá iterátor**

```
std::queue<Type> queue;

queue<string> names;

names.push("Milan"); // Milan
names.push("Jano"); // Milan,Jano
names.push("Fero"); //Milan,Jano,Fero

names.pop(); // Jano,Fero

names.front(); // Jano

names.back(); // Fero
```

https://dsa.interes.group/exercises/exercise-3