Лекция 2

СТАНДАРТНА БІБЛІОТЕКА ШАБЛОНІВ STL МОВИ ПРОГРАМУВАННЯ C++

Лекція 2. Стандартна бібліотека шаблонів STL мови C++

План

1. Загальні відомості про бібліотеку STL
2. Контейнери
3. Ітератори
4. Алгоритми
5. Функтори

1. Загальні відомості про бібліотеку STL

STL (Standard Template Library) – стандартна бібліотека шаблонів мови програмування C++, яка містить великий набір узагальнених класів, що описують різноманітні структури даних (контейнери або колекції), засоби доступу до їх елементів, а також стандартні алгоритми з обробки інформації.

На сьогодні бібліотека STL є невід'ємною частиною мови програмування C++ і підтримується її сучасними стандартами C++11, C++14 та C++17.

Бібліотека STL складається з наступних частин:

- **контейнерів** – класів, що зберігають певну сукупність (колекцію) елементів різної природи (векторів, списків, черг, стеків, хеш-таблиць тощо);
- **ітераторів** – класів, що реалізують уніфікований обхід всіх елементів контейнеру;
- **алгоритмів** – реалізації стандартних алгоритмів обробки даних, таких як пошук, сортування, визначення мінімальних та максимальних елементів колекції і т. п.;
- **функторів** – класів, що реалізують зберігання функціональних об'єктів.

2. Контейнери

Контейнером (**колекцією**) називається клас, призначений для зберігання певної множини об’єктів деякого типу.

В **STL** є велика кількість контейнерів, що реалізують різноманітні структури даних.

Найбільш популярними серед них є:

- **std::vector** – динамічний вектор з довільним доступом та автоматичною зміною розміру при додаванні або вилученні елементів:
- **std::list** – двонаправлений зв’язаний список;
- **std::stack** – контейнер, що реалізує функціональність стеку (LIFO);
- **std::queue** – контейнер, що реалізує функціональність черги (FIFO);
- **std::map** – колекція, в якій зберігаються відсортовані однозначні пари виду <ключ, значення>;
- **std::set** – впорядкована множина значень, що не повторюються.

2. Контейнери

Приклади застосування вектору std::vector

```
#include <vector> // Підключення відповідного заголовку

using namespace std;

vector<double> vec{1.5, 2.3, 4}; // Об’ява з ініціалізацією
vec.resize(10); // Зміна розміру
vec.push_back(3.14); // Додавання нового елементу в кінець вектору
vec.at(5) = -100; // Доступ по індексу
vec[6] = -1.4; // …
for (auto it : vec) // Ітерація по всім елементам vec (it – поточний елемент)
    cout << it << endl;
for (int i = 0; i < vec.size(); i++) // Ще один спосіб ітерації...
    cin >> vec[i];
```

2. Контейнери

Приклади роботи зі списком std::list

```
#include <list>

std::list<int> lst{1, 3, 2}; // Об’ява з ініціалізацією
lst.size(); // Кількість елементів
lst.push_back(5); // Додавання нового елементу в кінець списку
lst.pop_back(); // Вилучення елементу з кінця списку
for (auto it: lst) // Ітерація по всім елементам
      std::cout << it << std::endl;
for (auto it = lst.begin(); it != lst.end(); it++) // ...
      std::cout << *it << std::endl;
lst.sort(); // Сортування списку
```

2. Контейнери

Приклади застосування стеку std::stack

```
#include <stack>


using namespace std;

stack<int> stk; // Об’ява (ініціалізація не підтримується)
stk.size(); // Кількість елементів у стеці
stk.push(-10); // Додати елемент у стек
stk.pop(); // “Виштовхнути” елемент зі стеку
while (!stk.empty()) // Приклад обходу всього стеку
{
	cout << stk.top() << endl;
	stk.pop();
}
```

2. Контейнери

Приклади роботи з чергою std::queue

```
#include <queue>


using namespace std;

queue<float> qe; // Створення об’єкту
qe.size(); // Кількість елементів
qe.push(-1.0f); // Додавання нового елементу в кінець черги
qe.pop(); // Вилучення першого елементу з черги
qe.front(); // Перший елемент
qe.back(); // Останній елемент
while (!qe.empty()) // Обробка всіх елементів черги
{
	cout << qe.front() << endl;
	qe.pop();
}
```

2. Контейнери

Приклади використання словника std::map

```
#include <map>


using namespace std;

map<string, int> id{{"Li", 1}, {"Trump", 2}}; // Створення та ініціалізація
id.insert({"Buch", 3}); // Додавання нового значення
id["Buch"] = 4; // Зміна значення по ключу
id["Smith"] = 5; // Додавання (вилучення) значення по ключу
for (auto it : id) // Ітерація по контейнеру
	cout << it.first << ' ' << it.second << endl;
```

2. Контейнери

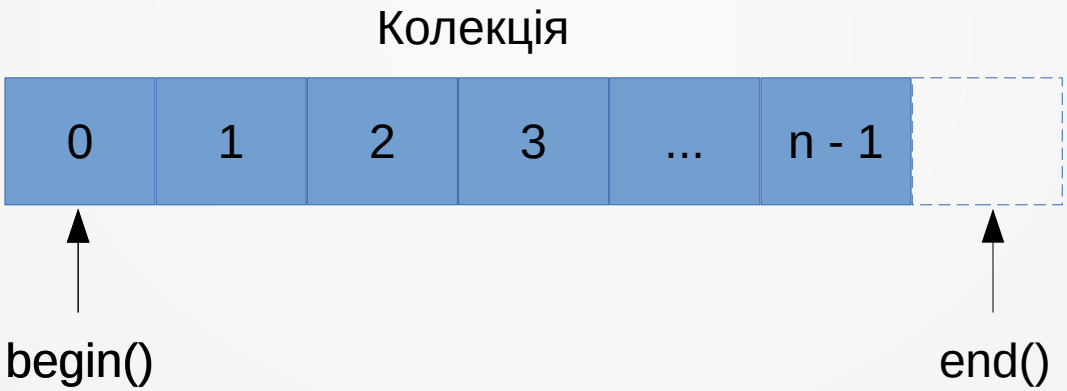
Приклади застосування множини std::set

```
#include <set>


using namespace std;

set<char> s{'a', 'b'}; // Створення та ініціалізація
s.size(); // Кількість елементів у множині
s.insert('H'); // Додавання нового елемента в множину
s.erase('a'); // Вилучення елемента
s.clear(); // Очищення контейнеру
for (auto it : s) // Ітерація по множині
	cout << it << endl;
```

3. Ітератори

Ітератор – це спеціальний клас-інтерфейс, який реалізує доступ до елементів колекції і навігацію по ним. Найпростішим типом ітератора є покажчик.

Ітератори в STL містять два обов'язкових метода: **begin()** і **end()**, які посилаються на перший та наступний за останнім елементи колекції (відповідно).



3. Ітератори

Існують наступні типи ітераторів:

- ітератор **вводу** (*it – доступ для читання, ++it, it++);
- **однонаправлений** ітератор (...);
- **двонаправлений** ітератор (..., --it, it--);
- ітератор **довільного доступу** (..., it += n, it -= n);
- **безперервний** ітератор (..., всі дані розташовані в пам’яті поруч).

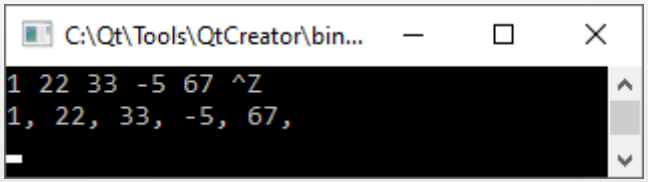
Ітератори вводу можуть застосовуватися тільки для того, щоб прочитати значення, на яке вказують. При цьому останнє значення, на яке вони вказують, стає недійсним під час даної операції. Тобто його не можна застосовувати для повторного ітерування по одному й тому же діапазону даних.

Прикладом такого ітератора є **std::istream_iterator**.

3. Ітератори

Приклад застосування ітератори вводу

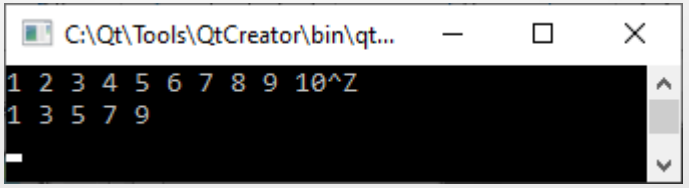
```
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

int main()
{
    istream_iterator<int> it_cin {cin}, end_cin;
    vector<int> v;

    copy(it_cin, end_cin, back_inserter(v));
    copy(begin(v), end(v), ostream_iterator<int>{cout, ", "});
    cout << '\n';
}
```



3. Ітератори

Приклад застосування ітератори довільного доступу

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

using namespace std;

int main()
{
    istream_iterator<int> it_cin {cin}, end_cin;
    vector<int> v;

    copy(it_cin, end_cin, back_inserter(v));
    for (auto it = begin(v); it != end(v); it += 2)
        cout << *it << ' ';
    cout << '\n';
}
```

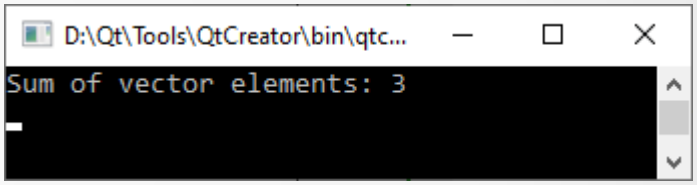


4. Алгоритми

Для роботи з контейнерами в STL реалізовано велику кількість різних алгоритмів, які описано у заголовному файлі algorithm.

for_each() – застосовує задану функцію до кожного елементу контейнера в заданому діапазоні.

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

int main()
{
    std::vector<int> vec{3, 4, 2, 9, -15};
    auto sum = 0;

    for_each(vec.begin(), vec.end(), [&](int &it){ sum += it; });
    std::cout << "Sum of vector elements: " << sum << ' ' << std::endl;
    return 0;
}
```



4. Алгоритми

У наведеному вище прикладі для кожного елемента вектора, що знаходиться в діапазоні від std::vector.begin() до std::vector.end(), виконується вираз, заданий лямбда-функцією:

```
[&](int &it){ sum += it; }
```

Лямбда-функція (**лямбда**, **лямбда-вираз**) це спосіб опису анонімних (безіменних) функціональних виразів, які можуть бути вбудовані в різні конструкції мови C++.

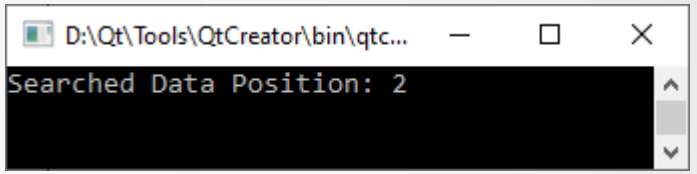
Наприклад, виведення всіх елементів вектора можна реалізувати наступним чином:

```
for_each(vec.begin(), vec.end(), [](int it){ std::cout << it << ' '; });
```

Відзначимо, що якщо лямбда-функція описана як [] ..., то в її тілі недоступні ніякі зовнішні змінні, якщо – [&] ..., то їй доступні за посиланням всі змінні із зовнішнього контексту, якщо – [=] ..., то їй доступні зовнішні змінні за значенням.

4. Алгоритми

search() – шукає перше входження заданої послідовності елементів в зазначений діапазон значень. Наприклад:

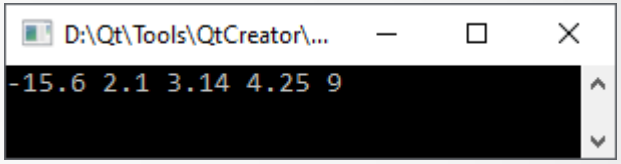



```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> data{3, 4, 2, 9, -15},
                     find{2, 9};
    auto ret = search(data.begin(), data.end(), find.begin(), find.end());

    if (ret != data.end())
        std::cout << "Searched Data Position: " << (ret - data.begin()) << std::endl;
    else
        std::cout <<  "Data not found" << std::endl;
    return 0;
}
```

4. Алгоритми

sort() – виконує сортування елементів в заданому діапазоні в порядку зростання. Наприклад:

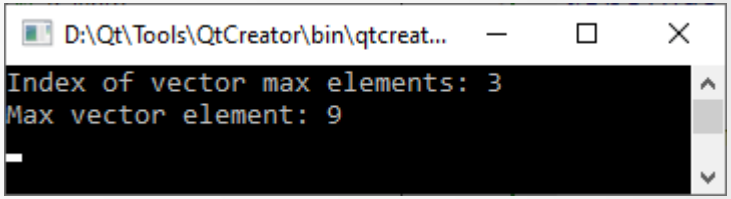
```
#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<double> vec{3.14, 4.25, 2.1, 9, -15.6};

    sort(vec.begin(), vec.end());
    for_each(vec.begin(), vec.end(), [](float it) { std::cout << it << ' '; });
    std::cout << std::endl;
    return 0;
}
```



4. Алгоритми

max_element() – знаходить найбільший елемент в заданому діапазоні. Наприклад:

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

using namespace std;

int main()
{
    vector<int> vec{3, 4, 2, 9, -15};
    auto index = max_element(vec.begin(), vec.end());

    cout << "Index of vector max elements: " << index - vec.begin() << ' ' << endl;
    cout << "Max vector element: " << vec.at(index - vec.begin()) << ' ' << endl;
    return 0;
}
```



4. Алгоритми

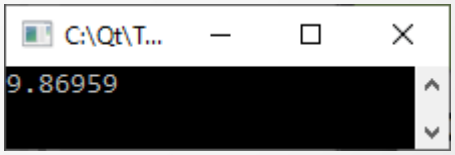
Крім того, в бібліотеці алгоритмів є ще такі часто вживані функції:

- **min_element()** – пошук найменшого елемента в заданому діапазоні;
- **minmax_element()** – пошук найменшого та найбільшого елементів в заданому діапазоні;
- **max()** – пошук найбільшого з двох аргументів;
- **min()** – пошук найменьшого з двох аргументів;
- **minmax()** – повертає більше та менше з двох елементів;
- **find()** – знаходить перший елемент, що задовольняє певним критеріям;
- **count()** – повертає кількість елементів, які відповідають певним критеріям;
- etc.

5. Функтори

Функтором (функціональним об’єктом) називається будь-який об’єкт, який можна викликати, як функцію. Наприклад, якщо в класі перезавантажено метод “()”, то об’єкт класу можна викликати як функцію.

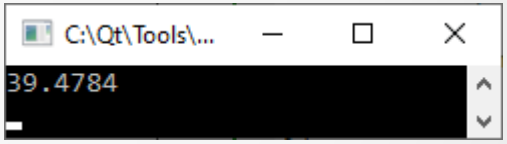
```
#include <iostream>

using namespace std;

struct Square
{
    double operator () (double value) const
    {
        return value * value;
    }
};

int main()
{
    Square square;
    cout << square(3.14159) << '\n';
    return 0;
}
```



5. Функтори

Починаючи зі стандарту C++11 в бібілотеці STL реалізовано клас **std::function** (описаний в заголовному файлі functional), який є високорівневою обгорткою для функторів і призначений для зберігання і виклику функціональних об’єктів. Наприклад:

```
#include <iostream>
#include <functional>

using namespace std;

// Опис Square
// ...

int main()
{
    Square square;
    function<double(double)> f = square;

    cout << f(3.14159 * 2) << '\n';
    return 0;
}
```



5. Функтори

```
#include <iostream>
#include <functional>

using namespace std;

// ...
double cube(double value)
{
    return value * value * value;
}

void print_value(const function<double(double)> & fun, double val)
{
    cout << fun(val) << '\n';
}

int main()
{
    Square square;
    function<double(double)> f1 = square, f2 = cube;

    print_value(f1, 3.14159);
    print_value(f2, 3.14159);
    return 0;
}
```