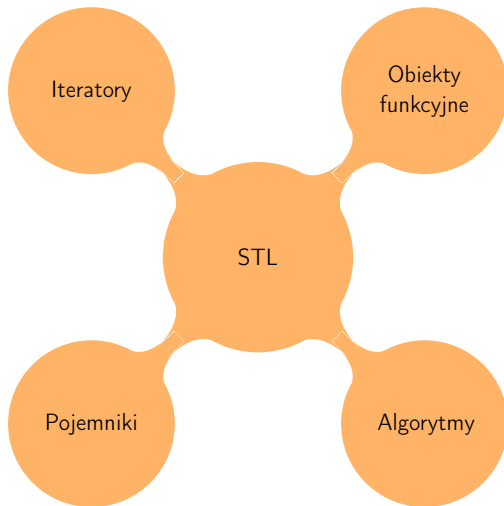


Standard Template Library - STL

Michał Marciniak

March 14, 2024



The C++ Standard Template Library (STL)

- ① <https://en.cppreference.com/>
- ② <https://cplusplus.com/reference/>
- ③ YT: CppCon; seria *Back to Basics*

- Przechowywanie danych (przy użyciu szablonów klas -> elastyczność)
- Metody (dużo metod jest wspólnych dla różnych pojemników)
- Wybór pojemnika zależy od funkcjonalności i złożoności operacji, ale:

Unless you have a solid reason not to, use a vector.

– Bjarne Stroustrup - *The C++ Programming Language*

- Pojemniki (Containers)
 - 1 sekwencyjne (vector, array, deque, list, forward_list)
 - 2 asocjacyjne (set, map, multiset, multimap)
 - 3 nieuporządkowane asocjacyjne (unordered_set, unordered_map, unordered_multiset, unordered_multimap)
 - 4 adaptory (queue, priority_queue, stack, flat_set, ...)

- Uogólnione wskaźniki, przystosowane do iteracji po elementach pojemnika
- Łącznik między pojemnikami, a algorytmami
- Sekwencje zdefiniowane na `[begin, end[`, tj.

```
*(container.begin()) = 5;    //ok  
*(container.end()) = 5;     //nie ok
```

- przykłady:

```
std::vector<int> vec{9,8,74,2,34,5,1};  
for(auto it = vec.begin(); it!=vec.end(); it++)  
    std::cout<<*it<<" ";  
for(auto it = vec.cbegin(); it!=vec.cend(); it++)  
    std::cout<<*it<<" ";
```

Rodzaje iteratorów

- *Input iterator* tylko do (wielokrotnego) odczytu; iteracja przez ++, dereferencja *, porównywanie == i !=
- *Output iterator* jednokrotny zapis przy użyciu *, iteracja przez ++
- *Forward iterator* (wielokrotny) zapis/odczyt za pomocą *; iteracja przez ++, wywołanie funkcji ->, porównywanie ==, != ; przykład forward_list
- *Bidirectional iterator* (wielokrotny) zapis/odczyt za pomocą *; iteracja ++ i --, wywołanie funkcji ->, porównywanie ==, != ; przykład list, map, set
- *Random-access iterator* (wielokrotny) zapis/odczyt za pomocą *, lub []; iteracja ++, +=, --, -=; wywołanie funkcji ->, porównywanie ==, !=, <, <=, >, >= ; przykład: deque
- *Contiguous iterator* (C++20) jak wyżej ale iterator może być czystym wskaźnikiem (raw pointer) przykład: vector, string, array

- zbiór ponad 100 algorytmów z header'a `<algorithm>`
- argumenty - iteratory, np:

```
std::vector<int> vec{9,8,74,2,34,5,1};  
std::sort(vec.begin(), vec.end());
```

```
std::ranges::sort(vec);           //od C++20
```

- nie wszystkie algorytmy działają na wszystkie pojemniki.
Przykład:

```
std::list<int> lis{ 1,2,43,4,5,68,9 };  
std::sort(lis.begin(), lis.end()); //nie działa  
auto pos = std::find(lis.begin(), lis.end(), 5);  
//ok, typ auto to std::list<int>::iterator
```

Obiekty funkcyjne (funktory)

- Wiele funkcji z **<algorithm>** może brać obiekty funkcyjne jako argumenty - poszerza to zakres stosowalności funkcji
- Funktor = obiekt posiadający zdefiniowany operator ()
- Wiele zdefiniowanych obiektów funkcyjnych w **<functional>**
np: greater/less, plus/minus, logical_and/logical_or

Obiekty funkcyjne (funktory)


- Wiele funkcji z **<algorithm>** może brać obiekty funkcyjne jako argumenty - poszerza to zakres stosowalności funkcji
- Funktor = obiekt posiadający zdefiniowany operator ()
- Wiele zdefiniowanych obiektów funkcyjnych w **<functional>**
np: greater/less, plus/minus, logical_and/logical_or
- Przykład : odwrotne sortowanie

```
std::vector<int> vec{95,8,74,2,16,34,5,1};
```

```
std::sort(vec.begin(), vec.end(), std::greater<>{});
```

std::vector, konstruktory

```
1  #include<iostream>
2  #include<vector>
3
4  template <typename T>
5  void print(const T& vec) {
6      for (auto& elem : vec) {
7          cout << elem << " ";
8      }
9      cout << endl;
10 }
11
12 int main() {
13
14     std::size_t n = 10;
15
16     std::vector<int> vec1 ;           //pusty vector
17     std::vector<int> vec2(n);         //n elementowy vector zer
18     std::vector<int> vec3(n, 5);      //n elementowy vector piątek
19     std::vector<int> vec4(5, n);      //pięcio-elementowy vector o wartościach n
20     std::vector<int> vec5{ 1,2,3,5,4,89,5 }; //vector o danych wartościach
21
22     std::vector<int> vec6 = vec2;      //vector taki sam jak vec2
23
24     print(vec1);
25     print(vec2);
26     print(vec3);
27     print(vec4);
28     print(vec5);
29     print(vec6);
30 }
31
32
```



```
Microsoft Visual Studio x + - □ x
print(vec1);
print(vec2); 0 0 0 0 0 0 0 0 0 0
print(vec3); 5 5 5 5 5 5 5 5 5 5
print(vec4); 10 10 10 10 10
print(vec5); 1 2 3 5 4 89 5
print(vec6); 0 0 0 0 0 0 0 0 0 0
```

std::vector, dostęp do elementów

```
int main() {  
  
    std::vector<int> vec{ 1,2,3,4,5,6,7,8,9,0 };  
    int element1 = vec[4]; //nie sprawdza poprawności adresu  
    int element2 = vec.at(4); //sprawdza poprawność adresu  
  
    int element3 = vec[41]; //ok, dostęp do adresu w pamięci  
    int element4 = vec.at(41); //program wyrzuci błąd  
  
    int element5 = *(&vec[0] + 3); //dereferencja odpowiedniego wskaźnika;  
    int element6 = *(vec.data() + 3);  
  
    int element7 = vec.front(); //dostęp do pierwszego elementu  
    int element8 = vec.back(); //dostęp do ostatniego elementu  
  
    return 0;  
}
```

std::vector, przejście po elementach

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

    for (size_t i = 0; i < vec.size(); i++) // ok
        std::cout << vec.at(i) << " ";
    std::cout << std::endl;
    //lepiej; tutaj auto = std::vector<int>::iterator
    for (auto i = vec.begin(); i != vec.end(); i++)
        std::cout << *i << " ";
    std::cout << std::endl;
    // przejście od końca;
    for (auto i = vec.rbegin(); i != vec.rend(); i++)
        std::cout << *i << " ";
    std::cout << std::endl;
    //najlepsza opcja; (jeśli nie interesują nas pozycje)
    for (auto& elem : vec) std::cout << elem << " ";
    std::cout << std::endl;
    return 0;
}
```

std::vector, metody 1

```
int main() {  
    std::vector<int> vec;           //tworzy pusty vector  
    std::cout << vec.size() << " " << vec.capacity() << std::endl;  
    //wypisz rozmiar i zarezerwowany rozmiar  
    for (int i = 0; i < 10; i++)vec.push_back(i); //dodawaj do vectora liczby od 0 do 9  
  
    print(vec);                    //wypisz vec  
  
    vec.pop_back();                //usuń ostatni element  
    vec.pop_back();                //usuń ostatni element  
  
    print(vec);                    //wypisz vec  
    std::cout << vec.size() << " " << vec.capacity() << std::endl; //do co poprzednio  
    vec.resize(20);                //zmień rozmiar  
    print(vec);                    //wypisz vec  
    return 0;  
}
```

std::vector, metody 2

```
int main() {  
    std::vector<int> vec{ 1,2,3,4,6,7,8,9,0 };  
    std::vector<int> vec1{ -1,-2,-3,-4 };  
  
    auto itr = vec.begin();    //iterator na pierwszy element  
  
    vec.insert(itr + 5, 5);    //wstaw value przed (pos - 1) element;  
    itr = vec.begin();        //ustaw iterator na pierwszy element  
    vec.insert(std::next(itr, 7), {1, 1});    //wstaw std::initializer_list  
    print(vec);  
  
    vec.insert(vec.begin(), vec1.begin(), vec1.end());    //wstaw vec1 na początek vec  
  
    print(vec)  
    // vec.erase działa na tak samo  
    return 0;  
}
```

- definiowanie wektora typu const:

```
const std::vector<int> vec{1,2,3,4,5};
```

- funkcje **begin()** i **end()** są też zdefiniowane jako funkcje działające na pojemniki:

```
std::vector<int>v{1,2,3}  
for(auto it=begin(v); it!=end(v);it++)
```

- ```
std::vector<bool> vec{1,0,1,1,1,1,0};
auto elem = vec[3];
```

kompilator wydedukuje typ zmiennej **elem** jako **std::vector<bool>::reference**. Operator **[]** zwraca referencję, a C++ zabrania referencji do poszczególnych bitów, dlatego używa się tzw. "proxy class"