

Biblioteka standardowa C++

- ▶ Wejście/wyjście (iostream)
- ▶ Napisy (string)
- ▶ Napisy jako strumienie (sstream)
- ▶ STL — Standard Template Library
 - ▶ Pojemniki (kolekcje)
 - ▶ Iteratory (wyliczanie elementów)
 - ▶ Algorytmy (sortowanie, etc.)
- ▶ Efektywne i wygodne klasy dla typowych zadań związanych z kolekcjami obiektów.

Pojemniki

- ▶ Pojemnik to obiekt, którego zadaniem jest przechowywanie innych obiektów.
- ▶ W STL pojemniki oparte są na wzorcach.
- ▶ Rodzaje pojemników (kolekcji)
 - ▶ Sekwencyjne (wektor, lista, kolejka)
 - ▶ Asocjacyjne (słowniki)
 - ▶ Nieuporządkowane (zbiór, multizbiór)

Caveat: STL jest dużą i skomplikowaną biblioteką, na wykładzie poruszymy tylko najprostsze zagadnienia.

Pojemniki

- ▶ Pojemniki z STL przechowują kopie obiektów.
- ▶ Obiekty przechowywane muszą posiadać odpowiedni konstruktor kopiujący.
- ▶ Zniszczenie pojemnika pociąga za sobą zniszczenie obiektów w nim przechowywanych. Jeśli przechowujemy wskaźniki do obiektów to obiekty wskazywane przez nie nie są niszczone.

Iteratory

- ▶ Iteratory to obiekty służące do poruszania się po pojemnikach.
- ▶ Uogólnienie wskaźników
- ▶ Iteratory umożliwiają
 - ▶ Ujednolicenie dostępu do pojemników
 - ▶ Równoczesny dostęp do jednego pojemnika przez wiele iteratorów
 - ▶ Różne rodzaje dostępu

Intuicja

```
T tab[10];  
T* begin = tab;  
T* end = tab+9;  
for(T* it=begin; it != end; it++)  
    f(*it);
```

Chcemy podobnie przechodzić po zawartości pojemnika, nawet jeśli nie jest on tablicą:

```
list<T> l;  
list<T>::iterator it;  
for(it=l.begin(); it != l.end(); it++)  
    f(*it);
```

vector

Pojemnik przeznaczony do przechowywania indeksowanych sekwencji (najlepiej o określonym z góry rozmiarze)

Dostarcza operacji

- ▶ indeksowania [] i at(),
- ▶ dodania i usuwania elementu na końcu wektora push_back() i pop_back() — wydajne o ile nie trzeba zwiększać pojemności,
- ▶ dodania i usuwanie elementu w środku insert() i erase() — mało wydajne

Rozmiary

- ▶ Wektor ma zawsze określoną pojemność; przy jej przekroczeniu zawartość jest automatycznie kopiowana do większego pojemnika.
- ▶ Informacji o pojemności dostarcza metoda `capacity()`
- ▶ Informacji o maksymalnej możliwej pojemności dostarcza metoda `max_size()` (dla G++/Linux $\sim 10^9$)
- ▶ Informacji o liczbie elementów dostarcza metoda `size()`
- ▶ Możemy z góry zarezerwować pewien rozmiar metodą `reserve(rozmiar)`

Przykład

```
#include <vector>
#include <iostream>
using namespace std;
int main()
{
    vector<int> v;
    for(int i=0;i<15;i++) {
        v.push_back(i);
        cout <<"Rozmiar " << v.size()
              <<" Pojemność: " << v.capacity() << endl;
    }
}
```


Rozmiar	1	Pojemność:	1
Rozmiar	2	Pojemność:	2
Rozmiar	3	Pojemność:	4
Rozmiar	4	Pojemność:	4
Rozmiar	5	Pojemność:	8
Rozmiar	6	Pojemność:	8
Rozmiar	7	Pojemność:	8
Rozmiar	8	Pojemność:	8
Rozmiar	9	Pojemność:	16
Rozmiar	10	Pojemność:	16
Rozmiar	11	Pojemność:	16
Rozmiar	12	Pojemność:	16
Rozmiar	13	Pojemność:	16
Rozmiar	14	Pojemność:	16
Rozmiar	15	Pojemność:	16

Zmiany rozmiaru

Przekroczenie zarezerwowanego rozmiaru pamięci powoduje

- ▶ Przydzielenie nowego większego rozmiaru pamięci
- ▶ Skopiowanie elementów ze starego przedziału do nowego
- ▶ Zniszczenie starych elementów
- ▶ Dealokację starego obszaru pamięci

Jeśli elementy są duże i mają kosztowne konstruktory i/lub destruktory może to być czasochłonne.

Przechodzenie wektora — indeksy

```
typedef vector<int> Liczby;
int main()
{
    Liczby v;
    for(int i=0;i<15;i++) v.push_back(i);
    for(int i=0;i<v.size();i++)
        cout << v[i] << endl;
}
```

Ale jeśli zamienimy wektor na inny pojemnik, to nie zadziała...

Przechodzenie wektora — iterator

```
typedef vector<int> Liczby;
int main()
{
    Liczby v;
    for(int i=0;i<15;i++) v.push_back(i);
    for(Liczby::iterator i=v.begin();i<v.end();i++)
        cout << *i << endl;
}
```

Ten kod zadziała nawet jeśli zamienimy wektor na inny pojemnik (byle definiował iterator).

Jeszcze o zaletach iteratorów

```
template<class Iterator> void
printIt(Iterator fst, Iterator last, string sep)
{
    for(Iterator it=fst; it != last; it++)
        cout << *it << sep ;
}

int main()
{
    vector<int> v;
    for(int i=0;i<15;i++) v.push_back(i);
    printIt(v.begin(),v.end()," ") ;
}
```

list

Pojemnik przeznaczony do przechowywania sekwencji z wstawianiem/usuwaniem w dowolnym miejscu.

Dostarcza operacji

- ▶ dodania i usuwania elementu na końcu `push_back()` i `pop_back()` — wydajne (tak jak dla wektora),
- ▶ dodania i usuwania elementu na początku `push_front()` i `pop_front()` — wydajne (tak jak dla wektora),
- ▶ dodania i usuwanie elementu w środku `insert()` i `erase()` — wydajne (inaczej niż dla wektora)

Uwaga: nie ma indeksowania!

Przykład

```
#include <list>
#include <iostream>
using namespace std;
int main() {
    list<int> v;
    for(int i=0; i<15; i++) {
        v.push_front(i);
    }
    while(v.size()) {
        cout << v.front() << ' '; v.pop_front();
    }
    // 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
}
```

W ten sposób mamy stos (LIFO) — może się przydać w programie zaliczeniowym.

Przykład

```
#include <list>
#include <iostream>
using namespace std;

int main()
{
    list<int> L;
    L.push_back(0);
    L.push_front(1);
    L.insert(++L.begin(), 2);
    printIt(L.begin(), L.end(), " ");
}
// 1 2 0
```


deque

Double Ended Queue (kolejka o dwóch końcach)

Pojemnik przeznaczony do przechowywania indeksowanych sekwencji z wstawianiem/usuwaniem na obu końcach.

Dostarcza operacji

- ▶ indeksowania [] i at(),
- ▶ dodania i susuwania elementu na końcu wektora push_back() i pop_back()
- ▶ dodania i usuwania elementu na początku push_front() i pop_front() —
- ▶ dodania i usuwanie elementu w środku insert() i erase()

Przykład

```
#include <deque>
#include <iostream>
using namespace std;
int main()
{
    deque<int> Q;
    Q.push_back(3);
    Q.push_front(1);
    Q.insert(Q.begin() + 1, 2);
    Q[2] = 0; // Tu było 3
    printIt(Q.begin(), Q.end(), " ");
}
// 1 2 0
```

Co możemy przechowywać?

Typ elementów T pojemnika powinien implementować co najmniej:

- ▶ Konstruktor kopiujący `T(const& T)`
- ▶ Operator przypisania
- ▶ Operator równości `==`
- ▶ Operator nierówności `!=`

takie, że

```
T a(b); assert(a == b);  
a = b; assert(a == b);  
a == a  
a == b wtw b == a  
a != b wtw !(a == b)
```

set

- ▶ Pojemnik przechowujący uporządkowane zbiory elementów.
- ▶ Typ elementu powinien definiować relację (operator)
 - <
 - ▶ antyzwrotną
 - ▶ antysymetryczną
 - ▶ przechodnią

Przykład

```
#include <set>
int main()
{
    set<string> indeks;
    string slowo;
    while(cin >> slowo){
        indeks.insert(slowo) ;
    }
    printIt(indeks.begin(), indeks.end(), " ");
}
// Wejście: Ala ma kota kota Ala ma
// Wyjście: Ala kota ma
```

map

- ▶ `map<key, value>` reprezentuje słownik o kluczach typu `key` i wartościach typu `value`.
- ▶ Typ kluczy powinien implementować operację `<` jak dla `set`
- ▶ Dostarcza operacji indeksowania, np.
 - `map<string, int> słownik;`
 - `słownik["answer"] = 42;`
- ▶ W rzeczywistości przechowywane są pary, o których można myśleć jako

```
template<class key, class val> struct pair {  
    key first;  
    val second;  
}
```

Przykład

```
map<string,int> indeks;  
string slowo;  
while(cin >> slowo) {  
    if(indeks.count(slowo))  
        indeks[slowo]++;  
    else  
        indeks[slowo]=1;  
}  
printIt(indeks.begin(),indeks.end()," ");  
// Wejście: Ala ma kota kota Ala ma kota kota m  
// Wyjście: Ala:2 kota:4 ma:3
```

Wypisywanie par

Tym razem, żeby funkcja `printIt` zadziałała, musimy zdefiniować operator `<<` dla par:

```
template<class key, class val>
ostream& operator<<(ostream& os, pair<key, val>
{
    os << p.first << ':' << p.second;
    return os;
}
```


multiset i multimap

- ▶ Warianty `set` i `map`, w których elementy (klucze) mogą się powtarzać
- ▶ Podobnie jak inne pojemniki asocjacyjne dostarczają metody `count(element)`, podającej ilość wystąpień elementu (klucza).
- ▶ Dla `set` i `map`, metoda `count` może dać w wyniku tylko 0 lub 1.

Napisy jako strumienie — `sstream`

- ▶ Nagłówek `sstream` deklaruje klasy pozwalające na czytanie/pisanie do/z napisów tak jak dla innych strumieni
- ▶ Dla pisania: klasa `ostringstream`
- ▶ Dla czytania: klasa `istringstream`

Przykład

```
#include <string>
#include <iostream>
#include <sstream>

using namespace std;
int main()
{
    string s("42");
    istringstream is(s);
    int answer;
    is >> answer;
    cout << answer << endl ;
    return 0;
}
```