

Standardowa biblioteka szablonów (STL)

Wykład 11

Pliki nagłówkowe języka C

- Wszystkie funkcje języka C zostały z przestrzeni globalnej przeniesione do przestrzenie nazw `std`
 - `//standard`

```
#include <cstdio>
using namespace std;
int main()
{ printf ("Hello World!"); return
0; }
```
 - `//niepraktykowane`

```
#include <stdio.h>
int main()
{ printf ("Hello World!"); return
0; }
```
- Stary sposób dołączania też jest możliwy, ale niepraktykowany

ANSI-C++	ANSI-C
<cassert>	<assert.h>
<cctype>	<ctype.h>
<cerrno>	<errno.h>
<cfloat>	<float.h>
<ciso646>	<iso646.h>
<climits>	<limits.h>
<locale>	<locale.h>
<cmath>	<math.h>
<setjmp>	<setjmp.h>
<csignal>	<signal.h>
<stdarg>	<stdarg.h>
<stddef>	<stddef.h>
<cstdio>	<stdio.h>
<stdlib>	<stdlib.h>
<cstring>	<string.h>
<ctime>	<time.h>
<wchar>	<wchar.h>
<wctype>	<wctype.h>

Standardowa biblioteka szablonów (STL)

- Rdzeniem standardowej biblioteki C++ jest tzw. standardowa biblioteka szablonów
- STL umożliwia zarządzanie kolekcjami danych przy użyciu wydajnych algorytmów, bez konieczności dogłębnego poznawania ich sposobu działania
- STL oferuje grupę klas kontenerowych zaspokajających rozmaite potrzeby wraz z algorytmami, które na nich operują
- STL wzbogaca język C++ o nowy poziom abstrakcji
 - Możemy zapomnieć o programowaniu dynamicznych tablic czy drzew oraz algorytmów do ich przeszukiwania
- Ze względu na swoją elastyczność STL wymaga objaśnienia
 - Trzeba się zapoznać z jego składnikami
 - Oraz nauczyć się wydajnego korzystania z dostarczonych algorytmów

Składniki STL

- **Kontenery** - służą do zarządzania kolekcjami obiektów określonego typu
 - Poszczególne kontenery mają różne zalety oraz wady i odzwierciedlają zróżnicowane potrzeby wobec kolekcji w tworzonych programach
- **Iteratory** - służą do poruszania się po kolekcjach
 - Oferują one interfejs wspólny dla każdego dowolnego typu kontenerowego
 - Interfejs iteratorów jest bardzo podobny operacji na wskaźnikach (możemy np. używać ++, *, ->)
- **Algorytmy** - służą do przetwarzania elementów kolekcji
 - Mogą one wyszukiwać, sortować, modyfikować lub po prostu wykorzystywać elementy
 - Algorytmy korzystają z iteratorów przez co mogą być używane do dowolnego typu kolekcji

Koncepcja STL

- Koncepcja biblioteki STL oparta jest na odseparowaniu danych od operacji
- Dane zarządzanie są przez klasy kontenerowe
- Operacje natomiast definiowane są przez konfigurowalne algorytmy
 - Operacje specyficzne dla danego kontenera są oczywiście implementowane w kontenerze
- Do łączenia danych i operacji używane są iteratory
- Koncepcja STL jest w pewnym sensie sprzeczna z ideą programowania zorientowanego obiektowo
 - Zamiast łączyć dane i algorytmy, rozdziela je
 - Wynika to z dużych możliwości takiego podejścia, ponieważ możliwe są dzięki temu różne kombinacje kontenerów i algorytmów z nimi współpracujących
- Biblioteka STL stanowi dobry przykład programowania uogólnionego (*generic programming*)

Rodzaje kontenerów

- **Kontenery sekwencyjne** - reprezentują kolekcje uporządkowane, w których każdy element posiada określoną pozycję
 - Pozycja zależy od momentu i miejsca wstawienia, ale nie zależy od samej wartości elementu
 - Należą do nich
 - **Vector** - wektor
 - **Deque** - kolejka dwustronna
 - **List** - lista
 - Łańcuchy - **string**, **basic_string<>**
 - Bardzo zbliżone do wektorów, ale ich elementami są znaki
- **Kontenery asocjacyjne** - będące kolekcjami sortowanymi
 - Położenie elementu zależy od jego wartości zgodnie z określonym kryterium sortowania
 - Należą do nich
 - **Set** - zbiór
 - **Multiset** - wielozbiór
 - **Map** - mapa
 - **Multimap** - multimapa
 - Nie ma w standardzie **hash**... (www.stlport.com, www.boost.org)

Wspólne cechy kontenerów

- Wszystkie kontenery zapewniają semantykę wartości
 - Przy wstawianiu wykonywana jest kopia obiektu
 - Elementy kontenera mogą być wskaźnikami do obiektów
- Elementy w kontenerach mają określoną kolejność
 - Możemy wykonywać wielokrotne iteracje w tej samej kolejności po wszystkich elementach
- Operacje na kontenerach nie zapewniają bezpieczeństwa
 - Funkcja wywołująca musi zapewnić spełnienie wymagań przez parametry operacji
 - Funkcje biblioteczne STL na ogół nie rzucają wyjątków

Wspólne operacje

Operacja	Skutek	Operacja	Skutek
<code>ConType c</code>	Pusty kontener	<code>c1.swap(c2)</code>	Zamiana
<code>ConType c1(c2)</code>	Inicjalizuje c2	<code>swap(c1, c2)</code>	Zamiana funkcja globalna
<code>ConType c1(beg, end)</code>	Inicjalizuje zakresem	<code>c.begin()</code>	Iterator do pierwszego elem.
<code>c ~ConType()</code>	Zwalnia pamięć	<code>c.end()</code>	Iter. do ost. elem.
<code>c.size()</code>	Liczba elem.	<code>c.rbegin()</code>	Iter odwrotny p.
<code>c.empty()</code>	Czy pusty	<code>c.rend()</code>	Iter odwrotny k.
<code>c.max_size()</code>	Maksymalna liczba elem.	<code>c.insert(pos, ele)</code>	Wstawia kopie elem.
<code>==, !=, <, ></code>	Operacje logiczne	<code>c.erase(beg, end)</code>	Usuwa elem. z zakresu
<code>c1 = c2</code>	Przypisanie	<code>c.clear()</code>	Opróżnia konten.

Operacje porównania

- Operatory `==`, `!=`, `<`, `>`, `<=`, `>=` zdefiniowane są według następujących reguł
 - Kontenery porównywane muszą być tego samego typu
 - Dwa kontenery są równe jeżeli ich elementy są równe i posiadają taką samą kolejność
 - W celu sprawdzenie czy jeden kontener jest mniejszy lub większy od innego przeprowadzane jest porównanie leksykograficzne

Porównanie leksykograficzne

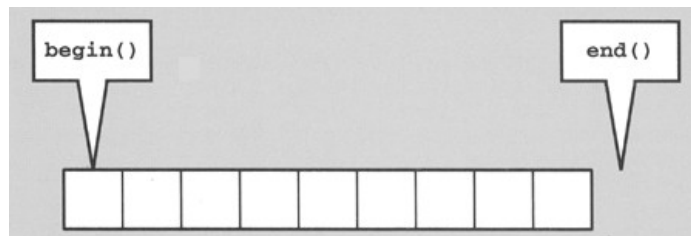
- Porównanie leksykograficzne polega na porównywaniu elementów po elemencie, aż wystąpienia takich przypadków
 - Jeżeli elementy nie są równe to wynik porównania jest wynikiem operacji
 - Jeżeli jeden ciąg nie posiada więcej elementów wówczas ciąg ten jest mniejszy od drugiego
 - W przypadku kiedy oba ciągi nie mają już elementów to są sobie równe

Iteratory

- Iteratory są obiektami, które potrafią nawigować po elementach kontenerów
- Podstawowe operacje definiowane dla iteratorów
 - `operator*` - zwraca element z aktualnej pozycji
 - `operator++` - przesuwa iterator na pozycję następną
 - `operator==` i `operator!=` zwracają wartość logiczną czy iteratory reprezentują tę samą (inną) pozycję
 - `operator=` - przypisanie
- Każdy kontener definiuje co najmniej dwa typy iteratorów
 - `kontener::iterator` - przeznaczony do nawigowania w trybie odczytu i zapisu
 - `kontener::const_iterator` - przeznaczony do nawigowania w trybie tylko do odczytu
 - Zrealizowane jest to za pomocą instrukcji `typedef`

Nawigowanie po kontenerach za pomocą iteratorów

- Wszystkie klasy kontenerowe zapewniają takie same podstawowe metody, które umożliwiają nawigowanie po ich elementach
 - `c.begin()` - zwraca iterator reprezentujący początek elementów w kontenerze, początkiem jest pozycja pierwszego elementu
 - `c.end()` - zwraca iterator reprezentujący koniec elementów w kontenerze, końcem jest pozycja za ostatnim elementem
 - Obie te funkcje definiują zakres półotwarty `[begin, end)`
 - Zaletą jest brak specjalnej obsługi zakresów pustych oraz proste kryterium zakończenia iteracji



Algorytmy

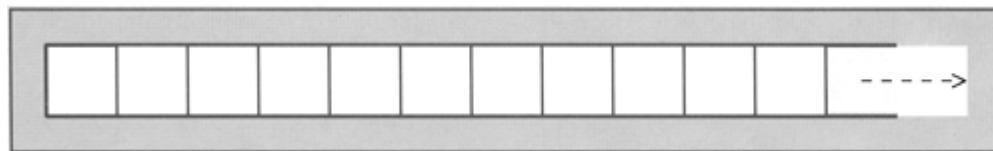
- Algorytmy służą do przetwarzania elementów kolekcji
 - Sortowanie, kopiowanie, przestawianie ...
 - Algorytmy są funkcjami globalnymi, a nie składowymi kontenerów
 - Pozwala to na jednokrotną implementację algorytmu dla wszystkich kontenerów, a nie dla każdego z osobna
 - Algorytmy pracują na zakresach (co najmniej jednym)
 - Algorytmy są oczywiście szablonami umieszczonymi w przestrzeni nazw `std`
- W celu używania algorytmów trzeba dołączyć plik nagłówkowy `<algorithm>`

Zakresy

- Zakres może obejmować cały kontener, ale nie musi
 - Dlatego w algorytmach podajmy początek i koniec zakresu
 - `sort(coll.begin(), coll.end());`
 - **Algorytmy nie sprawdzają poprawności zakresów!!!**
 - Funkcja wywołująca musi zapewnić poprawność zakresów, dla których wywołuje algorytm
 - Każdy algorytm przetwarza zakresy półotwarte
 - `[początek, koniec)`
 - **Nie jest brany pod uwagę ostatni element zakresu!!!**

Wektory

- Wektor zarządza swoimi elementami w tablicy dynamicznej
 - Pozwala na dostęp swobodny za pomocą indeksu
 - Dodawanie i usuwanie elementów na końcu jest szybkie
 - Wstawienie lub usunięcie z innego miejsca niż koniec nie jest już tak szybkie, bo wymaga reorganizacji pamięci
 - Elementy wektora zajmują ciągły obszar w pamięci
- W celu wykorzystywania wektorów należy dołączyć odpowiedni nagłówek
 - `#include <vector>`



■ Struktura wektora

Operacje

■ Operacje tworzenia

- ❑ `vector<T> c` - tworzy pusty wektor
- ❑ `vector<T> c1(c2)` - tworzy kopię wektora
- ❑ `vector<T> c(n)` - tworzy wektor o `n` elementach
- ❑ `vector<T> c(n, elem)` - tworzy wektor o `n` elementach o wartości `elem`
- ❑ `vector<T> c(beg, end)` - tworzy wektor inicjalizowany elementami z zakresu `[beg, end)`

■ Operacje niemodyfikujące

- ❑ Logiczne (np.. `==`, `!=`)
- ❑ `c.size()`, `c.empty()`, `c.max_size()` - wspólne z innymi kontenerami
- ❑ `c.capacity()` - zwraca pojemność wektora bez realokacji
- ❑ `c.reserve(n)` - rezerwuje pamięć dla `n` elementów

Operacje...

■ Operacje przypisania

- `c1 = c2`
- `c.assign(n, elem)` - przypisuje `n` kopii elementu `elem`
- `c.assign(beg, end)` - przypisuje elementy z zakresu `[beg, end)`
- `c1.swap(c2)` - zamienia `c1` z `c2`
- `swap(c1, c2)` - zamienia `c1` z `c2`

■ Operacje dostępu

- `c.at(idx)` - zwraca element o indeksie `idx`, jeśli `idx` poza zakresem to zgłasza wyjątek
- `c[idx]` - zwraca element o indeksie `idx` (brak kontroli zakresu), ale za to szybciej
- `c.front()` - zwraca pierwszy element
- `c.back()` - zwraca ostatni element

■ Przykład `cpp_11.1`

Funkcje iteratorowe

- Iteratory wektorów są iteratorami dostępu swobodnego, można więc korzystać ze wszystkich algorytmów STL
 - `c.begin()` - zwraca iterator dostępu swobodnego dla pierwszego elementu
 - `c.end()` - zwraca iterator dostępu swobodnego dla pozycji za ostatnim elementem
 - `c.rbegin()` - zwraca iterator odwrotny dla pierwszego elementu iteracji odwrotnej
 - `c.rend()` - zwraca iterator odwrotny dla pozycji za ostatnim elementem iteracji odwrotnej
- Operowanie na wektorach może również odbywać się tak jak na zwykłych tablicach
 - Wystarczy podać adres pierwszego elementu wektora
- Przykład `cpp_11.2`

Wstawianie i usuwanie elementów

- Wektory zapewniają szybkie operacje wstawiania i usuwania na końcu zarządzanej przez siebie tablicy
 - Operacje wstawiania lub usunięcia w innych miejscach są długie bo wymagają reorganizacji pamięci
- `c.insert(pos, elem)` - wstawia na pozycji `pos` element `elem`, zwraca pozycję nowego elementu
- `c.insert(pos, n, elem)` - wstawia na pozycji `pos` `n` elementów `elem`, nic nie zwraca
- `c.insert(pos, beg, end)` - wstawia na pozycji `pos` kopię elementów z zakresu `[beg, end)`, nic nie zwraca
- `c.push_back(elem)` - wstawia na koniec kopię elementu
- `c.pop_back()` - usuwa ostatni element (nie zwraca go)

Wstawianie i usuwanie elementów...

- `c.erase(pos)` - usuwa element na pozycji `pos`, zwraca pozycję następnego elementu
- `c.erase(beg, end)` - usuwa element z zakresu `[beg, end)`, zwraca pozycję następnego elementu
- `c.resize(n)` - zmienia rozmiar na `n`
- `c.resize(n, elem)` - zmienia rozmiar na `n` i ustawia elementy na wartość `elem`
- `c.clear()` - opróżnia wektor
- Przykład `cpp_11.3`

Rozmiar, a pojemność

- Dobrym sposobem zapewniania optymalnej wydajności jest wcześniejsze zarezerwowanie pamięci, tak żeby wszystkie elementy się zmieściły w niej
- W odniesieniu do rozmiaru wektory udostępniają funkcję `capacity()`, która zwraca liczbę możliwych do pomieszczenia elementów bez reorganizacji pamięci
 - Realokacja unieważnia wszystkie referencje, wskaźniki i iteratory dotyczące elementów wektora
 - Realokacja zajmuje czas
- W celu wcześniejszego zaalokowania pamięci możemy
 - Podać wielkość wektora w konstruktorze
`std::vector<T> Vec(100); //wywoływane konstruktory domyślne`
 - Zarezerwować odpowiednią ilość pamięci
`Vec.reserve(100); //zalecana wersja`
- Pojemność wektora nigdy nie maleje nawet jeśli usuwamy elementy

Rozmiar, a pojemność...

- Wektory zajmują ciągły obszar pamięci
- Pojemność wektorów nigdy nie maleje
 - Gwarantuje to, że nawet przy usuwaniu elementów z wektora referencje, wskaźniki i iteratory pozostają ważne pod warunkiem, że pokazują na istniejące elementy
- Istnieje pośredni sposób na zmniejszenie pojemności wektora
 - Utworzenie nowego wektora za pomocą konstruktora z inicjalizacją lub użycie metody `swap`
- Przykład `cpp_11.4`

Obsługa wyjątków

- Wektory zapewniają tylko minimalną kontrolę błędów
 - Jedyną funkcją od której standard wymaga obsługi błędów to funkcja `c.at()`
 - Innymi wyjątkami mogą być tylko wyjątki standardowe takie jak np. `std::bad_alloc`
- Założenia co do funkcji składowych jeżeli funkcje zdefiniowane przez użytkownika mogą zgłaszać wyjątki
 - `c.push_back()` - nie powoduje żadnego skutku lub OK
 - `c.insert()` - j.w.
 - `c.pop_back()` - zawsze OK
 - `c.erase()` i `c.clear()` - OK jeżeli operacje kopiowania nie zgłaszają wyjątków
- Jeśli używane elementy nigdy nie zgłaszają wyjątków podczas operacji kopiowanie (konstruktor kopiujący oraz operator `=`) to każda operacja kończy się powodzeniem lub nie powoduje żadnego skutku
 - Przy założeniu, że destruktory nigdy nie rzucają wyjątków

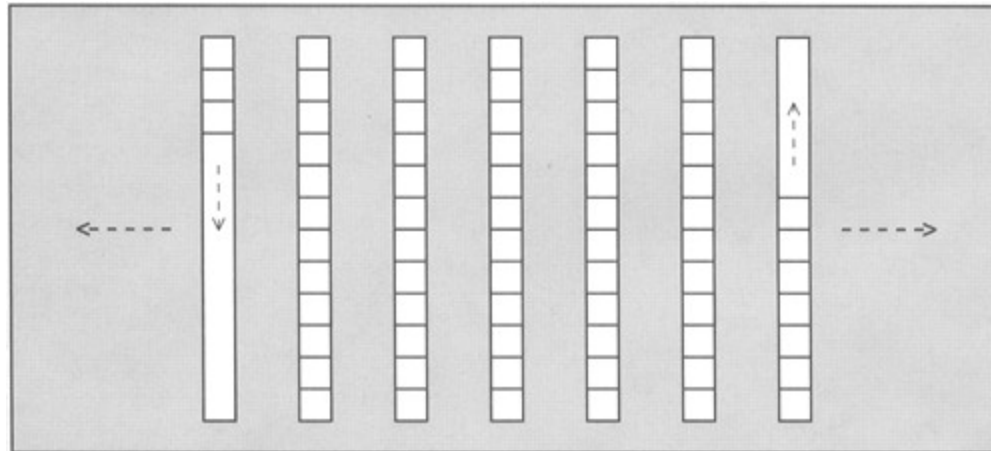
Kolejki dwustronne

- Kolejka dwustronna jest bardzo podobna do wektora
 - Inaczej Talia
- W celu wykorzystywania kolejek dwustronnych należy dołączyć odpowiedni nagłówek
 - `#include <deque>`
- Różnice w stosunku do wektorów
 - Dodawanie i usuwanie elementów na początku i na końcu jest szybkie
 - Opcje dostępu i ruchu iteratora są nieco wolniejsze
 - Kolejki nie zapewniają możliwości sterowania pojemnością ani momentem realokacji
 - Bloki pamięci mogą być zwalniane, czyli kolejka może zmaleć (zależne od implementacji)
- Podobieństwa do wektorów
 - Operacje wstawiania i usuwania w środku są wolne
 - Iteratory są iteratorami dostępu swobodnego

Wygląd kolejek



- Logiczna struktura kolejki



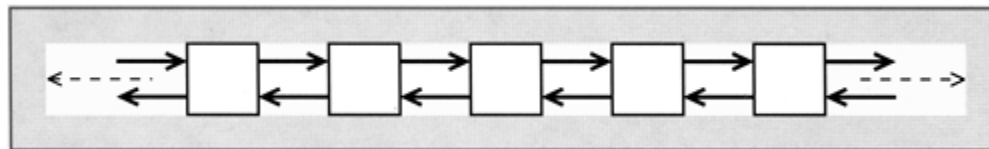
- Wewnętrzna struktura kolejki

Funkcje składowe

- Możliwości kolejek dwustronnych w stosunku do wektorów
 - Nie udostępniają metod
 - `c.capacity()`
 - `c.reserve()`
 - Udostępniają dodatkowo
 - `c.push_front(elem);`
 - `c.pop_front();`
 - Reszta metod jest taka sama ([slajd 16](#)), ([slajd 17](#)), ([slajd 19](#))
 - W obsłudze wyjątków
 - `c.pop_front()` zachowuje się tak jak `c.pop_back()`
 - `c.push_front()` zachowuje się tak jak `c.push_back()`
- Przykład `cpp_11.5`

Listy

- Elementy listy zorganizowane są w postaci listy dwukierunkowej
 - Możliwe jest poruszanie się do przodu lub do tyłu wzdłuż jej elementów
 - Iteratory są dwukierunkowe, a nie dostępu swobodnego
- W celu wykorzystywania list należy dołączyć odpowiedni nagłówek
 - `#include <list>`
- Wewnętrzna struktura list zasadniczo różni się od struktury wektorów i kolejek dwustronnych



■ Struktura listy

Możliwości list

- Lista nie zapewnia dostępu swobodnego
 - Aby dostać się do elementu np. 5 musimy przejść po kolei przez pierwsze cztery elementy
- Wstawianie i usuwanie na dowolnej pozycji jest szybkie
- Operacje wstawiania i usuwanie nie powodują unieważnienia wskaźników, referencji i iteratorów do innych elementów
- Działanie na listach prawie zawsze kończy się powodzeniem albo nie powoduje żadnej zmiany

Operacje

- Operacje tworzenia są takie same jak dla wektora i kolejki dwustronnej (slajd 16)
 - Np. `list<t> c(n);`
- Operacje niemodyfikujące są takie same jak dla kolejki dwustronnej (slajd 26)
 - Nie ma funkcji `reserve()` oraz `capacity()`
- Operacje przypisania są takie same jak dla wektora i kolejki dwustronnej (slajd 17)
 - np. `c.assign(n, elem);`
 - `swap(c1, c2);`

Operacje dostępu do elementów

- Bezpośredni dostęp do elementów
 - `c.front()` - zwraca pierwszy element listy (bez sprawdzanie poprawności)
 - `c.back()` - zwraca ostatni element listy (bez sprawdzanie poprawności)
- W celu dostępu do wszystkich elementów listy musimy użyć iteratorów
 - Są one dwukierunkowe (np. nie można wywołać dla listy algorytmu `sort`)
 - Funkcje iteratorowe są takie same jak dla wektora i kolejek dwustronnych

Operacje wstawiania i usuwanie

- Listy udostępniają takie same funkcje jak kolejki dwustronne (slajd 26)
- Dodatkowo w stosunku do kolejek, zapewniają metody (specjalne wersje algorytmów przystosowane do pracy z listami)
 - `c.remove(val)` ; - usuwa elementy o wartości `val`
 - `c.remove_if(op)` ; - usuwa elementy takie, dla których wywołanie `op(elem) == true`; (`op` - predykat)
- Przykład `cpp_11.6`

Funkcje splatające

- Listy udostępniają także wydajne metody służące do zmiany kolejności oraz powiązań między poszczególnymi elementami
 - W obrębie jednej listy
 - Pomiedzy dwoma listami tego samego typu
- Metody
 - `c1.splice(pos, c2)` - przenosi wszystkie elementy `c2` do `c1` i umieszcza je przed pozycją `pos`
 - `c1.splice(pos, c2, c2pos)` - przenosi element z pozycji `c2pos` z listy `c2` do `c1` i umieszcza go przed pozycją `pos` (`c1` i `c2` mogą być tą samą listą)
 - `c1.splice(pos, c2, c2beg, c2end)` - przenosi wszystkie elementy z zakresu `[c2beg, c2end)` z listy `c2` do `c1` i umieszcza je przed pozycją `pos` (`c1` i `c2` mogą być tą samą listą)
 - `c1.merge(c2)` - przy założeniu że `c1` i `c2` są posortowane przenosi elementy z `c2` do `c1` przy zachowaniu posortowania elementów
 - `c1.merge(c2, op)` - j.w. ale z użycie funkcji `op()`

Funkcje inne

■ Dalsze funkcje

- `c.sort()` - sortuje listę przy użyciu `<`
- `c.sort(op)` - sortuje listę używając do porównania `op()`
- `c.unique()` - usuwa powtórzenia kolejnych elementów o tej samej wartości
- `c.unique(op)` - j.w. ale gdy `op()` zwraca `true`
- `c.reverse()` - odwraca kolejność wszystkich elementów listy

■ Przykład `cpp_11.7`