Kontenery

#nawiasy_kątowe
#wektor #vector<>
#kolejka_o_dwóch_końcach #deque<>

W tym podrozdziale zajmiemy się dwoma najpopularniejszymi strukturami danych używanymi w języku C++: wektorem i kolejką o dwóch końcach. Obie te struktury określa się mianem *kontenerów* (ang. **container**, wymawiaj: *kontejner*), co oznacza ogólnie strukturę służącą do przechowywania danych.

Nas interesować będą standardowe kontenery, których szablony zostały zdefiniowane w bibliotece STL. Struktury te nieco różnią się swoją funkcjonalnością, ale mają dwie wspólne cechy: możemy w zasadzie dowolnie wybrać przechowywany w nich typ danych,*, a także są dynamiczne: możemy dodawać do nich dane lub usuwać je w trakcie biegu programu, tym samym zmieniając rozmiar kontenera. Tej drugiej cechy nie posiadają klasyczne tablice – można co prawda tworzyć tablice dynamiczne (w trakcie biegu programu), ale raz utworzonej tablicy nie da się powiększyć ani zmniejszyć.

Wektor i kolejka o dwóch końcach (oraz stos, kolejka FIFO i kolejka priorytetowa, które omówimy w następnym podrozdziale, a także `set` i mapa, na które przyjdzie czas później) stanowią swoisty kanon nawet dla niezbyt doświadczonych programistów.

Kontener `vector<>`

Jest to struktura przypominająca zwykłą tablicę o jednolitym (acz dowolnym) typie elementów. Oto przykładowa deklaracja wektora zawierającego liczby całkowite:

```
vector<int> V;
```

Jak widać, wybrany typ danych podajemy w nawiasach kątowych. Rozmiaru wektora nie musimy podawać (o tym zresztą za chwilę).

Powyższy wektor ma na razie rozmiar 0 (brak w nim elementów). Nowe elementy możemy najwygodniej (i najszybciej) dodawać na końcu wektora, na przykład:

```
V.push_back(11);
V.push_back(15);
V.push_back(20);
```

*Jest to podstawowa cecha *programowania generycznego*, zwanego też *programowaniem uogólnionym*.

Funkcja `size()` zwraca rozmiar, czyli ilość elementów wektora (tutaj: 3):

```
cout << V.size() << '\n';
```

Dostęp do elementów wektora jest taki sam, jak do elementów tablicy (są one numerowane od zera):

```
for(int i{ }; i < V.size(); i++)
  cout << V[i] << '\n';
```

Można także użyć do tego pętli **for-each** (*dla każdego*), gdzie zmienna x reprezentuje kolejne elementy wektora bez podawania ich numerów:

```
for(int x : V)
  cout << x << '\n';
```

W obu przykładach na ekranie pojawią się liczby:

```
11
15
20
```

Wartość elementów wektora można zmieniać, na przykład:

```
V[0]++;
V[1] = 10;
V[2] *= 2;
```

Teraz nasz wektor zawiera liczby: 12, 10, 40.

Zawartość wektora można posortować, posługując się wskaźnikami[†] `begin()` oraz `end()` (o sortowaniu już pisaliśmy):

```
sort(V.begin(), V.end());
```

Teraz wektor V zawiera elementy w kolejności niemalejącej: 10, 12, 40.

Elementy wektora można usuwać – najlepiej usuwa się element ostatni, choć w warunkach bojowych można usunąć element o dowolnym indeksie:

```
V.pop_back();
```

Ostatni element (o wartości 40) przestał istnieć – uczcijmy go minutą ciszy.

W tym momencie istnieją tylko dwa elementy: o indeksach 0 oraz 1. Jakiekolwiek odwołanie do elementu o innym indeksie spowoduje błąd, na przykład:

```
cout << V[10];
```

[†]Czy raczej *iteratorami*, którym poświęcimy oddzielny podrozdział.

zaowocuje mało eleganckim komunikatem (i przerwaniem działania programu):

```
Naruszenie ochrony pamięci
```

względnie w wersji angielskiej:

```
Segmentation fault
```

(wymawiaj: *segmentejszyn folt*). Dzieje się tak dlatego, że sięgamy do miejsca w pamięci, do którego sięgać nam nie wolno.[‡]

Przy deklaracji wektora można podać od razu jego spodziewany rozmiar, na przykład:

```
vector<string> U(5);
```

W tym momencie można korzystać z elementów o indeksach od 0 do 4, no i oczywiście można dodać nowe elementy (`push_back()`) albo usunąć istniejące (`pop_back()`).

Warto wiedzieć, że deklarowany w ten sposób wektor jest od razu inicjalizowany: jeśli jego elementy są na przykład typu `string`, wtedy mają one wartość pustego ciągu znaków. Jeśli są typu liczbowego, wtedy otrzymują wartość 0, a jeśli logicznego – wartość `false`. Wygodne to, prawda? Ale co zrobić, jeśli chcemy zainicjalizować elementy wektora inną wartością? Mieliśmy już taki przykład, gdzie tablicę liczb całkowitych inicjalizowaliśmy wartością -1 (w podrozdziale o rekurencji, przy okazji omawiania spamiętywania). Zawsze możemy posłużyć się wygodną funkcją `fill()`:

```
const int FMAX{45};
vector<int> Ftab(FMAX);
. . .
fill(Ftab.begin(), Ftab().end(), -1);
```

Pamiętamy, że użycie tej funkcji musi mieć miejsce we wnętrzu jakiejś funkcji – jakiejkolwiek, byle przed pierwszym wywołaniem funkcji obliczającej liczby Fibonacci'ego.

Można jednak zapisać to wszystko bardziej zwięźle. Otóż przy deklaracji wektora *Ftab* można podać drugi argument – właśnie tę wartość, jaką powinno się zainicjalizować jego elementy:

```
const int FMAX{45};
vector<int> Ftab(FMAX, -1);
```

Przy okazji zastanówmy się, jaka jest różnica w deklaracjach poniższych zmiennych:

```
vector<int> A(3), B[3], C{3};
```

Zmienna A to wektor zawierający 3 elementy (każdy o wartości 0), zaś B to tablica trzech pustych wektorów. Natomiast zmienna C to wektor o jednym elemencie o wartości 3.

[‡]Jedno z przykazań programisty brzmi: *Nie pchaj palca, gdzie Ci niemiło.*

Jeśli nie podaliśmy rozmiaru deklarowanego wektora, wtedy w domyśle pojawiał się wektor pusty (`size()` = 0). Jednak można od razu zarezerwować określoną wielkość wektora, podając tę liczbę w nawiasach okrągłych, na przykład to będzie wektor, który od razu będzie miał rozmiar 10 i każdy z jego elementów będzie miał wartość 0 (to było chwilę temu):

```
vector<int> W(10);
```

Oczywiście taki wektor można bez problemu zwiększać lub zmniejszać (funkcje: `push_back()`, `pop_back()`) – to też było. Wszelako istnieje metoda bardziej wydajna, powiększająca lub pomniejszająca wektor do dowolnego rozmiaru – jest to metoda[§] `resize()`. Na przykład instrukcja:

```
W.resize(20);
```

powiększa wektor W do rozmiaru 20 (nowe elementy otrzymują wartość 0), zaś instrukcja:

```
W.resize(3);
```

skraca jego długość do 3.

Mało tego: we wszystkich sytuacjach, gdy ustalamy rozmiar wektora (w deklaracji lub przy pomocy `resize()`), możemy użyć drugiego argumentu, którym będzie wartość nadawana wszystkim jego elementom. Na przykład poniższy 20-elementowy wektor zostanie wypełniony wartościami 5 (jak świadectwo bardzo pilnej uczennicy):

```
vector<int> Z(20, 5);
```

A w ten sposób powiększymy go do rozmiaru 25, dopisując same szóstki (przebrzydła kujonka!):

```
Z.resize(25, 6);
```

Kontener `deque<>`

Nieco podobnym do wektora jest kontener `deque<>` (ang. **double ended queue** czyli *kolejka o dwóch końcach*, wymawiaj: *dabl ended kiu*). Tak samo się go deklaruje, na przykład:

```
deque<int> D;
```

Tak samo działają na nim funkcje `size()`/`length()`, iteratory `begin()` oraz `end()` (a więc tak samo go się sortuje jak wektor). Tak samo przy deklaracji można podać jego spodziewany rozmiar i opcjonalnie: wartość, jaką trzeba nadać wszystkim jego elementom. Działa tak samo metoda `resize()`, która może zmienić jego rozmiar i ewentualnie nadać jego elementom jakąś (tą samą wartość). No to, po co nam ta żaba?

[§] W programowaniu obiektowym często zamiast słowa *funkcja* używa się słowa *metoda*. W gruncie rzeczy to jest to samo.

Otóż `deque<>` umożliwia wstawianie nowego elementu również na początku struktury i to w bardzo krótkim czasie. W wektorze da się to zrobić, ale trwa to znacznie dłużej. Na koniec wstawiamy element standardową funkcją `push_back()`, ale tutaj mamy także funkcję `push_front()`, która lokuje element na przedzie. Na przykład w wyniku poniższej sekwencji instrukcji:

```
deque<int> D;
D.push_back(5);
D.push_back(6);
D.push_front(9);
D.push_back(4);
for(int y : D)
  cout << y << ' ';
cout << '\n';
```

Na ekranie zobaczymy ciąg liczb:

```
9 5 6 4
```

Oczywiście do kompletu z funkcją `push_front()` mamy także funkcję `pop_front()`, która usuwa pierwszy element ze struktury.

Co lepsze?

Kiedy użyć `vector<>`, a kiedy `deque<>`? Różnica pomiędzy nimi tkwi w ich implementacji – w to nie chcielibyśmy zbyt głęboko wnikać. Wektor jest przydatny, gdy wykonujemy wiele operacji na długim ciągu sąsiednich elementów, bo – jak się okazuje – taka struktura jest zapisywana w bardzo szybkiej pamięci podręcznej procesora (ang. **cache**, wymawiaj: *kesz*). Natomiast `deque<>` nie gwarantuje aż takiej szybkości przy podobnych działaniach (ponieważ jego elementy są bardziej rozsiane w pamięci operacyjnej), ale przydaje się, gdy musimy korzystać z losowo wybranych jego elementów, na przykład w kolejce FIFO, dla której stanowi swego rodzaju bazę (omówimy to w innym podrozdziale).