Klasa vector

Język C++ wyposażony jest w bibliotekę **STL** (ang. Standard Template Library - Biblioteka Standardowych Szablonów). Zawiera ona wiele pożytecznych klas, które możemy wykorzystywać w swoich programach. Jedną z nich jest klasa **vector** definiująca tablicę dynamiczną. Jednakże w przeciwieństwie do zwykłych tablic dynamicznych klasa vector pozwala tablicę dynamicznie powiększać w czasie działania programu - nie musimy zatem rezerwować wyliczonej wcześniej liczby komórek.

Aby użyć klasy vector w programie , należy dołączyć plik nagłówkowy:

```
#include <vector>
```

Tablicę typu **vector** możemy w programie utworzyć na kilka różnych sposobów.
Oto trzy z nich:

1. vector<typ_elementów> nazwa_tablicy;
   Tworzona jest tablica bez elementów - zostaną one dodane później w trakcie wykonywania programu.
2. vector<typ_elementów> nazwa_tablicy(liczba_elementów, wartość_elementu);
   Tworzona jest tablica o zadanej liczbie elementów.
   Każdy element otrzymuje zadaną wartość.
3. vector<typ_elementów> nazwa_tablicy(inna_tablica);
   Tworzona jest tablica będąca kopią innej tablicy.

Dostęp do elementów tablicy odbywa się na dwa sposoby za pomocą indeksu:

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

using namespace std;

int main()
{
  vector<int> T(10,5);  // tworzymy tablicę 10 liczb całkowitych
  int i;

  for(i = 0; i < 10; i++) cout << i << ”  - ” << T[i] << endl;

  return 0;
}
```

Różnica pomiędzy **T[i]** a **T.at(i)** jest taka, iż funkcja składowa **at()** sprawdza, czy element o indeksie ***i*** leży wewnątrz tablicy. Jeśli nie, to zostaje zgłoszony wyjątek.

Dane w tablicy możemy umieszczać na kilka sposobów:

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

using namespace std;

int main()
{
  vector<int> T(10,5);          // tworzymy tablicę 10 liczb całkowitych
  unsigned int i;

  T[5]     = 25;                // komórka o podanym indeksie przyjmuje nową wartość
  T.at(6) = 315;                // komórka o podanym indeksie przyjmuje nową wartość + sprawdzenie
indeksu, czy wskazuje komórkę tablicy
```

T.push_back(1999); // na końcu tablicy zostaje dopisany nowy element o podanej wartości. Liczba komórek w tablicy zostaje zwiększona.

```
  for(i = 0; i < T.size(); i++) cout << T.at(i) << endl;

  return 0;
}
```

`T[indeks] = wartość;` **komórka o podanym indeksie przyjmuje nową wartość. Indeks nie jest sprawdzany.**

`T.at(indeks) = wartość;` **jak wyżej, jednakże indeks jest sprawdzany, czy wskazuje komórkę w tablicy. Jeśli nie, powstanie wyjątek.**

`T.push_back(wartość);` **na końcu tablicy zostaje dopisany nowy element o podanej wartości. Liczba komórek w tablicy zostaje zwiększona.**

Ponieważ rozmiar tablicy T może się dynamicznie zmieniać, to aktualną liczbę komórek poda nam funkcja składowa T.size().

Kolejny przykład pokazuje, jak można stworzyć prosty stos:

Stos (ang. Stack) – liniowa struktura danych, w której dane dokładane są na wierzch stosu i z wierzchołka stosu są pobierane (bufor typu **LIFO**, *Last In, First Out*; *ostatni na wejściu, pierwszy na wyjściu*).

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

using namespace std;

int main()
{
  vector<int> T;                // tworzymy pustą tablicę
  int i;
```

```
  for(i = 1; i < 16; i++) T.push_back(i);

  while(!T.empty())  // sprawdzamy za pomocą funkcji składowej T.empty(), czy tablica jest pusta
  {
     cout << T.back() << endl;      // wyświetlamy w oknie konsoli ostatni element
     T.pop_back();                  // usuwamy z tablicy ostatni element
  }

  return 0;
}
```

W pętli dopisujemy do końca tablicy kolejne liczby naturalne od 1 do 15. W drugiej pętli sprawdzamy za pomocą funkcji składowej T.empty(), czy tablica jest pusta. Jeśli nie, to wyświetlamy w oknie konsoli ostatni element - funkcja składowa T.back(), a następnie element ten usuwamy z tablicy za pomocą funkcji składowej T.pop_back().

Z biblioteką STL wiąże się pojęcie **iteratora**. Iterator jest klasą wskaźnika, który wskazuje elementy np. tablicy. Zmienną typu iterator dla tablic **vector** tworzymy następująco:

```
vector<typ_elementów>::iterator nazwa iteratora;
```

Do obsługi iteratorów w klasie vector mamy dwie funkcje składowe:

`begin()` - zwraca iterator na pierwszy element tablicy
`end()` - zwraca iterator wskazujący poza ostatni element tablicy

Poniższy przykład pokazuje wykorzystanie iteratorów do przeglądania tablicy.

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

using namespace std;

int main()
{
  vector<int> T;                            // tworzymy pustą tablicę
  vector<int>::iterator it;                 // tworzymy iterator
  int i;

  for(i = 1; i < 10; i++) T.push_back(i*i);  // wypełniamy tablicę

  for(it = T.begin(); it != T.end(); it++)   // przeglądamy tablicę
    cout << *it << endl;                    // *it jest elementem tablicy

  return 0;
}
```

Iteratory wykorzystuje się w operacjach usuwania elementów z tablicy.

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

using namespace std;

int main()
{
  vector<int> T;               // tworzymy pustą tablicę
  vector<int>::iterator it;    // tworzymy iterator
  unsigned int i;

  for(i = 0; i < 10; i++) T.push_back(i*10); // wypełniamy tablicę liczbami

  it = T.begin();              // ustawiamy iterator na pierwszy element tablicy
  T.erase(it+5);               // usuwamy z tablicy element o indeksie 5
  for(i = 0; i < T.size(); i++) cout << T[i] << endl;

  return 0;
}
```

Drugi przykład pokazuje sposób **usuwania ciągu kolejnych elementów**:

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

using namespace std;

int main()
{
  vector<int> T;                       // tworzymy pustą tablicę
  vector<int>::iterator it;            // tworzymy iterator
  unsigned int i;

  for(i = 0; i < 10; i++) T.push_back(i*10);  // wypełniamy tablicę liczbami

  it = T.begin();              // ustawiamy iterator na pierwszy element tablicy
  T.erase(it+2,it+5);          // usuwamy z tablicy elementy o indeksach od 2 do 4
  for(i = 0; i < T.size(); i++) cout << T[i] << endl;

  return 0;
}
```

Iteratory można również wykorzystywać do **wstawiania nowych elementów** do tablicy. Wstawianie polega na tym, iż elementy tablicy od pozycji iteratora zostają przesunięte w górę, a w powstałe w ten sposób miejsce jest wstawiany nowy element.

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

using namespace std;

int main()
{
  vector<int> T;                      // tworzymy pustą tablicę
  vector<int>::iterator it;           // tworzymy iterator
  unsigned int i;

  for(i = 0; i < 10; i++) T.push_back(i); // wypełniamy tablicę liczbami

  it = T.begin()+5;                   // ustawiamy iterator na element o indeksie 5
  it = T.insert(it,100);              // do tablicy wstawiamy 100
  for(i = 0; i < T.size(); i++) cout << T[i] << endl;

  return 0;
}
```

Zwróć uwagę na instrukcję:

```
it = T.insert(it,100);
```

W tym programie nie ma znaczenia, co się stanie dalej z iteratorem it, ponieważ po wstawieniu nowego elementu już z niego nie korzystamy. Funkcja składowa insert() w przypadku wstawiania jednego elementu zwraca iterator do pozycji, na której ten element został wstawiony. Jest to bardzo istotne, gdy chcesz dalej korzystać z iteratora, np. do wstawiania kolejnych elementów, ponieważ adres tablicy może ulec zmianie w pamięci. Wstawienie elementu powoduje wzrost obszaru pamięci zajmowanego przez tablicę. Jeśli przekroczy on zarezerwowaną wartość, to zostaje utworzony nowy, większy obszar i cała zawartość tablicy jest kopiowana do tego nowego obszaru. Stary obszar jest zwracany do puli systemowej. Powoduje to zmianę adresów wszystkich komórek tablicy i unieważnienie twojego iteratora - przestaje on wskazywać cokolwiek. Jeśli jednak odświeżymy jego zawartość wynikiem zwracanym przez insert(), to iterator pozostanie aktualny nawet po zmianie adresu tablicy. Pamiętaj o tym, aby uniknąć w przyszłości niespodzianek!

Kolejny przykład pokazuje wstawianie do tablicy wielu komórek o zadanej wartości. W tym przypadku iterator staje się nieaktualny i należy go bezwzględnie odświeżyć za pomocą funkcji składowej begin() - insert() dla wielu wstawień nic nie zwraca!

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

int main()
{
  vector<int> T;             // tworzymy pustą tablicę
  vector<int>::iterator it; // tworzymy iterator
  unsigned int i;

  for(i = 0; i < 10; i++) T.push_back(i); // wypełniamy tablicę liczbami

  it = T.begin()+5;   // ustawiamy iterator na element o indeksie 5
  T.insert(it,5,100); // do tablicy wstawiamy 5 wartości 100
  for(it = T.begin(); it != T.end(); it++) cout << *it << endl;

  return 0;
}
```

Podsumowanie

`#include <vector>` - należy wstawić na początku programu, aby uzyskać dostęp do klasy **vector**.

`vector<typ> nazwa;`	- tworzy pustą tablicę
`vector<typ> nazwa(n,m);`	- tworzy tablicę *n* komórek o wartości *m*
`vector<typ> nazwa(nazwa);`	- tworzy tablicę, która jest kopią innej tablicy
`nazwa.size()`	- podaje liczbę komórek w tablicy
`nazwa.empty()`	- zwraca true, jeśli tablica jest pusta
`nazwa[i]`	- element o indeksie *i*
`nazwa.at(i)`	- element o indeksie *i* ze sprawdzaniem przynależności do tablicy
`nazwa.push_back(m)`	- dodaje na końcu tablicy element *m*
`nazwa.front()`	- pierwszy element tablicy
`nazwa.back()`	- ostatni element tablicy
`nazwa.pop_back()`	- usuwa ostatni element tablicy
`vector<typ>::iterator it;`	- tworzy iterator do elementów tablicy
`nazwa.begin()`	- zwraca iterator do pierwszego elementu tablicy
`nazwa.end()`	- zwraca iterator wskazujący poza ostatni element tablicy
`nazwa.erase(it)`	- usuwa element na pozycji iteratora *it*
`nazwa.erase(it1,it2)`	- usuwa wszystkie elementy od pozycji iteratora it_1 do pozycji tuż przed iteratorem it_2 - element na pozycji iteratora it_2 pozostaje w tablicy
`nazwa.insert(it,m)`	- wstawia *m* na pozycji wskazywanej przez iterator *it*. Zwraca iterator do wstawionego elementu
`nazwa.insert(it,n,m)`	- wstawia od pozycji iteratora *it* *n* elementów o wartości *m*. Iterator traci ważność.
`nazwa.clear()`	- usuwa wszystkie elementy z tablicy

Wprowadzanie/wyprowadzanie danych

Dane dla programu zwykle muszą być odczytywane ze zewnętrznego źródła - konsoli lub pliku. W takim przypadku nie wiemy z góry (tzn. w trakcie pisania programu) ile ich będzie. Narzucającym się rozwiązaniem jest zastosowanie tablic dynamicznych. Ze źródła danych odczytujemy rozmiar tablicy, tworzymy tablicę dynamiczną o odpowiednim rozmiarze, a następnie wczytujemy do jej komórek poszczególne dane.

Poniżej podajemy sposoby odczytu zawartości tablicy z konsoli. Sposób ten jest bardzo ogólny. Wykorzystanie standardowego wejścia jako źródła danych daje nam kilka możliwości wprowadzania danych:

1. Dane podajemy bezpośrednio z klawiatury. Sposób skuteczny i prosty dla niedużego zbioru danych. Jednakże przy większej ich liczbie staje się bardzo uciążliwy.
2. Skopiowanie danych poprzez schowek. Procedura postępowania jest następująca:

   - tworzymy w notatniku Windows (aplikacja zawsze pod ręką) odpowiedni zbiór danych
   - zbiór kopiujemy do schowka (zaznaczamy całość Ctrl-A i naciskamy Ctrl-C)
   - uruchamiamy program
   - klikamy prawym przyciskiem myszki w pasek tytułowy okna konsoli
   - z menu kontekstowego wybieramy polecenie Edytuj → Wklej
   - gotowe

3. Przekierowanie standardowego wejścia z konsoli na plik na dysku. W tym przypadku program będzie pobierał dane z pliku, a nie z klawiatury. Aby to uzyskać uruchamiamy program w oknie konsoli następująco:

   nazwa_programu < nazwa_pliku_wejściowego

   Na przykład nasz program nazywa się szukaj.exe, a plik nosi nazwę dane.txt. Odpowiednie polecenie odczytu danych z pliku przez nasz program wygląda następująco:

   szukaj < dane.txt

   To rozwiązanie umożliwia również zapis danych wynikowych nie na ekran konsoli, lecz do pliku na dysku. W tym celu wystarczy wydać polecenie:

   nazwa_programu > nazwa_pliku_wynikowego

   Wejście i wyjście można przekierować w jednym poleceniu. Np. nasz program szukaj może odczytać dane wejściowe z pliku dane.txt, a wyniki swojej pracy umieścić w pliku wyniki.txt. W tym celu wydajemy takie oto polecenie:

   szukaj < dane.txt > wyniki.txt

   Jeśli często korzystasz z takich opcji uruchamiania programu, to zamiast wpisywać polecenie z klawiatury, można stworzyć sobie prosty plik wsadowy (ang. batch file), w którym umieszczamy niezbędne polecenia. Plikowi można nadać prostą nazwę, np. !.cmd. Wtedy w celu uruchomienia zawartych w nim poleceń wystarczy wpisać !. Oczywiście plik wsadowy należy umieścić w katalogu projektowym, ale to chyba już wiesz. Poniżej podaję przykładową zawartość takiego pliku wsadowego:

   ```
   @echo off
   cls
   echo DANE WEJSCIOWE:
   echo.
   type dane.txt
   echo.
   ```

```
prg < dane.txt > wyniki.txt
echo WYNIKI:
echo.
type wyniki.txt
echo.
```