klasa vector

W języku C++ dostępna jest biblioteka STL (ang. *Standard Template Library*), która dostarcza wiele przydatnych klas i funkcji. Biblioteka STL zawiera między innymi tak zwane klasy kontenerów czyli obiektów, które mogą przechowywać inne obiekty dowolnego typu. Jedną z takich najczęściej wykorzystywanych klas kontenerowych z biblioteki STL jest klasa `vector`. W programowaniu często wykorzystujemy tablice dynamiczne do przechowywania danych, których ilości nie jesteśmy w stanie dokładnie przewidzieć w trakcie pisania programu. Tablice dynamiczne w języku C++ tworzymy przy pomocy wskaźników oraz operatora `new`. Jednakże tak utworzona tablica posiada bardzo ograniczone możliwości - wszelkie operacje na danych programista musi zdefiniować sam. Ponadto, ilość obiektów przechowywana w takiej tablicy jest z góry określona i nie może ulegać zmianie w czasie wykonania programu. Klasa `vector`, rozwiązuje te problemy gdyż posiada wiele przydatnych metod ułatwiających jej użycie a ponadto ilość obiektów przechowywana w wektorze może się dynamicznie zmieniać w czasie wykonania programu. Ważnym jest by pamiętać, że obiekty w kontenerze `vector` są przechowywane jeden za drugim podobnie jak w klasycznej tablicy w języku C/C++. Oznacza to, że z jednej strony można do nich uzyskać szybki dostęp za pomocą indeksu obiektu w wektorze, z drugiej zaś strony jeśli w wektorze przechowywana jest bardzo duża ilość obiektów to będą one zajmować bardzo dużą ilość pamięci. Dla przykładu, jeśli w wektorze mamy miliard liczb zmiennoprzecinkowych typu `float`, z których każda zajmuje cztery bajty, to całkowita ilość pamięci RAM potrzebnej dla tego wektora wyniesie $4 \cdot 10^{9} \approx 4GiB$.

Aby użyć w kodzie źródłowym klasę `vector` należy dołączyć odpowiedni plik nagłówkowy:

```
#include <vector>
```

Obiekt klasy `vector` przechowujący liczby całkowite typu `int` tworzy się w następujący sposób:

```
vector<int> nazwa_obiektu;
```

Bezpośrednio po utworzeniu, wektor nie zawiera żadnych obiektów. Indeksowanie elementów w wektorze zaczyna się od 0. Poniżej kilka przydatnych metod z klasy `vector`:

`[n]`	umożliwia dostęp do *n*-tego elementu wektora
`size()`	zwraca bieżącą ilość obiektów w wektorze
`begin()`	zwraca wskaźnik (iterator) do pierwszego obiektu
`end()`	zwraca wskaźnik (iterator) do pierwszego obiektu po ostatnim
`push_back(obj)`	umieszcza obiekt `obj` na końcu wektora
`insert(iterator position, obj)`	wstawia obiekt `obj` na miejsce wskazywane przez `position`
`erase(iterator position)`	usuwa obiekt na miejscu wskazywanym przez `position`
`resize(n)`	ustawia rozmiar wektora na *n* elementów

Dla przykładu, dodanie nowej wartości (np. 5) na końcu wektora liczb całkowitych będzie mieć postać

```
nazwa_wektora.push_back(5);
```

zaś wstawienie nowego obiektu na drugiej pozycji licząc od początku

```
nazwa_wektora.insert( nazwa_wektora.begin() + 1, nowy_obiekt );
```

Elementy znajdujące się w kontenerze vector możemy wyświetlić na kilka sposobów:

```
vector<int> v {1,2,3,4,5,6,7,8,9,10};
// pętla for
for(int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
// range-based loop
for(auto i : v) {
    cout << i << endl;
// iteratory
for(vector<int>::iterator iter = v.begin(); iter < v.end(); ++iter) {
    cout << *iter << endl; }
```

Pierwsza metoda ta opiera się na tradycyjnej pętli `for`. Po prostu przechodzimy po kolei przez wszystkie elementy kontenera. Druga metoda to tak zwany *range-based loop* (wprowadzony w C++11). Zmienna `i` przechowuje bieżący element kontenera. Trzeci sposób wykorzystuje tak zwane iteratory czyli obiekty są wskaźnikami do elementu w kontenerze.

W pliku nagłówkowym algorithm jest zdefiniowane wiele przydatnych funkcji ułatwiających wykorzystanie wektorów oraz innych kontenerów z bilioteki standardowej STL. Poniżej kilka najbardziej przydatnych funkcji.

- `count_if` - zlicza ilość elementów spełniających zadany warunek

```
bool IsOdd (int i) { return ((i%2)==1); }
vector<int> v = {7, -1, 9, 0, 10, 11, 15, 3};
int mycount = count_if (v.begin(), v.end(), IsOdd);
cout << "myvector contains " << mycount  << " odd values.\n";
```

- `find` - znajduje pierwsze wystąpienie wskazanego elementu w przedziale `[fisrt, last)`. Jeśli element nie występuje w tym przedziale to zwracany jest iterator `last`

```
vector<int> v = {7, -1, 9, 0, 10, 11, 15, 3};
auto it = find(v.begin(), v.end(), 11); // znajdź element o wartości 11 w wektorze v
if (it != myvector.end())
      cout << "Element found in v: " << *it;
  else
      cout << "Element not found in v";
```

- `find_if` - znajduje pierwsze wystąpienie elementu w przedziale `[fisrt, last)` spełniającego zadany warunek. Jeśli element nie występuje w tym przedziale to zwracany jest iterator `last`

```
bool IsOdd (int i) { return ((i%2)==1); }
vector<int> v = {7, -1, 9, 0, 10, 11, 15, 3};
vector<int>::iterator it = find_if (v.begin(), v.end(), IsOdd);
cout << "The first odd value is " << *it << '\n';
```

- `sort` - sortuje elementy kontenera w rosnącej kolejności. Jeśli chcemy posortować kontener w malejącej kolejności to należy posłużyć się dodatkową funkcją lambda, która porównuje odpowiednio dwa elementy wektora

```
vector<int> v = {7, -1, 9, 0, 10, 11, 15, 3};
sort(v.begin(), v.end()); // sortuj wektor v w rosnącej kolejności
sort(v.begin(), v.end(), [](int x, int y)->bool { return x > y; }); // sortuj wektor w
malejącej kolejności
```

- `transform` - wykonuje wskazaną operację na danym zakresie

```
vector<int> in = {1, 4, 7, 9, 0, -1};
vector<int> out;
out.resize( in.size() );
// zwiększa o jeden każdy element wektora in
```

```
transform(in.begin(), in.end(), out.begin(), [](int x)->int { return ++x; });
// dodaj odpowiednie elementy wektora in oraz out i wynik zapisz w wektorze out
transform(in.begin(), in.end(), out.begin(), out.end(), [](int x, int y)->int { return x
+ y; });
```

- `unique` - usuwa duplikaty w kontenerze

```
vector<int> v = {7, -1, 9, 9, 3, 5, 5, 7};
vector<int>::iterator it;
it = unique (v.begin(), v.end());
v.resize(distance(v.begin(),it));
```

- `reverse` - odwraca kolejność elementów w kontenerze

```
vector<int> v;
for (int i=1; i<10; ++i) v.push_back(i); // 1 2 3 4 5 6 7 8 9
reverse(v.begin(), v.end());   // 9 8 7 6 5 4 3 2 1
```

- `rotate` - rotacja elementów w wektorze w taki sposób, że element wskazany przez middle staje się pierwszym

```
vector<int> v;
for (int i=1; i<10; ++i) v.push_back(i); // 1 2 3 4 5 6 7 8 9
rotate(v.begin(),v.begin()+3,v.end()); // 4 5 6 7 8 9 1 2 3
```

- `random_shuffle` - losowo zmień kolejność elementów w kontenerze

```
int myrandom (int i) { return rand()%i;}
vector<int> v;
srand ( unsigned ( time(0) ) );
for (int i=1; i<10; ++i) v.push_back(i); // 1 2 3 4 5 6 7 8 9
random_shuffle ( v.begin(), v.end() ); //using built-in random generator
random_shuffle ( v.begin(), v.end(), myrandom); //using myrandom
```

- `next_permutation` - kolejna permutacja elementów kontenera

```
int myints[] = {1,2,3};
sort (myints,myints+3);
cout << "The 3! possible permutations with 3 elements:\n";
  do {
      cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
  } while ( next_permutation(myints,myints+3) );
  cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
```

1. Załóżmy, że piszesz program, który zarządza listą zakupów. Każdy zakupiony przedmiot jest reprezentowany jako obiekt klasy `string` w kontenerze `vector`. Program wymaga utworzenia funkcji o nazwie `print`, która wypisuje na ekranie listę zakupów. W funkcji `main` wykonaj następujące zadania:

    a. Utwórz pusty wektor obiektów klasy `string`. Wypisz wektor na ekranie.

    b. Dodaj do wektora obiekty “wino”, “buraki”, “banany”, “cukier”, “czekolada”, marchew”.

    c. Wypisz na ekranie ostatni element i usuń go z wektora.

    d. Dodaj obiekt “kawa” jako trzeci w kolejności element w wektorze.

    e. Napisz pętlę, która znajduje w wektorze obiekt “cukier” a następnie zastępuje go obiektem “cukierki”.

    f. Napisz pętlę, która znajduje w wektorze obiekt “czekolada” a następnie usuwa go.

    g. Posortuj wektor za pomocą funkcji `sort` z pliku nagłówkowego `algorithm`.

Wypisz na ekranie stan wektora po każdym z powyższych podpunktów.

2. Napisz program, który obliczy sumę dwóch wektorów. W tym celu stwórz w funkcji `main` dwa wektory liczby zmiennoprzecinkowych. Wypełnij je losowymi liczbami. Następnie napisz funkcję o nazwie `sum` której argumentami są dwa wektory liczb zmiennoprzecinkowych i która zwraca ich sumę. W funkcji `main` wywołaj funkcję `sum` dla utworzonych dwóch wektorów i wypisz na ekranie wynik jej działania.

3. Kod Graya, zwany również kodem refleksyjnym to bezwagowy i niepozycyjny kod, który charakteryzuje się tym, że dwa kolejne *słowa kodowe* różnią się tylko stanem jednego bitu. Jest również kodem cyklicznym, bowiem ostatni i pierwszy wyraz tego kodu także spełniają wyżej wymienioną zasadę. Algorytm tworzenia *n*-bitowego kodu Graya jest następujący:

    a. Niech L_1 oznacza listę *n-1*-bitowych kodów Graya. Utwórz kolejną listę L_2, która zawiera kody Graya w odrtonej kolejności.

    b. Zmodyfikuj listę L_1 poprzez dodanie cyfry `0` na początku każdego kodu.

    c. Zmodyfikuj listę L_2 poprzez dodanie cyfry `1` na początku każdego kodu.

    d. Połącz listę L_1 and L_2 w jedną całość. Połączona lista zawiera *n*-bitowe kody Graya.

    Na przykład, utwórzmy *2*-bitowy kod Graya. Zaczynamy od kodu 1-bitowego, który ma postać L_1={`0, 1`}. Lista L_2 będzie mieć postać L_2={`1, 0`}. Następnie, zmodyfikowane listy L_1 oraz L_2: L_1={`00, 01`}, L_2={`11, 10`}. Ostatecznie połączona lista zawierająca *2*-bitowy kod Graya ma postać L_1+L_2={`00, 01, 11, 10`}. Napisz program, który wczyta z klawiatury liczbą całkowitą *n*. Następnie wypisze na ekranie *n*-bitowe kody Graya.

4. Zastanów się jak stworzyć w pamięci komputera macierz za pomocą obiektów klasy `vector`. Następnie napisz funkcję o nazwie `sum` oraz `mul`, która obliczą odpowiednio sumę i iloczyn dwóch macierzy. Napisz również funkcję o nazwie `print`, która wypisze macierz na ekranie. W funkcji `main` stwórz dwie macierze i wypełnij je losowymi liczbami a następnie wypisz je na ekranie. Oblicz iloczyn tych macierzy i wypisz wynik na ekranie.

5. Napisz program, który wczyta z klawiatury dowolną ilość liczb całkowitych. Wprowadzanie kolejnych liczb ma się zakończyć po wprowadzeniu z klawiatury liczby -1. Wprowadzone liczby zapisz w wektorze. Następnie, posługując się funkcjami z pliku nagłówkowego algorithm wykonaj następujące czynności:

    a. usuń z wektora powtarzające się wartości,

    b. policz i wyświetl na ekranie liczbę nieparzystych elementów wektora,

    c. Zmień znak każdego elementu wektora na przeciwny,

    d. posortuj wszystkie elementy wektora w malejącej kolejności.

6. Napisz program, który wczyta z klawiatury lub pliku jedną linię tekstu (`str`) a następnie wczyta drugi string (`sep`), który będzie separatorem. Następnie napisz funkcję `vector<string> tokenizer(const string& str, const string& sep)`, która zwraca rozdzielone separatorem `sep` części stringu `str`. Wypisz na ekranie uzyskane części (tokeny) stringu `str`.