Kolekcja vector z SL

Niemal od początku semestru używamy kolekcji vector z biblioteki standardowej i warto byłoby z nią zapoznać się nieco bliżej.
Najpierw warto przyjrzeć się możliwościom iterowania, czyli przechodzenia po elementach, w wektorze.
Standardowo dostajemy się do elementów za pomocą operatora indeksowania:

```
vector<int> vint = {1, 3, 5, 7, 11};
for (unsigned int idx = 0; idx < vint.size(); ++idx)
    cout << vint[idx] << ' ';
cout << endl;
```

Iteratory w kolekcji

Inną możliwością przechodzenia przez elementy kolekcji jest wykorzystanie iteratorów – obiekty tej klasy umożliwiają „wskazywanie” elementów kolekcji, dostawanie się do ich wartości i wykonywanie innych operacji.
Przejście przez elementy kolekcji z pełną specyfikacją wygląda tak:

```
vector<int> vint = {1, 3, 5, 7, 11};
for (vector<int>::iterator cit = vint.begin();
  cit != vint.end(); ++cit)
    cout << *cit << ' ';
cout << endl;
```

Iteratory w kolekcji

Na co warto zwrócić uwagę w poprzednim przykładzie:

`begin()`	daje iterator skojarzony z pierwszym elementem kolekcji – metoda klasy `vector`
`end()`	daje iterator za ostatnim elementem kolekcji (dziwne, ale ta wartość służy przekazaniu informacji: *przeszedłeś już przez wszystkie elementy kolekcji*) – metoda klasy `vector`
`!=`	sprawdza, czy dwa iteratory odnoszą się do różnych elementów kolekcji
`++`	przesuwa iterator na następny element kolekcji
`*`	daje wartość elementu kolekcji skojarzonego z iteratorem

Ta piątka operacji jest do zapamiętania, bo wykorzystamy ją z każdą kolekcją i iteratorem!

Jeszcze inne przejście przez elementy

Jeśli zależy nam po prostu na przejściu przez kolejne elementy kolekcji, bez innych niecnych zamiarów, możemy wykorzystać jeszcze prostszą formę pętli for:

```
vector<int> vint = {1, 3, 5, 7, 11};
for (int val : vint)
    cout << val << ' ';
cout << endl;
```

Warto wiedzieć, że zamiast kopii wartości (`int val`) możemy wziąć referencję na wartość (`int& val`), bo mamy zamiar modyfikować te wartości lub ustaloną referencję (`const int& val`), bo chcemy uniknąć kopiowania.

`vector` – parę użytecznych operacji

Przy działaniach z wektorami przydadzą się niewątpliwie:

`size()`	znana już metoda dająca liczbę elementów w kolekcji,
`push_back()`	dołożenie na końcu kolekcji nowej wartości
`pop_back()`	usunięcie z kolekcji ostatniej wartości
`front()`	referencja do pierwszego elementu kolekcji
`back()`	referencja do ostatniego elementu kolekcji

To jest jedynie część możliwości wektora (i wielu innych kolekcji). Pełniejszy obraz można sobie wyrobić zaglądając na stronę:
http://www.cplusplus.com/reference/vector/vector/

Iteratory i algorytmy

Iteratory otwierają piękne możliwości skorzystania z różnych algorytmów zdefiniowanych w pliku nagłówkowym `algorithm`.

Do dzisiejszego zadania przydadzą się dwa z nich:

`find()` i `sort()`. W obu przypadkach pierwsze dwa parametry to iteratory określające zakres przeszukiwania lub sortowania. Dla całej kolekcji idealne wartości to `begin()` i `end()`. W find ostatni parametr, to szukana wartość. Zatem sprawdzenie, czy w kolekcji mamy np. wartość 8 wygląda tak:

```
auto it = find(vint.begin(), vint.end(), 8);
if (it != vint.end())
   // znalezione, można dzialac
```

Zwróćcie uwagę, jak sprawdza się, czy element został znaleziony.

Iteratory i algorytmy

Ciekawiej wygląda sprawa z sort(), bo ten algorytm ma dwie wersje: z dwoma i trzema argumentami.

Sekwencja:

```
vector<float> vflt = {8.9, 3.4, 1.7, 14.2, 2.3};
sort(vflt.begin(), vflt.end());
```

zostawi nas z `vflt` posortowanym w porządku niemalejącym.

Trzeci parametr jest wskazaniem na funkcję porównującą – domyślnie jest używany operator <. Zatem:

```
vector<float> vflt = {8.9, 3.4, 1.7, 14.2, 2.3};
sort(vflt.begin(), vflt.end(), greater_than);
```

Da mi wektor w porządku nierosnącym, o ile mam funkcję `greater_than` w postaci:

```
bool greater_than(float a, float b)
    { return a > b; }
```

Zadanie 7

Zadanie polega na zaimplementowaniu klasy `word_counter` z wykorzystaniem kolekcji `vector` z biblioteki standardowej oraz klasy `entry` z poprzedniego zadania.

Klasa `word_counter` powinna udostępniać:

- Konstruktor domyślny tworzący pusty licznik (konkretyzację szablonu `vector` dla `entry`).
- Metodę `add_word` dodającą do bieżącego licznika słowo, które jest parametrem metody.
- Metodę `size` dającą liczbę różnych słów w liczniku.
- Metody `sort_alpha` i `sort_count` sortujące zawartość licznika alfabetycznie lub według częstości wystąpień.
- Operator wyprowadzenia na strumień wyjściowy.
- Metodę `get_vector` zwracającą ustaloną referencję na wektor obiektów typu `entry`.

Dostarczanie rozwiązania

Rozwiązaniem zadania są pliki **`entry.h`** i **`word_counter.h`** (każdy plik zawiera definicję klasy z implementacją wszystkich metod).

Na początku każdego pliku trzeba umieścić w komentarzu imię, nazwisko i nr albumu autora.

Rozwiązanie należy załadować w `Moodle` do:

15 grudnia 2021 23:59