Zadanie 10

Zadanie polega na zaimplementowaniu klasy `word_counter` z wykorzystaniem kolekcji `map` z biblioteki standardowej.
Użycie gotowej kolekcji uprości z jednej strony zadanie, ale skomplikuje także dostęp do naliczonych słów według malejącej liczby wystąpień.
Do tego celu trzeba będzie zaimplementować w word_counter swój własny, dość specyficzny iterator.

Uwaga: ze względu na złożoność klasa iteratora jest punktowana ekstra. Za wersję podstawową, która działa przy założeniu, że liczby powtórzeń wyrazów są unikalne, można dodatkowo uzyskać 4 punkty.

Klasa `word_counter`

Klasa `word_counter` powinna udostępniać:

- Konstruktor domyślny tworzący pusty licznik (konkretyzację szablonu `map` dla klucza `string` i wartości `int`)
- Metodę `add_words` dodającą do bieżącego licznika słowa ze strumienia wejściowego, który jest parametrem metody
- Metodę `size` dającą liczbę różnych słów w liczniku
- Operator wyprowadzenia na strumień wyjściowy
- Klasę iteratora `desciter`, który będzie przechodzić przez kolejne elementy kolekcji począwszy od elementu o największej liczbie wystąpień
- Metodę `begin` dającą iterator skojarzony z elementem o największej liczbie wystąpień
- Metodę `end` dającą iterator „pusty” (tzn. za ostatnim elementem)

Klasa iteratora

Ponieważ iterator implementujemy po raz pierwszy, a nie jest to zupełnie klasyczny iterator, przyda się parę dodatkowych informacji.
Pierwsze wnioski można wyciągnąć z kodu testującego:

```
// word_counter::desciter it; // Błąd kompilacji
word_counter::desciter it = wc.descbegin();
while (it != wc.descend())
{
    cout << *it << endl;
    ++disp_count;
    ++it;
    if (disp_count >= limit)
        break;
}
```

Przede wszystkim, iterator jest klasą wewnętrzną klasy `word_counter`. Zamysł jest taki, żeby wykorzystać iterator zdefiniowany dla map (mimo innej kolejności przechodzenia).

Klasa iteratora

W sekcji `public` klasy `word_counter` warto pomieścić:

```
class desciter
{
    const std::map<std::string, int>& aparent;
    std::map<std::string, int>::const_iterator it;
    desciter moveto(int upper_limit);
public:
    desciter(const std::map<std::string, int>& parent,
        const std::map<std::string, int>::const_iterator& cur)
        : aparent(parent), it(cur)
    {
        if (it != aparent.end())
            moveto(std::numeric_limits<int>::max());
    }

    const std::pair<std::string, int> operator*() const;
    desciter operator++();
    bool operator!=(const desciter& rha)
    {
        return it != rha.it;
    }
};
```

Klasa iteratora

Jak widać, nie można utworzyć niezainicjowanego iteratora (skoro jest konstruktor z parametrami, to kompilator nie generuje bezparametrowego; konstruktor kopiujący – tak), a zestaw operacji jest podstawowy (można się pokusić o dodanie operatora ==, choć nie jest używany).
Definicje, które dają się zaimplementować jedną instrukcją, są od razu zaimplementowane. Te wymagające większego zachodu implementowałbym w `word_counter.cpp`.
W klasie iteratora mam referencję do obiektu `counter` z klasy macierzystej (składowa `aparent`). Do wyszukiwania kolejnych elementów będzie potrzebny dostęp do początku i *końca* kolekcji – dzięki referencji będzie można wywoływać funkcje `begin()` i `end()` dające dostęp do tych iteratorów szablonu `map`.

Klasa iteratora

W operatorach * i ++ należy wykrywać sytuację, w której iterator jest „*(za) końcem*”. Jak zwykle wypada wygenerować wtedy wyjątek, ale w tym zadaniu wystarczy wyświetlenie na konsoli komunikatu (w testach taka sytuacja nie powinna wystąpić).
Można się domyślać, że główna praca do wykonania jest w funkcji `moveto`. Ta funkcja przesuwa iterator na element o największej liczbie powtórzeń, która jest jednocześnie mniejsza od argumentu `upper_limit`. Poszukiwanie nowej pozycji oznacza przejście wszystkich elementów kolekcji od aktualnej do bezpośrednio ją poprzedzającej. To może oznaczać przejście przez koniec licznika i wtedy przyda się możliwość wywołania `begin()` i `end()` na składowej `aparent`.

Klasa iteratora – tworzenie

Obiekty `desciter` powinny być tworzone wyłącznie za pomocą funkcji `descbegin()` i `descend()` klasy `word_counter`. Ich implementacja powinna być oczywista:

```
desciter descbegin() const
{
    return desciter(counter, counter.begin());
}
desciter descend() const
{
    return desciter(counter, counter.end());
}
```

Przed uruchomieniem testu na większym tekście warto sprawdzić działanie iteratora na małym przykładzie:

```
0: [quattro 4]
1: [tre 3]
2: [due 2]
3: [uno 1]
```

`word_counter` – testy

Wynik podstawowych testów licznika słów się nie zmienił:

```
[]
[]
[[alina 1] ]
[[alina 1] ]
[[halina 1] [helena 2] ]
[[helena 2] [halina 1] ]
[[halina 2] [hanna 1] [helena 2] ]
[[halina 2] [helena 2] [hanna 1] ]
[[halina 3] [hanna 1] [helena 2] ]
[[halina 3] [helena 2] [hanna 1] ]
```

Test duży wymaga jednak zaimplementowania iteratora, zatem w wersji podstawowej tego zadania testy będą bardzo skromne.

Technikalia

Klasa `word_counter` powinna znaleźć się w pliku `word_counter.h` (definicja) i `word_counter.cpp` (implementacja klasy i funkcji pomocniczych).

Do nieco poważniejszego testu wykorzystam *The Voyage of Beagle* Karola Darwina (źródłem tekstu jest Projekt Gutenberg - http://www.gutenberg.org)

Warto porównać aktualne osiągi z czasem zliczania słów w poprzednim zadaniu.

Test większy – wyniki

```
The Voyage of Beagle by Charles Darwin most frequent words

[the 16924]
[of 9429]
[and 5765]
[a 5327]
[in 4287]
[to 4091]
[is 2413]
[it 1998]
[that 1940]
[on 1868]
[i 1815]
[was 1795]
[as 1650]
[with 1636]
[which 1619]
[by 1609]
[at 1433]
[from 1402]
[this 1380]
[are 1180]
```

Terminy i wyniki

Do dostarczenia są:

`word_counter.h` – definicja klasy

`word_counter.cpp` – implementacja klasy i funkcji pomocniczych

Rozwiązanie należy wysłać do mnie do:

1 lutego 2018