Instrukcja laboratoryjna nr 2

Programowanie w języku C 2

(C++ poziom zaawansowany)

Kontenery i iteratory.

Wykorzystanie kontenerów w praktyce.

dr inż. Jacek Wilk-Jakubowski

mgr inż. Maciej Lasota

dr inż. Tomasz Kaczmarek

Wstęp

W poprzednim semestrze studenci kierunku informatyka poznali jeden kontener będący częścią standardowej biblioteki szablonów języka C++ (ang. Standard Template Library STL), a mianowicie kontener *vector*. Teraz przyszedł czas na poznanie kolejnych.

Lista dwukierunkowa

Lista dwukierunkowa, czyli klasa *list* jest kolejnym kontenerem udostępnianym przez bibliotekę STL. Dostęp do elementów listy dwukierunkowej jest możliwy tylko przy użyciu iteratorów w odróżnieniu od wektora, do którego elementów dostęp jest możliwy zarówno przy pomocy iteratorów jak i przez podanie indeksu. Dodawanie i usuwanie elementów listy jest szybsze niż dodawanie i usuwanie elementów wektora, poza tym operacje te w przypadku listy realizowane w stałym czasie w odróżnieniu od wektora, dla którego czas dostępu zależy od usytuowania elementu w kontenerze. Trzecią, dość istotną różnicą pomiędzy klasą *list* oraz klasą *vector* jest to, że adresy elementów listy są niezmienne, natomiast adresy elementów wektora ulegają dość częstym zmianom.

Sposób korzystania z list bardzo przypomina korzystanie z wektorów. Oto przykład wykorzystania listy:

```
#include <list>          // Nagłówek <list> zawiera deklarację
                         // std::list

using namespace std;

int main()
{
     // Tak wygląda utworzenie pustej listy typu int i
     // nazwanie jej lista:
     list <int> lista;

     // Aby umieścić na końcu listy pojedynczy element,
     // używamy funkcji "push_back():"
     lista.push_back(1);

     // Aby usunąć pojedynczy element z końca listy, używam
     // funkcji "pop_back()":
     lista.pop_back();

     // Funkcja "push_front()" umieszcza pojedynczy element na
     // początku listy:
     lista.push_front(0);     // Teraz pierwszym elementem jest
                              // 0, a nie 1
```

```
    // Podobnie, funkcja "pop_front()" usuwa pierwszy element
    // listy:
    lista.pop_front();
    return 0;
}
```

Iteratory

Iteratory usprawniają pracę związaną z obsługą kontenerów. Poza tym umożliwiają dotarcie do danego składnika pojemnika bez konieczności poznawania jego struktury. Posługiwanie się iteratorem przypomina używanie zwykłych wskaźników. Jest on jednak czymś więcej niż wskaźnik - w przeciwieństwie do wskaźników, korzystając z iteratora nie musimy martwić się czy przypadkiem nie przekroczyliśmy zakresu pojemnika ani czy poprawnie wskazuje on na wybrany element. Tym wszystkim zajmuje się sam iterator, co pozwala programiście skupić się na innych problemach w trakcie tworzenia programu.

Oto przykład wykorzystania iteratorów do wyświetlania elementów listy dwukierunkowej:

```
#include <iostream>
#include <list>

using namespace std;

int main ()
{
  list<int> lista;

  // Inicjujemy listę kolejnymi liczbami naturalnymi:
  lista.push_back(1);
  lista.push_back(2);
  lista.push_back(3);

  // Wyświetlenie składników listy w pętli przy pomocy iteratora przejścia w
  // przód:
  list<int>::iterator it;
  for( it=lista.begin(); it!=lista.end(); ++it )
  {
    cout<< *it <<'\n';
  }

  return 0;
}
```

Metody begin() i end() skonstruowane są do przeglądania kontenera od początku do końca. Jeśli chcemy działać na składowych listy w odwrotnej kolejności korzystamy w tym celu z metody rbegin(), która zwraca odwrócony iterator wskazujący na ostatni element pojemnika (mówi się także, że jest to odwrócony początek). Odwołuje się on do elementu

bezpośrednio poprzedzającego iterator wskazywany przez end. Jest to odwrócony iterator bezpośredniego dostępu. Mamy także metodę rend(), która zwraca odwrócony iterator do elementu odwołującego się do elementu bezpośrednio poprzedzającego pierwszy element kontenera list (zwany także odwróconym końcem). rend() wskazuje miejsce bezpośrednio poprzedzające składnik do którego odwoływałby się begin(). Oto przykład:

```
#include <iostream>
#include <list>

using namespace std;

int main ()
{
  list<int> lista;

  // Inicjujemy listę kolejnymi liczbami naturalnymi:
  lista.push_back(1);
  lista.push_back(2);
  lista.push_back(3);

  // Wyświetlenie składników listy w pętli przy pomocy iteratora przejścia w
  // przód:
  list<int>::reverse_iterator it;
  for( it=lista.rbegin(); it!=lista.rend(); ++it )
  {
    cout<< *it <<'\n';
  }

  return 0;
}
```

Zbiory

Zbiory są jednym z kontenerów biblioteki STL, których struktura oparta jest na drzewach. Elementy które są w nich przechowywane są posortowane, według pewnego klucza. Drzewiasta struktura zapewnia szybkie wyszukiwanie, jednak są z tym związane także pewne mankamenty, mianowicie modyfikacja elementu jest możliwa tylko w taki sposób, że kasujemy stary element, a następnie wstawiamy w to miejsce nowy. Zbiory są tzw. kontenerami asocjacyjnymi (o zmiennej długości, pozwalającymi na operowanie elementami przy użyciu kluczy). Oto przykład wykorzystujący kontener *set*:

```
#include<iostream>
#include<set>

using namespace std;
```

```
int main()
{
  set<int> zbior;
  zbior.insert(4);
  zbior.insert(3);
  zbior.insert(1);
  zbior.insert(2);

  set<int>::iterator it;
  for( it=zbior.begin(); it!=zbior.end(); ++it )
    cout<<*it<<'\n';

  return 0;
}
```

Mapy

Mapy są posortowanymi kontenerami asocjacyjnymi, czyli zbiornikami o zmiennej długości gromadzącymi dane, które można dodawać i usuwać. Nie można jednak dodawać danych na konkretną pozycje, ponieważ kolejność ustalana jest według danego klucza. Mapa jest również parowym zbiornikiem asocjacyjnym, czyli jej elementami są pary wartości klucz i dana. Pierwszej wartości (first), czyli klucza mapy, nie można zmieniać, natomiast druga wartość czyli wartość danej (second) jest możliwa do zmiany. Mapa jest w końcu unikalnym kontenerem asocjacyjnym, co oznacza, że każdy element ma indywidualny klucz. Oto przykład wykorzystujący kontener *map*:

```
#include<iostream>
#include<map>

using namespace std;

int main()
{
  map<int, string> miesiac;
  miesiac[1] = "styczen";
  miesiac[2] = "luty";
  miesiac[3] = "marzec";
  miesiac[4] = "kwiecien";
  miesiac[5] = "maj";
  miesiac[6] = "czerwiec";
  miesiac[7] = "lipiec";
  miesiac[8] = "sierpien";
  miesiac[9] = "wrzesien";
  miesiac[10] = "pazdziernik";
  miesiac[11] = "listopad";
```

```
  miesiac[12] = "grudzien";

  cout << "5 miesiac to:  " << miesiac[5] << '\n';

  map<int, string>::iterator cur;

  // zwrócenie elementu o kluczu 5
  cur = miesiac.find(5);

  // pierwszy sposób dostępu do klucza i danej
  cout<<cur->first<<" miesiąc to: "<<cur->second<<endl;

  // drugi sposób dostępu do klucza i danej
  cout<<(*cur).first<<" miesiąc to: "<<(*cur).second<<endl;

  // elementy o kluczach większych i mniejszych
  map<int, string>::iterator prev = cur;
  map<int, string>::iterator next = cur;
  ++next;
  --prev;

  cout << "Wczesniejszy, czyli "<<(*prev).first<<" element mapy to " <<
(*prev).second << '\n';
   cout << "Nastepny, czyli "<<next->first<< " element mapy to "<<
next->second << '\n';

  return 0;
}
```

Zadania do wykonania

1. Napisać program, który będzie implementował stos przy pomocy kontenera list. W szczególności chodzi o umożliwienie wykonywanie takich operacji jak umieszczanie elementu na stosie, zdejmowanie elementu ze stosu, wyświetlanie zawartości stosu, przy użyciu prostego menu.
2. Napisać program, który będzie implementował listę uczniów w klasie przy pomocy kontenera set. W szczególności chodzi o umożliwienie wykonywanie takich operacji jak umieszczanie ucznia na liście, usuwanie ucznia z listy, wyświetlanie listy uczniów, przy użyciu prostego menu.
3. Napisać program, który będzie implementacją słownika angielsko-polskiego przy pomocy kontenera map. W szczególności chodzi o umożliwienie wykonywanie takich operacji jak umieszczanie słowa angielskiego wraz z odpowiednim tłumaczeniem w słowniku, wyszukanie słowa angielskiego i wyświetlenie odpowiednika polskiego, wyświetlenie zawartości słownika na ekranie, przy użyciu prostego menu.