Biblioteka STL
Kontenery i iteratory

C++/Iteratory

Idea iteratorów opiera się na tym, by ułatwić i usprawnić pracę na kontenerach. Daje możliwość dotarcia do danego składnika pojemnika bez konieczności znajomości jego struktury. Słusznie używanie iteratora przypomina pracę przy pomocy zwykłych wskaźników. Iterator umożliwia wygodny dostęp sekwencyjny do wszystkich elementów kontenera.

Można używać gotowych iteratorów dla kontenerów z STL przez dołączenie biblioteki <iterator>. Istnieją pewne analogie między iteratorem a wskaźnikiem. Przede wszystkim znajomo wyglądają wyrażenia:

```
*nasz_iterator   // wartością jest element pojemnika wskazywany przez iterator
                 // nasz_iterator
wart = *nasz_iter   // podstawienie wartości elementu pod zmienną wart
*nasz_iterator = wart   // podstawienie wartości zmiennej w miejsce pojemnika wskazane
                        // przez nasz_iterator
```

Umożliwia nam to przeciążony operator*, dzięki któremu mamy dostęp do obiektu wskazywanego przez iterator jak również możliwość modyfikowania jego zawartości.

Podział iteratorów

- Iteratory wejścia:

Taki iterator może odczytać wartość elementu, na który wskazuje, o ile nie wskazuje przekroczenia zakresu – end(). Musi posiadać domyślny konstruktor, konstruktor kopiujący oraz operatory =, ==, !=, ++. Odczytać wartość można:

```
wart = *iterator;    // poprawne użycie iteratora wejścia
*iterator = wartosc;    // niepoprawne użycie iteratora wejścia- nie może zapisywać
```

Można inkrementować iterator – aby wskazywał następny składnik:

```
iterator++  lub  ++iterator
```

- Iteratory wyjścia:

Ten typ iteratora umożliwia tylko zapis wartości do danego składnika – odczyt jest niemożliwy. Musi posiadać domyślny konstruktor, konstruktor kopiujący oraz operatory =, ++. np.

```
*iterator = wartosc;  // poprawnie
wartosc = *iterator;  // błędnie - próbujemy odczytać
```

- Iteratory przejścia w przód:

Jest to połączenie operatora wejścia i wyjścia, posiada domyślny konstruktor, konstruktor kopiujący oraz operatory =, ==, != ,++. Możliwy jest zarówno zapis jak i odczyt. Przesuwanie się po kolejnych składnikach tak jak poprzednio jest możliwe tylko w przód poprzez inkrementacje iteratora.

- Iteratory dwukierunkowe:

Różnią się tylko możliwością dekrementacji iteratora od iteratorów przejścia w przód.

- Iteratory bezpośredniego dostępu:

Można powiedzieć, że ten typ z kolei dziedziczy wszystko po iteratorach dwukierunkowych, przy czym posiada możliwość dostępu bezpośrednio do wybranego składnika bez potrzeby skanowania struktury (w najgorszym wypadku całej). Z racji tego, że można przeskoczyć o większą liczbę składników niż jeden, iteratory te posiadają operatory: +, +=, -, -=, []. A także dodatkowo operatory <, >, <=, >=.

Sposób użycia iteratora bezpośredniego dostępu: (załóżmy ze nasz iterator wskazuje już składnik n-ty)

```
iterator += 5;   // teraz wskazuje (n+5)-ąty składnik
iterator++;   // teraz wskazuje (n+6)-ty składnik
*iterator[n] = wartosc;   // przypisujemy n-temu składnikowi wartosc
```

Użycie w poszczególnych kontenerach

Nazwy niektórych metod powtarzają się w różnych pojemnikach a także pełnią te same funkcje.

Poruszanie się za pomocą iteratorów po kolejnych składowych może odbywać się na dwa sposoby:

- w przód - czyli od początku do końca np.:

```
kontener<typ kontenera>::iterator iter;   // iterator do przodu
```

początek struktury wyznacza metoda begin(); zaś koniec end();

- od końca - czyli od końca do początku np.:

```
kontener<typ kontenera>::reverse_iterator iter;   // iterator od końca
```

skanować możemy od rbegin(); do rend();, co można utożsamiać z odwróconym początkiem i odwróconym końcem.

Przed użyciem iteratora należy najpierw nadać mu wartość. Dlatego gdy chcemy powtarzać pewną operacje w pętli dla każdego elementu wektora – od początku do końca – inicjujemy iterator wartością bezparametrowej funkcji begin(). Metoda ta zwraca iterator wskazujący na pierwszy element pojemnika wektor. Jest to iterator bezpośredniego dostępu. Nie mamy jednak możliwości porównywania iteratora z wartością NULL (bo iterator zwykłym wskaźnikiem nie jest) musi istnieć metoda która pokazuje koniec naszego wektora. Tak też jest – metoda end() zwraca iterator za ostatnim elementem. To także jest iterator bezpośredniego dostępu.

Przykład:

```
#include <iostream>
#include <vector>
using namespace std;
int main ()
{
   vector<int> tab;

   // inicjujemy wektor kolejnymi liczbami naturalnymi
   tab.push_back(1);
   tab.push_back(2);
   tab.push_back(3);

   // wyswietlenie skladnikow wektora tab w petli przy pomocy iteratora
   vector<int>::iterator it;
   for( it=tab.begin(); it!=tab.end(); ++it )
```

```
    {
      cout<< *it <<'\n';
    }
    return 0;
}
```

Program spowoduje wyświetlenie kolejnych elementów pojemnika:

```
1
2
3
```

Metody begin() i end() skonstruowane są do przeglądania wektora od początku do końca. Co jeśli chcemy działać na składowych wektora w odwrotnej kolejności? Nie ma problemu. Istnieje bowiem metoda 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 wector (zwany także odwróconym końcem). rend() wskazuje miejsce bezpośrednio poprzedzające składnik do którego odwoływałby się begin().

Przykład:

```
#include <iostream>
#include <vector>
using namespace std;
int main ()
{
   vector<int> tab;
   // inicjujemy wektor kolejnymi liczbami naturalnymi
   tab.push_back(1);
   tab.push_back(2);
   tab.push_back(3);

   // wyświetlenie skladników wektora tab w pętli przy pomocy odwróconego iteratora
   vector<int>::reverse_iterator it;
   for( it=tab.rbegin(); it!=tab.rend(); ++it )
   {
      cout<<*it<<'\n';
   }
   return 0;
}
```

Wykonanie programu spowoduje wyświetlenie składników kontenera w odwrotnej kolejności:

```
3
2
1
```

Należy jeszcze podkreślić, że przy użyciu odwróconego iteratora, by przejrzeć wszystkie elementy pojemnika nie używamy a ++! Rzeczywiście, chcemy przejrzeć wektor od końca do początku i właśnie ku temu służy odwrócony iterator – zdefiniowany w nim porządek strukturalny jest odwrotny do porządku w zwykłym iteratorze.

Powyższe przykłady pokazują jak dużym ułatwieniem dla korzystania z iteratorów są przeładowane operatory inkrementacji i dekrementacji: operator++ i operator--, których używamy tu jak na zwykłej liczbie int. Mamy tu także operator= podstawienia, jak i porównanie iteratorów - operator!=.

Uwagi

Używanie iteratorów w pętli for skutkuje dość długim (i w opinii niektórych - mało czytelnym) kodem. Czasami, można zwiększyć cztelność kodu stosując makro FOREACH:

```
#define VAR(v,n) decltype(n) v=(n)
#define FOREACH(i,c) for(VAR(i,(c).begin());i!=(c).end();++i)
```

Użycie w kodzie:

```
vector<int> v;
v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(4);
FOREACH(it, v)
{
    cout << *it << endl;
}
```

W C++11 wprowadzono pętlę for opartą na zakresie (ang. range-based for loop), dzięki czemu powyższy kod można uprościć do:

```
vector<int> v;
v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(4);
for(auto &x : v)
{
    cout << x << endl;
}
```

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.