STRUKTURA DANYCH - LISTA

Podstawową zaletą *tablic* jest to, iż mamy *szybki dostęp* do każdego z elementów. Elementy tablicy są umieszczane jeden obok drugiego w *spójnym bloku pamięci*. Tablice posiadają również wady:

- Jeśli chcemy wstawić jakiś element do tablicy, to musimy przestawiać inne elementy tablicy, aby zrobić w niej miejsce na nowy element. To kosztuje i spowalnia działanie algorytmów, które często muszą coś wstawiać lub usuwać ze zbioru danych.
- Drugą wadą tablic jest to, iż wymagają one ciągłego obszaru pamięci na wszystkie dane. Jeśli komputer nie ma pod ręką tak dużego obszaru, to tablica nie zostanie utworzona - chociaż pamięć może być dostępna, lecz w kilku mniejszych kawałkach.

Aby rozwiązać te problemy, wymyślono *dynamiczną strukturę danych*, zwaną **listą** (ang. *list*). Lista jest ciągiem powiązanych ze sobą elementów. Z danego elementu listy można przejść do elementu następnego lub do elementu poprzedniego. Każdy element listy posiada następującą strukturę:

```
struct node{                    //węzeł listy dwukierunkowej
    node * next;                // wskaźnik elementu następnego
    node * prev;                // wskaźnik elementu poprzedzającego
    typ  data;                  // dowolne dane, które są przechowywane w
                                // elemencie listy
};
```

Lista w STL

```
#include<list>
list<typ> nazwa;
    np.: list<int> liczby;      //stworzenie pustej listy
list<typ>::iterator nazwa;
    np. list<int>::iterator wsk = liczby.begin(); //deklaracja iteratora
```

Operacje na listach:

`int size()`	zwraca aktualną liczbę elementów listy;
`bool empty()`	zwraca *true* jeśli lista jest pusta;
`front()`	zwraca *referencję* na pierwszy element listy;
`back()`	zwraca *referencję* na ostatni element listy;
`push_back(typ obj)`	dołącza *kopię* elementu *obj* na końcu listy;
`push_front(typ obj)`	dołącza *kopię* elementu *obj* na początku listy;
`pop_back()`	usuwa ostatni element listy, nic nie zwraca;
`pop_front()`	usuwa pierwszy element listy, nic nie zwraca;

`iterator insert(iterator pos, typ obj)` – wstawia element *obj* przed wskazywaną przez iterator *pos* pozycją i zwraca *iterator* na dostawiony element;

`iterator erase(iterator pos)` – usuwa element wskazywany przez *pos* i zwraca *iterator* na następny element;

`clear()` usuwa wszystkie elementy z listy;

`begin()` zwraca *iterator* na pierwszy element listy;

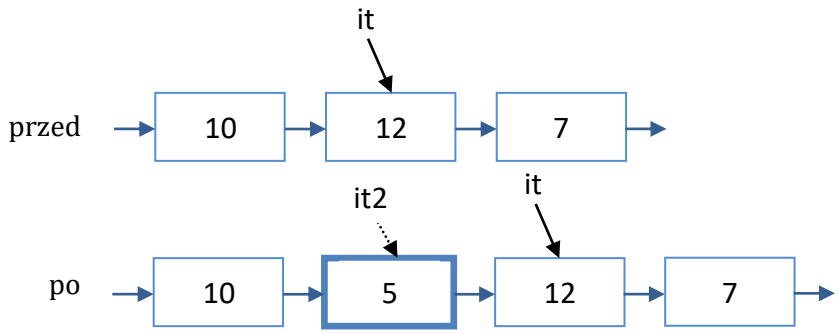
`end()` zwraca *iterator* na element występujący bezpośrednio za ostatnim elementem listy;

`iterator rbegin()` zwraca odwrócony *iterator* na „pierwszy” (od końca) element;

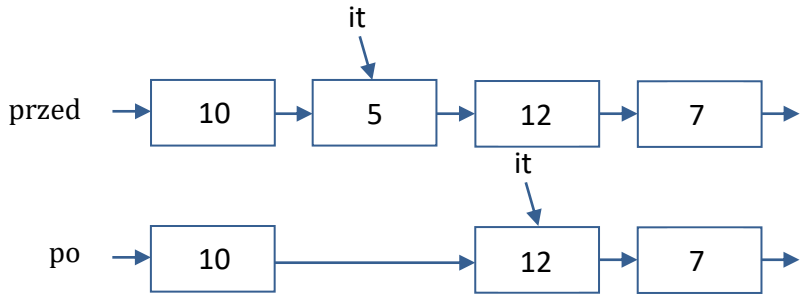
`iterator rend()` zwraca odwrócony *iterator* na element występujący bezpośrednio przed pierwszym elementem listy.

Przykład. Implementacja listy w STL, zobacz projekt *zadania_lista*.

it2 = ***insert***(it, 5)



it = ***erase***(it)



Zadanie 1. Napisz funkcję wyszukującą na liście podaną wartość.

Zadanie 2. Napisz funkcję usuwającą *i*-ty węzeł z listy (węzły numerujemy od 1).

Zadanie 3. Napisz funkcję scalającą (*merge*) dwie uporządkowane listy w jedną uporządkowana listę.

Zadanie 4. Napisz funkcję zapisującą do listy liczby w taki sposób, aby w każdym momencie działania programu lista była posortowana.