Вступ до ітераторів у Cі++

Ітератор - це об'єкт (як вказівник), який вказує на елемент всередині контейнера. Ми можемо використовувати ітератори для переміщення по вмісту контейнера. Їх можна уявити як щось подібне до вказівника, що вказує на певне місце, і ми можемо отримати доступ до вмісту з цього конкретного місця.

Ітератори забезпечують доступ до елементів контейнера. За допомогою ітераторів дуже зручно перебирати елементи. Итератор описується типом iterator. Але для кожного контейнера конкретний тип ітератора буде відрізнятися. Так, итератор для контейнера list <int> представляє тип list <int> :: iterator, а итератор контейнера vector <int> представляє тип vector <int> :: iterator і так далі. Для отримання ітераторів контейнери в Cі++ мають такі методи, як begin () і end (). Функція begin () повертає ітератор, який вказує на перший елемент контейнера (при наявності в контейнері елементів). Функція end () повертає ітератор, який вказує на наступну позицію після останнього елемента, тобто по суті на кінець контейнера. Якщо контейнер порожній, то ітератори, які повертаються обома методами begin і end збігаються. Якщо ітератор begin НЕ дорівнює Ітератору end, то між ними є як мінімум один елемент. Обидві ці функції повертають ітератор для конкретного типу контейнеру:

int ar[]= { 1,2,3,4 }

std::vector<int> v(ar,ar+4) ;

std::vector<int>::iterator iter = v.begin(); // отримуємо ітератор

В даному випадку створюється вектор - контейнер типу vector, який містить значення типу int. І цей контейнер ініціалізуєтся числами {1, 2, 3, 4}. І через метод begin () можна отримати ітератор для цього контейнера. Причому цей ітератор буде вказувати на перший елемент контейнеру.

Операції з ітераторами

З ітераторами можна проводити наступні операції:

\* Iter: отримання елемента, на який вказує ітератор. Якщо цей елемент має члени або методи, то за допомогою оператору → (або через оператори зірочка та крапка) можна отримати доступ до них безпосередньо з ітератора.

++ iter: переміщення ітератора вперед для звернення до наступного елементу

--iter: переміщення ітератора назад для звернення до попереднього елемента. (Ітератори контейнера forward\_list не підтримують операцію декремента).

iter1 == iter2: два ітератора рівні, якщо вони вказують на один і той же елемент

iter1! = iter2: два ітератора не рівні, якщо вони вказують на різні елементи.

Оператор присвоєння = присвоює ітератор (позицію елемента, на яку він посилається).

Наприклад, використовуємо ітератори для перебору елементів вектора:

vector<int>::iterator iter = v.begin(); // отримуємо ітератор

while(iter!=v.end()) // поки не досягнемо кінця вектору

{

std::cout << \*iter << std::endl;// виводимо результат як значення вказівника

++iter; // рухаємося по вектору інкрементуючи ітератор

}

Ітератори відіграють важливу роль у підключенні алгоритму з контейнерами разом з маніпуляціями даними, що зберігаються всередині контейнерів. Найбільш очевидною формою ітератора є вказівник. Вказівник може вказувати на елементи в масиві і може перебирати їх за допомогою оператору інкременту (++). Але не всі ітератори мають всю функціональність вказівників.  
Залежно від функціональності ітераторів вони можуть бути класифіковані на п'ять категорій, як показано на діаграмі нижче, причому зовнішнє є найпотужнішим, а отже, внутрішнє є найменш потужним з точки зору функціональності.

**Довільний доступ (Random-Access)**

**Двосторнній (BIDERECTIONAL)**

**Односторнній(FORWARD)**

**Виведення**

**(OUTPUT)**

**Введення**

**(Input)**

Не кожен з цих ітераторів підтримується всіма контейнерами в STL, різні контейнери підтримують різні ітератори, наприклад, вектори підтримують ітератори довільного доступу, в той час як списки підтримують двонаправлені ітератори. Весь список наведено нижче:

|  |  |
| --- | --- |
| Контейнер | Тип підтримуваних ітераторів |
| вектор | Довільного доступу |
| список | Двонаправлений |
| дек | Довільного доступу |
| масив | Довільного доступу(?) |
| Однонаправлений список | Однонаправлений ітератор |
| відображення | Двонаправлений |
| мультивідображення | Двонаправлений |
| множина | Двонаправлений |
| мультимножина | Двонаправлений |
| стек | Немає ітераторів |
| черга | Немає ітераторів |
| черга з пріорітетом | Немає ітераторів |

Таким чином, грунтуючись на функціональності ітераторів, вони можуть бути розділені на п'ять основних категорій:  
1. Ітератори вводу**(Inputiterators**): Вони є найслабкішими з усіх ітераторів і мають дуже обмежену функціональність. Вони можуть використовуватися тільки в алгоритмах з одним проходом, тобто в тих алгоритмах, які обробляють контейнер послідовно, так що жоден елемент не доступний більш ніж один раз.  
2. Ітератори виводу([**Output Iterators**](https://www.geeksforgeeks.org/output-iterators-cpp/)): Так само, як і ітератори вводу, вони також дуже обмежені у своїй функціональності і можуть бути використані тільки в алгоритмі з одним проходом, але не для доступу до елементів, а для визначення(зміни) елементів.  
3. Однонаправлений ітератор ([**ForwardIterator**](https://www.geeksforgeeks.org/forward-iterators-in-cpp/)): Вони мають вищу ієрарахію, ніж вхідні і вихідні ітератори, і містять всі функції, присутні в цих двох ітераторах. Але, як випливає з назви, вони також можуть рухатися лише у прямому напрямку, і це теж один крок за один раз.  
4. Двонаправлені ітератори([**Bidirectional Iterators**](https://www.geeksforgeeks.org/bidirectional-iterators-in-cpp/)): Вони мають всі особливості форвард ітераторів разом з але вони можуть рухатися в обох напрямках, тому їх звуть двонаправленими.  
5. Ітератори прямого доступу([**Random-AccessIterators**](https://www.geeksforgeeks.org/random-access-iterators-in-cpp/)): Вони є найпотужнішими ітераторами. Вони не обмежуються переміщенням послідовно, як свідчить їх назва, вони можуть безпосередньо звертатися до будь-якого елемента всередині контейнера. Їх функціональні можливості такі ж, як і у вказівників.  
Наступна діаграма показує різницю в їх функціональності щодо різних операцій, які вони можуть виконувати.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тип Ітератору | Доступ | Читання | Запис | Рух | Порівняння |
| Введення | -> | = \*i |  | ++ | ==, != |
| Виведення |  |  | \*i= | ++ |  |
| Форвард | -> | = \*i | \*i= | ++ | ==, != |
| Двонаправлений |  | = \*i | \*i= | ++, -- | ==, != |
| Прямого доступу | ->, [] | = \*i | \*i= | ++,--, +=,  -=, +, - | ==, !=, <, >,>=, <= |

Можна побачити, що оператори які використвуються для ітереторів мають інтерфейс, що точно збігається з інтерфейсом звичайних вказівників у мовах Cі та Cі++, за допомогою яких можна обійти елементи і в звичайному масиві.

Різниця полягає в тому, що ітератор є інтелектуальним вказівником, тобто може обходити більш складні структури даних. Внутрішня поведінка ітераторів залежить від структури даних, по якій вони переміщаються. З цієї причини кожен контейнерний тип передбачає свій власний вид ітераторів. У результаті ітератори мають загальний інтерфейс, але різні типи. Це безпосередньо приводить до концепції узагальненого програмування: *операції використовують однаковий інтерфейс, але мають різні типи*, тому можна використовувати шаблони для формулювання узагальнених операцій, що застосовуються до довільних типів, що задовольняють зазначеному інтерфейсу.

Методи контенерів для роботи з ітераторами

Усі контейнерні класи (послідовні контейнери та асоціативні контейнери) мають однакові основні функції-члени, що дозволяють переміщати ітератори по елементах контейнера.

Методи роботи з ітераторами:

* [begin()](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/) – повертає ітератор на перший обєкт вектору
* [end()](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/) – повертає ітератор на останній обєкт вектору

Починаючі зі стандарту С++11 додано ще наступні варіанти доступу до ітераторів:

* [rbegin()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/) – повертає ітератор на останній обєкт вектору як на початковий (reverse beginning). Рухається з останнього елементу до першого;
* [rend()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/) – повертає ітератор на перший обєкт вектору як на останній (reverse beginning). Рухається з останнього елементу до першого;
* [cbegin()](https://www.geeksforgeeks.org/vector-cbegin-vector-cend-c-stl/) – повертає константний ітератор на преший елемент;
* [cend()](https://www.geeksforgeeks.org/vector-cbegin-vector-cend-c-stl/) – повертає константний ітератор на останній елемент;
* [crbegin()](https://www.geeksforgeeks.org/vectorcrend-vectorcrbegin-examples/) – повератає константний реверсивний оператор на початок;
* [crend()](https://www.geeksforgeeks.org/vectorcrend-vectorcrbegin-examples/) – повератає константний реверсивний оператор на кінець вектору.

Функція begin()повертає ітератор, що представляє початок контейнера, тобто позицію першого елемента, якщо такий мається в контейнері.

Функція end() повертає ітератор, що представляє кінець контейнера, тобто позицію, що слідує за останнім елементом.

Такий ітератор називається позамежним (past-the-end iterator).

Таким чином, функції – члени begin()і end() визначають напіввідкритий діапазон (half-open range), що включає перший елемент і не включає останній.

Напіввідкритий діапазон має дві переваги.

1. Існує простий критерій зупинки циклу при обході всіх елементів: Цикл продовжується, поки не буде досягнута позиція end().

2. Він дозволяє уникнути спеціальної обробки порожніх діапазонів. Для порожніх діапазонів позиція begin() збігається з позицією end(). Наступний приклад демонструє використання ітераторів для виводу на екран всіх елементів списку(це варіант попереднього прикладу, але з використанням ітераторів):

// iterator, begin() and end()

#include<iostream>

#include<iterator> // for iterators

#include<vector> // for vectors

usingnamespacestd;

intmain()

{

vector<int> ar = { 1, 2, 3, 4, 5 };

// Declaring iterator to a vector

vector<int>::iterator ptr;

// Displaying vector elements using begin() and end()

cout << "The vector elements are : ";

for(ptr = ar.begin(); ptr < ar.end(); ptr++)

cout << \*ptr << " ";

}

З Сі++11 для отримання константного ітератору const\_iterator можна використовувати та методи отримання cbegin() та cend():

std::vector<int> v { 1, 2, 3, 4, 5 };

//std::vector<int>::const\_iterator iter;

for ( auto iter = v.cbegin(); iter != v.cend(); ++iter){

std::cout << \*iter << " ";

// це неможливо бо ітератор константний

//\*iter = (\*iter) \* (\*iter);

}

std::cout << std::endl;

Результат:

1 2 3 4 5

Реверсивні ітератори (Сі++11)

Реверсивні ітератори дозволяють перебирати елементи контейнеру в зворотному напрямку. Для отримання реверсівного ітератору використовують методи контейнерів rbegin() та rend(), а сам ітератор утворює тип reverse\_iterator.

Так саме для реверсівногоітератору існує констаннта форма const\_reverse\_iterator яку можна отримати за допомогою методів crbegin() и crend():

std::vector<int> v { 1, 2, 3, 4, 5 };

for (std::vector<int>::reverse\_iterator iter = v.rbegin(); iter != v.rend(); ++iter){

std::cout << \*iter << " ";

\*iter = (\*iter) \* (\*iter);

}

std::cout << "\n";

for (std::vector<int>::const\_reverse\_iterator iter = v.crbegin(); iter != v.crend(); ++iter)

{

std::cout << \*iter << " ";

// неможливо бо ітератор константний

//\*iter = (\*iter) \* (\*iter);

}

Результат:

5 4 3 2 1

25 16 9 4 1

Функції роботи з ітераторами бібліотеки **iterator**

В бібліотеці <**iterator> визначені деякі додаткові до стандартних функцій корисні функції для маніпулювання з ітераторами.**

**Серед них можна виділити наступні:**

**advance(iter, n) - інкрементує ітератор iter на визначену кількість позицій n.**

int mas[] ={1,2,3,4,5, 10,20,30};

std::vector<int> ar(mas,mas+5);

std::vector<int>::iterator ptr = ar.begin();

std::advance(ptr, 3);

//ptr += 3; // буде виконувати те саме

std::cout << "Third element is : "<<\*ptr;

std::list<int> list1(mas,mas+5);

std::list<int> list2(mas+6,mas+8);

std::list<int>::iterator ptr2 = list1.begin();

std::advance(ptr2, 3);

//ptr += 3; // А ось для списку цей варіант некоректний

std::cout << "Third element is : "<<\*ptr2;

**6. inserter(cont, iter) - функція для вставки елементів в будь-яку позицію контейнеру. Аргументи — контейнер та ітератор на позицію куди треба вставляти елементи. Функція повертає insert\_iterator який дозволяє вставити елементи в інший контейнер.**

#include<vector> // for vectors

#include<deque> // for deque

#include<list> // for list

// друк елементів будь-якої колекції

template<class Coll> void printCollect(Coll & v){

typename Coll::const\_iterator it = v.begin();

for(;it!=v.end();++it){

std::cout<<\*it<<" ";

}

std::cout<<"\n";

}

int main() {

int mas[] ={1,2,3,4,5, 10,20,30};

std::vector<int> resulting(mas,mas+5); // куди вставляємо

std::vector<int> to\_insert(mas+6,mas+8); // що вставляємо

std::vector<int>::iterator ptr = resulting.begin();

std::advance(ptr, 3); // на 3 позицію ітератор

//ptr += 3; // для вектору це те саме що і попоредній рядок

std::cout << "Third element is : "<<\*ptr<<"\n";

std::insert\_iterator<std::vector<int> > ari = inserter(resulting, ptr); // куди вставляти

//std::insert\_iterator<std::vector<int> > ari(resulting, ptr);

// копіюємо вміст to\_insert(без 1-го ел-ту) в resulting

std::copy(to\_insert.begin()+1, to\_insert.end(), ari);

std::cout << "The new vector after inserting elements is : ";

printCollect(resulting);

// те саме для списку

std::list<int> list1(mas,mas+5);

std::list<int>::iterator ptr2 = list1.begin();

std::advance(ptr2, 3);

//ptr += 3; // а ось для списку це некоректно

std::cout << "Third element is : "<<\*ptr2<<"\n";

std::copy(to\_insert .begin(), to\_insert.end(), inserter(list1,ptr2));

std::cout << "The new list after inserting elements is : ";

printCollect(list1);

}

Результат:

Third element is : 4

The new vector after inserting elements is : 1 2 3 30 4 5

Third element is : 4

The new list after inserting elements is : 1 2 3 20 30 4 5

Аналогічно до функції **inserter визначені функції**

**back\_inserter(), front\_inserter() які вставляють відповідно в початок та кінець колекції.**

std::deque<int> foo,bar;

for (int i=1; i<=5; i++){ foo.push\_back(i); bar.push\_back(i\*10); }

std::copy (bar.begin(),bar.end(),std::front\_inserter(foo));

std::cout << "foo after front insert contains:";

/\*for ( std::deque<int>::iterator it = foo.begin(); it!= foo.end(); ++it )

std::cout << ' ' << \*it;

std::cout << '\n';\*/

printCollect(foo);

std::vector<int> foo1,bar1;

for (int i=1; i<=5; i++)

{ foo1.push\_back(i); bar1.push\_back(i\*10); }

std::copy (bar1.begin(),bar1.end(),back\_inserter(foo1));

std::cout << "foo after back insert contains:";

/\* for ( std::vector<int>::iterator it = foo.begin(); it!= foo.end(); ++it )

std::cout << ' ' << \*it;

std::cout << '\n';

\*/

printCollect(foo1);

Результат:

foo after front insert contains:50 40 30 20 10 1 2 3 4 5

foo after back insert contains:1 2 3 4 5 10 20 30 40 50

Також можна вказати функцію distance, що визнчає відстань між двома ітерторами:

std::vector<int> v(mas+1,mas+7);

std::cout << "distance(first, last) = "

<< std::distance(v.begin(), v.end()) << '\n'

<< "distance(last, first) = "

<< std::distance(v.end(), v.begin()) << '\n';

//the behavior is undefined (until C++11)

Результат:

distance(first, last) = 6

distance(last, first) = -6

З С++11 додано також наступні функції:

begin(coll) — повертає ітератор на початок колекції;

end(coll) - повертає ітератор на кінець колекції;

**next(iter, n) - функція повертає новий ітератор, що вказує на позицію яка слідує за тією що стала після застосування n разів інкременту для iter.**

**prev(iter, n) - функція повертає новий ітератор, що вказує на позицію яка передує тієй, що стала після застосування n разів декременту для iter.**

#include<iostream>

#include<iterator> // for iterators

#include<vector> // for vectors

using namespace std;

template <class Coll> void printColl(const Coll & v){

for(const auto & x: v){

cout<< x <<" ";

}

cout<<endl;

}

int main(){

vector<int> ar = { 1, 2, 3, 4, 5 };

// Declaring iterators to a vector

vector<int>::iterator ptr = begin(ar);

vector<int>::iterator ftr = end(ar);

// Using next() to return new iterator

auto it = next(ptr, 3); // points to 4

// Using prev() to return new iterator

auto it1 = prev(ftr, 3); // points to 3

// Displaying iterator position

cout << "The position of new iterator using next() is : ";

cout << \*it << " " << endl;

// Displaying iterator position

cout << "The position of new iterator using prev()  is : ";

cout << \*it1 << " " << endl;

int foo[] = {10,20,30,40,50, 60};

std::vector<int> bar;

// iterate foo: inserting into bar

// be carefull - this fails if size of collection is odd!!!!!

for (auto it = std::begin(foo); it!=std::end(foo); advance(it,2))

bar.push\_back(\*it);

// iterate bar: print contents:

std::cout << "bar contains:";

/\*for (auto it = std::begin(bar); it!=std::end(bar); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

\*/

printColl(bar);

}

Результат:

The position of new iterator using next() is : 4

The position of new iterator using prev()  is : 3

bar contains:10 30 50

Створення власного ітератору

Оскільки ми знаймо методи ітераторів, то неважко власноруч створити власний ітератор.

#include <iostream>

using namespace std;

#define MAXSIZE 100

// власний шаблон-огортка над масивом

template<class Type>class MyMasiv{

Type mas[MAXSIZE];

int size;

public:

// створимо власний ітератор для цього класу

class iterator{

Type \*current;

public:

// перевантажимо оператори

iterator() { current = 0; }

void operator++() {

current++;

}

void operator+=(int temp) {

current += temp;

}

void operator-=(int temp) {

current -= temp;

}

void operator=(Type& temp) {

current = &temp;

}

bool operator!=(Type& temp) {

return current != &temp;

}

bool operator==(Type& temp) {

return current == &temp;

}

// будемо виводити всі числа поділени навпіл

Type operator \*() {

return \*current / 2;

}

Type\* operator ->() {

return current; }

};

// методи нашого масиву

MyMasiv(){

size = 0;

}

void add(int temp){

size++;

mas[size - 1] = temp;

}

void del(){

size--;

mas[size + 1] = 0;

}

void show(){

cout << "Массив:\n";

for (int i = 0;i < size;i++)

cout << mas[i] << ' ';

cout << endl;

}

// методи для отримання ітераторів

Type& begin() { return mas[0]; }

Type& end() { return mas[size]; }

};

int main(){

MyMasiv<int> a;

for (int i = 0;i < 5;i++){

a.add(i\*2);

}

MyMasiv<int>::iterator it;

for (it = a.begin();it!=a.end();++it){

cout << \*it << ' ';

}

}

Результат:

0 1 2 3 4

Асоціативні контейнери

Асоціативні контейнери — це контейнери які зберігають відсортовані дані використовуючі їх хеш-значення. Такі конейнери містять Сі++ реалізацію зокрема таких колекцій як Hashtable(Java) та Dictionary(Python), тобто це структури які оптимізовані під пошук у великій кількость даних та там де неважливий порядок введення даних, але важливе їх значення та порядок у якому вони будуть зберігатись.

Множина Set (STL)

Множина (Set) - це тип асоціативних контейнерів, в яких кожен елемент повинен бути унікальним, оскільки значення елемента його ідентифікує. Значення елемента не може бути змінено після його додавання до набору, хоча можна видалити та додати змінене значення цього елемента. Методи, які повязані з Set можуть бути розбити на наступні групи:

* Ідентифікатори та модифікатори обсягу ()
* Модифікатори даних
* Методи пошуку
* Робота з ітераторами

До Сі++11 шаблон класу множина мав наступні конструктори

1) конструктор за замовченням (default constructor), що створює порожню множину

2) Копіконструктор — копіює вміст іншого деку

3) Конструктор за інтервалом (range constructor) — коструює множину за двома ітераторами іншої колекції.

Ідентифікатори та модифікатори обсягу (Capacity)

size() – кількість елементів колекції;

max\_size() – максимально можлива кількість елементів колекції;

empty() – якщо порожня колекція повертає true, інакше false.

Модифікатори даних (Modification):

insert(const g) – додає новий елемент ‘g’ до множини;

insert (iterator position, const g) – додає елемент ‘g’ в позицію вказану ітертором;

erase(iterator position) – видаляє елемент з позиції вказаної ітертором;

erase(const g)– видаляє значення ‘g’ з множини;

clear() – видаляє всі елементи колекції.

Починаючи з C++11 додали також настпні методи:

emplace()– вставляє елемент в множину, якщо цього елементу ще в неї міститься;

emplace\_hint()– повертає ітертор на елемент,я кий вставляється в множину, якщо цього елементу до цього там не було;

swap()– обмінює вміст двох множин ода в іншу.

Методи пошуку (Searching) та спостереження (Observation):

key\_comp() / value\_comp() – повертає обєкт, що вказує як елементи множини впорядкована (‘<‘ за замовченням).

find(const g) – повертає ітератор на елемент ‘g’ якщо він є у множині, або ітератр на кінець множини, якщо його немає.

count(const g) – повертає 1 або 0 в залежності від того є ‘g’ в множині або немає.

lower\_bound(const g) – повертає ітератор на перший елемент, що еквіалентний ‘g’ або на перший елемент, що точно більший за ‘g’;

upper\_bound(const g) – повертає ітератор на перший елемент, що еквіалентний ‘g’ або на останній елемент, що точно менший за ‘g’;

equal\_range((const g) – повертає ітератор на тип пара (key\_comp) в якому перше поле pair::first яквівалентне lower bound, а pair::second еквівалентне upper bound.

Робота з ітераторами (iterators) — аналогічно до послідовних контейнерів:

* [begin()](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/) – повертає ітератор на перший обєкт колекції;
* [end()](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/) – повертає ітератор на останній обєкт колекції;

Починаючі зі стандарту С++11 додано ще наступні варіанти доступу до ітераторів:

* [rbegin()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/) – повертає ітератор на останній обєкт вектору як на початковий (reverse beginning). Рухається з останнього елементу до першого;
* [rend()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/) – повертає ітератор на перший обєкт вектору як на останній (reverse beginning). Рухається з останнього елементу до першого;
* [cbegin()](https://www.geeksforgeeks.org/vector-cbegin-vector-cend-c-stl/) – повертає константний ітератор на преший елемент;
* [cend()](https://www.geeksforgeeks.org/vector-cbegin-vector-cend-c-stl/) – повертає константний ітератор на останній елемент;
* [crbegin()](https://www.geeksforgeeks.org/vectorcrend-vectorcrbegin-examples/) – повертає константний реверсивний оператор на початок;
* [crend()](https://www.geeksforgeeks.org/vectorcrend-vectorcrbegin-examples/) – повертає константний реверсивний оператор на кінець вектору.

Крім того, для множини (як і для інших колекцій) перевизначений оператор присвоєння (operator=).

#include <iostream>

#include <set>

#include <iterator>

using namespace std;

// function comarator — порядок визначається квадратом числа

bool fncomp (int lhs, int rhs) {return lhs\*lhs<rhs\*rhs;}

//class Comparator — порядок по спаданню, а не зростанню

struct Classcomp {

bool operator() (const int& lhs, const int& rhs) const

{return lhs>rhs;}

};

// для множини з одним параметром конструктора

template<typename T>

void printSet(const set<T> v){

typename set<T>::iterator itr;

for(itr=v.begin(); itr!=v.end();++itr){

cout <<" "<<\*itr;

}

cout<< endl;

}

// для множини з двома параметрами конструктору

template<typename T, class U>

void printSet(const set<T,U> v){

typename set<T,U>::iterator itr;

for(itr=v.begin(); itr!=v.end();++itr){

cout <<" "<<\*itr;

}

cout<< endl;

}

int main (){

set<int> first\_set; // empty set of ints

int myints[]= {10,-20,30,40,50,50,40};

set<int> second\_set(myints,myints+7); // range

set<int> third\_set (second\_set); // a copy of second

set<int> fourth\_set (third\_set.begin(), third\_set.end()); // iterator ctor.

cout<<"2,3,4 set:";

printSet(fourth\_set);

cout<<"set size:"<< second\_set.size()<<endl;

cout<<"set max size:"<< second\_set.max\_size()<<endl;

cout<<"how many of 20 there"<< second\_set.count(20)<< " of " << \*second\_set.find(20)<<endl;

cout<<"how many of 50 there"<< second\_set.count(50)<<" of " << \*second\_set.find(50)<<endl;

cout<<"equal range for 20: " << \*second\_set.equal\_range(20).first << ", " << \*second\_set.equal\_range(20).second<<endl;

set<int,Classcomp> fifth\_set; // class as Compare

bool(\*fn\_pt)(int,int) = fncomp;

set<int,bool(\*)(int,int)> sixth\_set (fn\_pt); // function pointer as Compare

set <int,greater <int> > seventh\_set;

//insert elements in random order

seventh\_set.insert(40);

seventh\_set.insert(30);

seventh\_set.insert(60);

seventh\_set.insert(-20);

seventh\_set.insert(50);

seventh\_set.insert(30); // only one 30 will be added to the set

seventh\_set.insert(10);

for (set<int,greater <int> >::iterator it = seventh\_set.begin(); it!=seventh\_set.end();++it){

first\_set.insert(\*it);

fifth\_set.insert(\*it);

sixth\_set.insert(\*it);

}

cout<<"1 and 7th set:";

printSet(first\_set);

cout<<"5th set:";

printSet(fifth\_set);

cout<<"6th set:";

printSet(sixth\_set);

// видалити всі елменти до 30 в second\_set

cout << "second\_set after removal of elements less than 30:";

second\_set.erase(second\_set.begin(), second\_set.find(30));

printSet(second\_set);

// видалити елемент 50 в third\_set

int num;

num = third\_set.erase (50);

cout << "third\_set.erase(50) : ";

cout << num << " removed " ;

printSet(third\_set);

//lower bound та upper bound для seventh\_set

cout << "seventh\_set.lower\_bound(40) : "<< \*seventh\_set.lower\_bound(40) << endl;

cout << "seventh\_set.upper\_bound(40) : "<< \*seventh\_set.upper\_bound(40) << endl;

// lower bound та upper bound для second\_set

cout << "second\_set.lower\_bound(40) : "<< \*second\_set.lower\_bound(40) << endl;

cout << "second\_set.upper\_bound(40) : "<< \*second\_set.upper\_bound(40) << endl;

}

Результат:

2,3,4 set: -20 10 30 40 50

set size:5

set max size:461168601842738790

how many of 20 there 0 of 5

how many of 50 there 1 of 50

equal range for 20: 30, 30

1 and 7th set: -20 10 30 40 50 60

5th set: 60 50 40 30 10 -20

6th set: 10 -20 30 40 50 60

second\_set after removal of elements less than 30: 30 40 50

third\_set.erase(50) : 1 removed -20 10 30 40

seventh\_set.lower\_bound(40) : 40

seventh\_set.upper\_bound(40) : 30

second\_set.lower\_bound(40) : 40

second\_set.upper\_bound(40) : 50

Мультимножини Multiset (STL)

Мультимножини - це тип асоціативних контейнерів, який схожий на множини, але відмінність полягає в тому, що елементи можуть повторюватись (тобто не обвязкова унікальність елементів).

Методи для роботи з мультмножиною класифікуються так само як і для множини:

* Ідентифікатори та модифікатори обсягу ()
* Модифікатори даних
* Методи пошуку
* Робота з ітераторами

До Сі++11 шаблон класу множина мав наступні конструктори

1) конструктор за замовченням (default constructor), що створює порожню множину

2) Копіконструктор — копіює вміст іншого деку

3) Конструктор за інтервалом (range constructor) — коструює множину за двома ітераторами іншої колекції.

Ідентифікатори та модифікатори обсягу (Capacity):

size() – кількість елементів колекції;

max\_size() – максимально можлива кількість елементів колекції;

empty() – якщо порожня колекція повертає true, інакше false.

Модифікатори даних (Modification):

insert(const g) – додає новий елемент ‘g’ до множини;

insert (iterator position, const g) – додає елемент ‘g’ в позицію вказану ітертором;

erase(iterator position) – видаляє елемент з позиції вказаної ітертором;

erase(const g)– видаляє значення ‘g’ з множини;

clear() – видаляє всі елементи колекції.

Починаючи з C++11 додали також настпні методи:

emplace()– вставляє елемент в множину, якщо цього елементу ще в неї міститься;

emplace\_hint()– повертає ітертор на елемент, який вставляється в множину, якщо цього елементу до цього там не було;

swap()– обмінює вміст двох множин ода в іншу.

Методи пошуку (Searching):

key\_comp() / value\_comp() – повертає обєкт, що вказує як елементи множини впорядкована (‘<‘ за замовченням);

find(const g) – повертає ітератор на елемент ‘g’ якщо він є у множині, або ітератр на кінець множини, якщо його немає;

count(const g) – повертає кількість входжень елементу ‘g’ в мультимножину;

lower\_bound(const g) – повертає ітератор на перший елемент, що еквіалентний ‘g’ або на перший елемент, що точно більший за ‘g’;

upper\_bound(const g) – повертає ітератор на перший елемент, що еквіалентний ‘g’ або на останній елемент, що точно менший за ‘g’;

equal\_range((const g) – повертає ітератор на тип пара (key\_comp) в якому перше поле pair::first яквівалентне lower bound, а pair::second еквівалентне upper bound.

Робота з ітераторами (iterators) — аналогічно до множини та послідовних контейнерів.

Крім того, для множини (як і для інших колекцій) перевизначений оператор присвоєння (operator=).

#include <iostream>

#include <set>

#include <vector>

//#include <iterator>

using namespace std;

// function comarator

bool fncomp (int lhs, int rhs) {return lhs\*lhs<rhs\*rhs;}

//class Comparator

struct Classcomp {

bool operator() (const int& lhs, const int& rhs) const

{return lhs>rhs;}

};

template<class T>

void printSet(const T & v){

typename T::iterator itr;

for(itr=v.begin(); itr!=v.end();++itr){

cout <<" "<<\*itr;

}

cout<< endl;

}

int main (){

multiset<int> first\_set; // empty set of ints

int myints[]= {10,-20,30,40,50,50,40};

multiset<int> second\_set(myints,myints+7); // range

multiset<int> third\_set (second\_set); // a copy of second

//set<int> fourth\_set (third\_set.begin(), third\_set.end()); // iterator constructor

vector<int> v(third\_set.begin(), third\_set.end());

multiset<int> fourth\_set (v.begin(), v.end());

cout<<"2,3,4 set:";

printSet(fourth\_set);

cout<<"set size:"<< second\_set.size()<<endl;

cout<<"set max size:"<< second\_set.max\_size()<<endl;

cout<<"how many of 20 there "<< second\_set.count(20)<< " of " << \*second\_set.find(20)<<endl;

cout<<"how many of 50 there "<< second\_set.count(50)<<" of " << \*second\_set.find(50)<<endl;

cout<<"equal range for 20: " << \*second\_set.equal\_range(20).first << ", " << \*second\_set.equal\_range(20).second<<endl;

multiset<int,Classcomp> fifth\_set; // class as Compare

bool(\*fn\_pt)(int,int) = fncomp;

multiset<int,bool(\*)(int,int)> sixth\_set (fn\_pt); // function pointer as Compare

multiset <int,greater <int> > seventh\_set;

//insert elements in random order

seventh\_set.insert(40);

seventh\_set.insert(30);

seventh\_set.insert(60);

seventh\_set.insert(-20);

seventh\_set.insert(50);

seventh\_set.insert(30); // only one 30 will be added to the set

seventh\_set.insert(10);

for (multiset<int,greater <int> >::iterator it = seventh\_set.begin(); it!=seventh\_set.end();++it){

first\_set.insert(\*it);

fifth\_set.insert(\*it);

sixth\_set.insert(\*it);

}

cout<<"1 and 7th set:";

printSet(first\_set);

cout<<"5th set:";

printSet(fifth\_set);

cout<<"6th set:";

printSet(sixth\_set);

// remove all elements up to 30 in

cout << "second\_set after removal of elements less than 30:";

second\_set.erase(second\_set.begin(), second\_set.find(30));

printSet(second\_set);

// remove element with value 50 in

int num;

num = third\_set.erase (50);

cout << "third\_set.erase(50) : ";

cout << num << " removed " ;

printSet(third\_set);

//lower bound and upper bound for set seventh\_set

cout << "seventh\_set.lower\_bound(40) : "<< \*seventh\_set.lower\_bound(40) << endl;

cout << "seventh\_set.upper\_bound(40) : "<< \*seventh\_set.upper\_bound(40) << endl;

//lower bound and upper bound for set second\_set

cout << "second\_set.lower\_bound(40) : "<< \*second\_set.lower\_bound(40) << endl;

cout << "second\_set.upper\_bound(40) : "<< \*second\_set.upper\_bound(40) << endl;

}

Результат:

2,3,4 set: -20 10 30 40 40 50 50

set size:7

set max size:461168601842738790

how many of 20 there 0 of 7

how many of 50 there 2 of 50

equal range for 20: 30, 30

1 and 7th set: -20 10 30 30 40 50 60

5th set: 60 50 40 30 30 10 -20

6th set: 10 -20 30 30 40 50 60

second\_set after removal of elements less than 30: 30 40 40 50 50

third\_set.erase(50) : 2 removed -20 10 30 40 40

seventh\_set.lower\_bound(40) : 40

seventh\_set.upper\_bound(40) : 30

second\_set.lower\_bound(40) : 40

second\_set.upper\_bound(40) : 50

Зауважимо, що другий необовязковий параметр у конструкторі множини або мультмножини — це так званий функціональний обєкт, що вказує на те за яким критерієм сортуються значення в множині.

Так для впорядкування множини у спадаючому порядку можна записати:

typedef set<int,greater<int>> IntSet;

Об'єкт greater<>— це стандартний функціональний об'єкт, що вказує, що обєкти сортуються за зростанням (докладнгіше про це — в розділі про функтори).

Відображення Map в STL

Відображення — асоціативний контейнер, який зберігає дані у вигляді пари (ключ, значення). При цьому дані відсортовані по значенню ключа та кожен ключ — унікальний.

Методи для роботи з мультмножиною класифікуються так само як і для множини:

* Ідентифікатори та модифікатори обсягу ()
* Модифікатори даних
* Методи пошуку
* Робота з ітераторами

До Сі++11 шаблон класу множина мав наступні конструктори

1) конструктор за замовченням (default constructor), що створює порожню множину

2) Копіконструктор — копіює вміст іншого деку

3) Конструктор за інтервалом (range constructor) — коструює відображення за двома ітераторами на колекцію, що складється з пар обєктів.

Ідентифікатори та модифікатори обсягу (Capacity)

size() – кількість елементів колекції;

max\_size() – максимально можлива кількість елементів колекції;

empty() – якщо порожня колекція повертає true, інакше false.

Модифікатори даних (Modification):

insert(const pair g) – додає нову пару ключ/значення ‘g’ до відображення;

оператор “квадратні дужки” (operator[ key])– отримує доступ до значення за ключем;

at(key) - отримує доступ до значення за ключем (з виключенням при некоректному доступі);

erase(iterator position1, iterator position2) – видаляє елемент з інтервалу вказаного ітераторами;

erase(const g)– видаляє значення ‘g’ з множини;

clear() – видаляє всі елементи колекції.

Починаючи з C++11 додали також наступні методи:

emplace()– вставляє пару в множину, якщо цього елементу ще в неї міститься;

emplace\_hint()– повертає ітератор на елемент, який вставляється в відображення, якщо цього елементу до цього там не було;

swap()– обмінює вміст двох множин ода в іншу.

Методи пошуку (Searching):

key\_comp() / value\_comp() – повертає обєкт, що вказує як ключ/значення впорядковані (‘<‘ за замовченням);

find(const g) – повертає ітератор на значення ключа ‘g’ якщо він є у множині, або ітератор на кінець відображення, якщо його немає;

count(const g) – повертає кількість входжень ключу ‘g’ в мультимножині;

lower\_bound(const g) – повертає ітератор на перший ключ, що еквіалентний ‘g’ або на перший елемент, що точно більший за ‘g’;

upper\_bound(const g) – повертає ітератор на перший ключ, що еквіалентний ‘g’ або на останній елемент, що точно менший за ‘g’;

equal\_range((const g) – повертає ітератор на тип пара (key\_comp) в якому перше поле pair::first яквівалентне lower bound, а pair::second еквівалентне upper bound.

Робота з ітераторами (iterators): така сама як і для множини та контейнерів послідовного доступу.

#include <iostream>

#include <map>

#include <stdexcept>

using namespace std;

template <typename K, typename V>

void printMap(const map<K,V> & dict, string name){

// printing map gquiz1

typename map<K, V>::const\_iterator itr;

cout << "The map " << name <<" is : \n";

for (itr = dict.begin(); itr != dict.end(); ++itr) {

cout << " " << itr->first

<< " " << itr->second << " ";

}

cout << endl;

}

int main(){

// empty map container

map<int, int> dict1;

// insert elements in random order

dict1.insert(pair<int, int>(1, 40));

dict1.insert(pair<int, int>(3, 30));

dict1.insert(pair<int, int>(2, 60));

dict1.insert(pair<int, int>(5, 20));

dict1.insert(pair<int, int>(4, 50));

dict1.insert(make\_pair(7, 50));

dict1.insert(make\_pair(6, 10));

//dict1.insert(make\_pair(6, 15)); // replace old value

printMap(dict1, "dict1");

// assigning the elements from gquiz1 to gquiz2

map<int, int> dict2(dict1.begin(), dict1.end());

// access and change value by key

dict2[6] = 20;

dict2.at(7) = 30;

try{

dict2.at(8) = 30;

}

catch(out\_of\_range & ex){

cout<<"error Incorrect key"<<ex.what()<<endl;

}

printMap(dict2, "dict2");

// remove all elements up to

// element with key=3 in dict2

cout << "dict2 after removal of"

" elements less than key=3 : ";

dict2.erase(dict2.begin(), dict2.find(3));

printMap(dict2, "dict2 erased: ");

// remove all elements with key = 5

int num;

num = dict2.erase(5);

cout << "dict2.erase(5) : " << num << " removed \n";

printMap(dict2, "dict2 erased 5");

// lower bound and upper bound for map gquiz1 key = 5

cout << "dict1.lower\_bound(5) : " << "KEY = ";

cout << dict1.lower\_bound(5)->first << " ";

cout << " ELEMENT = "<< dict1.lower\_bound(5)->second << endl;

}

Результат:

The map dict1 is :

1 40 2 60 3 30 4 50 5 20 7 50

error Incorrect index map::at

The map dict2 is :

1 40 2 60 3 30 4 50 5 20 6 20 7 30

dict2 after removal of elements less than key=3 : The map dict2 erased: is :

3 30 4 50 5 20 6 20 7 30

dict2.erase(5) : 1 removed

The map dict2 erased 5 is :

3 30 4 50 6 20 7 30

dict1.lower\_bound(5) : KEY = 5 ELEMENT = 20

Мультвідображення (Multimap) в STL

Мультвідображення відрізняється від відображення можливістю додавати пари значень з еквівалентними ключами. Це виключаэ можливість використання квадратних дужок для досупу до елементів але розширює можливості додавання елементів.

#include <iostream>

#include <map>

#include <stdexcept>

using namespace std;

template <typename K, typename V>

void printMap(const multimap<K,V> & dict, string name){

// printing map gquiz1

typename multimap<K, V>::const\_iterator itr;

cout << "The map " << name <<" is : \n";

for (itr = dict.begin(); itr != dict.end(); ++itr) {

cout << " " << itr->first

<< " " << itr->second << " ";

}

cout << endl;

}

int main(){

// empty map container

multimap<string, string> dict1;

// insert elements in random order

dict1.insert(make\_pair("1", "word1"));

dict1.insert(make\_pair("2", "word2"));

dict1.insert(make\_pair("1", "word3"));

dict1.insert(make\_pair("2", "word4"));

dict1.insert(make\_pair("3", "word5"));

dict1.insert(make\_pair("4", "word6"));

dict1.insert(make\_pair("5", "word7"));

dict1.insert(make\_pair("5", "word8"));

printMap(dict1, "dict1");

// assigning the elements from gquiz1 to gquiz2

multimap<string, string> dict2(dict1.begin(), dict1.end());

printMap(dict2, "dict2");

// remove all elements up to

// element with key=3 in dict2

cout << "dict2 after removal of"

" elements less than key=1 : \n";

dict2.erase(dict2.begin(), dict2.find("2"));

printMap(dict2, "dict2 erased: ");

// remove all elements with key = 5

int num;

num = dict2.erase("4");

cout << "dict2.erase(4) : " << num << " removed \n";

printMap(dict2, "dict2 erased 4");

// lower bound and upper bound for map gquiz1 key = 5

cout << "dict1.lower\_bound(5) : " << "KEY = ";

cout << dict1.lower\_bound("5")->first << " ";

cout << " ELEMENT = "<< dict1.lower\_bound("5")->second << endl;

dict1.insert(make\_pair("5", "word9"));

dict1.insert(make\_pair("6", "word10"));

cout<<"all values with key=5:\n";

pair <multimap<string,string>::iterator, multimap<string,string>::iterator> eq\_range;

eq\_range = dict1.equal\_range("5");

for (multimap<string,string>::iterator it=eq\_range.first; it!=eq\_range.second; ++it)

cout << " " << it->second;

}

Результат:

The map dict1 is :

1 word1 1 word3 2 word2 2 word4 3 word5 4 word6 5 word7 5 word8

The map dict2 is :

1 word1 1 word3 2 word2 2 word4 3 word5 4 word6 5 word7 5 word8

dict2 after removal of elements less than key=1 :

The map dict2 erased: is :

2 word2 2 word4 3 word5 4 word6 5 word7 5 word8

dict2.erase(4) : 1 removed

The map dict2 erased 4 is :

2 word2 2 word4 3 word5 5 word7 5 word8

dict1.lower\_bound(5) : KEY = 5 ELEMENT = word7

all values with key=5:

word7 word8 word

Контейнери та методи, що були додані в Сі++11

Деякі корисні нововведення Сі++11

У відповідності зі стандартом C++11 ключове слово auto дозволяє вказати точний тип ітератора (за умови, що ітератор був ініціалізованим під час оголошення, так що його тип можна вивести з його початкового значення). Таким чином, безпосередня ініціалізація ітератора за допомогою функції begin() дозволяє використовувати ключовеслово auto для оголошення його типу:

for (auto pos = coll.begin(); pos != coll.end(); ++pos) {

cout << \*pos << ’ ’;

}

Легко бачити, що використання ключового слова auto робить код більш компактним.

Без ключового слова auto оголошення ітератора в циклі виглядало б у такий спосіб:

for (list<char>::const\_iterator pos = coll.begin();

pos != coll.end(); ++pos) {

cout << \*pos << ’ ’;

}

Інша перевага застосування auto полягає в тім, що цикл є стійким до змін коду, таким як модифікація типу контейнера.

Однак у такої конструкції є недолік — ітератор втрачає свою константність, тобто з'являється ризик ненавмисного присвоювання. Вираз auto pos = coll.begin() робить ітератор pos неконстантним, тому що функція begin() повертає об'єкт типу контейнер::iterator.

Для того щоб зберегти константність ітератору, у стандарті С++11 передбачені функції

cbegin() і cend().

Вони повертають об'єкт типу контейнер::const\_iterator.

Таким чином, в стандарті C++11 цикл, що дозволяє обходити всі елементи контейнеру без використання діапазонного циклу for, може мати наступний вигляд:

for (auto pos = coll.cbegin(); pos != coll.cend(); ++pos) {

...

}

Крім того, в Сі++11 зявився цикл foreach типу, який дозволяє ітеруватись по контейнеру проходячи всі елементи колекції. Тотбто, конструкція

for (type elem : coll) {

...

}

Інтерпретується як

for (auto pos=coll.begin(), end=coll.end(); pos!=end; ++pos) {

type elem = \*pos;

...

}

Приклади:

1) Виведення вектору можна записати настпним чином:

vector<string> v = {“aaa”,”bbb”, “cccc”};

for(string s: v){

cout<<s;

}

Або більш універсально:

for(const auto & s: v){

cout<<s;

}

Вставка контейнеру у контейнер

Всі асоціативні контейнери передбачають метод insert()для вставки нового елементу.

coll.insert(3);

coll.insert(1);

...

В С++11 можна просто:

coll.insert ( { 3, 1, 5, 4, 1, 6, 2 } );

Кожен вставлений елемент автоматично займає правильну позицію відповідно до критерію сортування.

Ініціалізація контейнеру

В Сі++11 додали більш зручну можливість ініціалізації контейнеру за допомогою фігурних дужок

vector<int> v{1,2,3,4,5};

Аналогічно, оскільки в нас є копіконструктор стола можлива і наступна ініціалізація

vector<int> v = {1,2,3,4,5};

Клас array

Введення класу масиву з C ++ 11 запропонувало кращу альтернативу для масивів C-стилю. Переваги класу масиву над масивом C-стилю:   
• Класи масивів знають свій власний розмір, тоді як масиви C-стилю не мають цієї властивості. Тому при переході до функцій нам не потрібно передавати розмір масиву як окремий параметр.  
• З масивом стилів С більший ризик розпаду масиву в вказівник. Класи масивів не розпадаються на вказівники  
• Класи масивів, як правило, є більш ефективними, легкими та надійними, ніж масиви C-стилю.

Методи array :  
1. at() - доступ до елементу за його номером;  
2. get() - ця функція також дозволяє отримати доступ до елементу масиву, але це не метод контейнеру, а дружня функція класу tuple.  
3. operator[] - доступ до елементу.

// C++ code to demonstrate working of array,

// to() and get()

#include<iostream>

#include<array> // for array, at()

#include<tuple> // for get()

using namespace std;

int main() {

// Initializing the array elements

array<int,6> ar = {1, 2, 3, 4, 5, 6};

// Printing array elements using at()

cout << "The array elemets are (using at()) : ";

for( inti=0; i<6; i++)

cout << ar.at(i) << " ";

cout << endl;

// Printing array elements using get()

cout << "The array elemets are (using get()) : ";

cout << get<0>(ar) << " "<< get<1>(ar) << " ";

cout << get<2>(ar) << " "<< get<3>(ar) << " ";

cout << get<4>(ar) << " "<< get<5>(ar) << " ";

cout << endl;

// Printing array elements using operator[]

cout << "The array elements are (using operator[]) : ";

for( inti=0; i<6; i++)

cout << ar[i] << " ";

cout << endl;

return0;

4. front() - повертає перший елемент масиву  
5. back() - повертає останній елемент масиву

// C++ code to demonstrate working of

// front() and back()

#include<iostream>

#include<array> // for front() and back()

using namespace std;

int main(){

// Initializing the array elements

array<int,6> ar = {1, 2, 3, 4, 5, 6};

// Printing first element of array

cout << "First element of array is : ";

cout << ar.front() << endl;

// Printing last element of array

cout<< "Last element of array is : ";

cout << ar.back() << endl;

return0;

}

6. size() - розмір масиву (цього методу не має в Сі-масивах).  
7. max\_size() - максимально можливий розмір масиву.

// C++ code to demonstrate working of

// size() and max\_size()

#include<iostream>

#include<array> // for size() and max\_size()

usingnamespacestd;

intmain()

{

// Initializing the array elements

array<int,6> ar = {1, 2, 3, 4, 5, 6};

// Printing number of array elements

cout << "The number of array elements is : ";

cout << ar.size() << endl;

// Printing maximum elements array can hold

cout << "Maximum elements array can hold is : ";

cout << ar.max\_size() << endl;

return0;

}

9. empty() - чи порожній масив;  
10. fill() - метод дозволяє заповнити масив певним значенням;

#include<iostream>

#include<array> // for fill() and empty()

usingnamespacestd;

intmain()

{

// Declaring 1st array

array<int,6> ar;

// Declaring 2nd array

array<int,0> ar1;

// Checking size of array if it is empty

ar1.empty()? cout << "Array empty":

cout << "Array not empty";

cout << endl;

// Filling array with 0

ar.fill(0);

// Displaying array after filling

cout << "Array after filling operation is : ";

for( inti=0; i<6; i++)

cout << ar[i] << " ";

return 0;

}

Клас forward\_list

Однозвязний список – в деяких ситуаціях працює швидше ніж двозвязний список, але методи доступу та роботи з ітератором тут відрізняються, оскільки на відміну від інших контенерів ітератор тут може рухатись лише в одному напрямку.

Методи цього контейнеру

Модифікатори:

operator= - оператор присвоєння

assign(n,val) – присвоює значення контейнеру

Доступ до елементів:

front доступ до першого елементу

Робота з ітераторами:

before\_begin() - повертає ітератор перед початком списку;

cbefore\_begin - повертає константний ітератор перед початком списку;

begin() - повертає початковий ітератор;

cbegin() - повертає константний початковий ітератор;

end() - повертає ітератор на кінець списку;

cend() - повертає контстантний ітератор на кінець списку;

Ідентифікатори обсягу:

empty() - чи порожній контейнер;

max\_size - максиальна кількість елементів контейнеру;

Модификатори:

clear() - очищення вмісту контейнеру;

insert\_after() - вставка елементів після даного ітертору;

emplace\_after() - створення елементів після даного ітератору;

erase\_after() - видаляє елемент після ітератору;

push\_front() - вставляє елемент в початок;

emplace\_front() - створює елемент на початку;

pop\_front() - видаляє перший елемент;

resize() - змінює кількість виділенних елементів масиву;

swap() - обмін змісту двох списків;

merge() - обєднує два відсортованих списки;

splice\_after () - перемещує елементи з іншого forward\_list;

remove(g) - видаляє даний елемент або список елементів;

remove\_if( func) - видаляє елементи по вказаному критерію (func - функція або функціональний обєкт)

reverse() - інвертує список;

unique() - видаляє послідовні однакові елементи;

sort() - сортує список

#include <forward\_list>

#include <string>

#include <iostream>

template<typename T>

std::ostream& operator<<(std::ostream& s, const std::forward\_list<T>& v) {

s.put('[');

char comma[3] = {'\0', ' ', '\0'};

for (const auto& e : v) {

s << comma << e;

comma[0] = ',';

}

return s << ']';

}

int main()

{

// c++11 initializer list syntax:

std::forward\_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};

std::cout << "words1: " << words1 << '\n';

// words2 == words1

std::forward\_list<std::string> words2(words1.begin(), words1.end());

std::cout << "words2: " << words2 << '\n';

// words3 == words1

std::forward\_list<std::string> words3(words1);

std::cout << "words3: " << words3 << '\n';

// words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}

std::forward\_list<std::string> words4(5, "Mo");

std::cout << "words4: " << words4 << '\n';

}

Результат

words1: [the, frogurt, is, also, cursed]

words2: [the, frogurt, is, also, cursed]

words3: [the, frogurt, is, also, cursed]

words4: [Mo, Mo, Mo, Mo, Mo]