Лекція 12: Стандартна бібліотека шаблонів STL

Потреба в бібліотеці шаблонів

Створення шаблонів класів дозволяє програмістам створювати корисні програмні та алгоритмічні класи та функції. Звичайно, розробники Сі++ скористалися цією можливістю для того щоб збільшити функціонал та можливості мови. В доповнення до бібліотеки потокового введення/виведення та бібліотеки роботи з рядками <string> було створено потужну систему бібліотек побудованих за об'єктно-орієнтовним принципом та об'єднаних спільною ідеєю та структурою. Ця бібліотека робить легкою можливість використання багатьох відомих алгоритмів на Сі++. Вона була розроблена спочатку як вільна бібліотека, а потім була включена в стандарт та отримала назву "стандартна бібліотека шаблонів" (STL – Standard Template Library).

Огляд стандартної бібліотеки шаблонів C ++ (STL)

Стандартна бібліотека шаблонів (STL) - це набір шаблонів C ++ для створення загальних структурних даних і функцій, таких як списки, стеки, масиви і т.д. Це бібліотека контейнерних класів, алгоритмів і ітераторів. Це узагальнена бібліотека, тому її параметри параметризуються. Робоче знання шаблонів класів є обов'язковою умовою для роботи з STL.

Стандартна бібліотека шаблонів складається з таких чотирьох компонентів:

- Algorithms (алгоритми) : набір корисних функцій, спеціально призначених для використання на діапазонах елементів, які містяться в заголовочних файлах <algorithm> та <numeric>. Вони діють на контейнери або масиви і забезпечують засоби для виконання корисних дій з вмістом контейнерів.
- Containers (контейнери) : шаблони класів, що дозволяють зберігати об'єкти і дані різних типів за допомогою різних структур даних, а отже дозволяють оптимізувати роботу з ними під різні типи задач.
- Functions (функціонали або функтори): STL включає класи, які перевантажують оператор виклику функції. Екземпляри таких класів називаються функціональними об'єктами або функторами. Функтори дозволяють налаштувати роботу пов'язаної функції за допомогою параметрів, що передаються.
- Iterators (ітератори): STL включає класи, що можуть використовуватися для роботи з послідовністю значень в різних контейнерах забезпечуючи таким чином універсальність типів даних STL.
- Утілити та псевдоконтейнери. Ще одним компонентом, що може включатись до STL – це утілити бібліотеки Сі++ та так звані псевдоконтейнери,

зокрема ті, що входять до складу пакетів <utility>, <bitset>, <tuple>, <valarray> і т.п., а іноді сюди відносять також і бібліотеку <string>.

Утілити utility

Розгляд бібліотеки ми почнемо з розгляду деяких корисних функцій та класів бібліотеки <utility>.

Контейнер Пара

До Сі++11 в даній бібліотеці містився лише шаблон класу Пари (pair).
Контейнер Pair (пара) являє собою простий контейнер, визначений у заголовочному файлі <utility>, що складається з двох елементів даних або об'єктів. Даний шаблон класу можна описати наступним чином:
• Перший елемент позначається як «first», а другий - як «second», а порядок фіксований (first, second).
• Пара використовується для об'єднання двох значень, які можуть бути різними за типом. Пара забезпечує спосіб зберігання двох неоднорідних об'єктів у вигляді однієї змінної.
• Пара може бути означена, скопійована та порівняна. Масив об'єктів, виділених для відображення (map) або hash_map, за замовчуванням мають тип "пари", в якому всі елементи "first" є унікальними ключами, пов'язаними зі значеннями "second".
• Для доступу до елементів використовується ім'я змінної, за яким слідує оператор точки, за яким слідує ключове слово first або second.

```
//CPP program to illustrate pair STL
#include <iostream>
#include <utility>
using namespace std;

int main() {
pair <int, char> PAIR1;
PAIR1.first = 100;
PAIR1.second = 'G';

cout << PAIR1.first << " ";
cout << PAIR1.second << endl ;
pair <string,unsigned> PAIR2 ("Мехмат", 65);

cout << PAIR2.first << " ";
cout << PAIR2.second << endl ;
```

```
}
```

Ініціалізація пари:
pair (data_type1, data_type2) Pair_name (value1, value2) ;
Приклади:

```
pair g1; //default
pair g2(1, 'a'); //initialized, different data type
pair g3(1, 10); //initialized, same data type
pair g4(g3); //copy of g3
```

Ще один варіант ініціалізації — визначена в цьому ж пакеті функція make_pair(), яка визначена наступним чином:

```
template <class T1,class T2>
pair<T1,T2> make_pair (T1 x, T2 y){
return ( pair<T1,T2>(x,y) );
}
g2 = make_pair(1, 'a');
```

В класі визначені конструктори:

- Стандартний конструктор за замовченням:

```
pair();
```

- Стандартний конструктор копіювання:

```
template<class U, class V> pair (const pair<U,V>& pr);
```

- Конструктор ініціалізації (сеттер):

```
pair (const first_type& a, const second_type& b);
```

Примітка: Якщо поле не ініціалізоване конструктором за замовченням, то воно автоматично ініціалізується нулем.

```
int main() {
pair <int, double> PAIR1 ;
pair <string, char> PAIR2 ;
cout << PAIR1.first ;  //it is initialised to 0
cout << PAIR1.second ; //it is initialised to 0
cout << " ";
cout << PAIR2.first ;  //it prints nothing i.e NULL
cout << PAIR2.second ; //it prints nothing i.e NULL
}
```

Методи класу

make_pair() : створює пару без визначення типів полів явно
Pair_name = make_pair (value1,value2);
Приклад

```
pair <int, char> PAIR1 ;
```

```
pair <string, double> PAIR2 ("Студент", 4.23) ;
pair <string, double> PAIR3 ;

PAIR1.first = 100;
PAIR1.second = 'G';

PAIR3 = make_pair ("Мехмат кращій",4.56);

cout << PAIR1.first << " ";
cout << PAIR1.second << endl ;

cout << PAIR2.first << " ";
cout << PAIR2.second << endl ;

cout << PAIR3.first << " ";
cout << PAIR3.second << endl ;
```

Оператори **(=, ==, !=, >=, <=) :**

- **(=)** Присвоює нову пару – конструктор копіювання.

pair& operator= (const pair& pr);

- **Рівність (==)** – порівнює поля pair1 та pair2. Пари рівні якщо pair1.first рівне pair2.first та pair1.second рівне pair2.second.
- **Нерівність (!=)** – протилежний до рівності.
- **Логічні (>=, <=) оператори порівняння -** порівнює лише перше поле пари.

Приклад.

```
pair<int, int>pair1 = make_pair(1, 12);
pair<int, int>pair2 = make_pair(9, 12);
cout << (pair1 == pair2) << endl;
cout << (pair1 != pair2) << endl;
cout << (pair1 >= pair2) << endl;
cout << (pair1 <= pair2) << endl;
cout << (pair1 > pair2) << endl;
cout << (pair1 < pair2) << endl;
```

Результат:

```
0
1
0
1
0
```

1
Примітка: З C++20 у якості операторів порівняння радять використовувати оператор трьохстороннього порівняння (operator<=>)
Примітка: з C++11 додано також метод **swap,** що змінює місцями значення однієї пари зі значеннями іншої пари тих самих типів.

```
pair<char, int>pair1 = make_pair('A', 1);
pair<char, int>pair2 = make_pair('B', 2);

cout << "Before swapping:\n ";
cout << "Contents of pair1 = "<< pair1.first << " "<< pair1.second ;
cout << "Contents of pair2 = "<< pair2.first << " "<< pair2.second ;
pair1.swap(pair2);

cout << "\nAfter swapping:\n ";
cout << "Contents of pair1 = "<< pair1.first << " "<< pair1.second ;
cout << "Contents of pair2 = "<< pair2.first << " "<< pair2.second ;
```

Результат:

```
Before swapping:
Contents of pair1 = (A, 1)
Contents of pair2 = (B, 2)

After swapping:
Contents of pair1 = (B, 2)
Contents of pair2 = (A, 1)
```

З використанням цього шаблону полегшується використання в багатьох ситуаціях роботи зі стандартними контейнерами STL, особливо це стосується шаблонів класів відображення (хештаблиць) map/multimap.

Контейнери

Контейнери або класи контейнерів зберігають об'єкти і дані. В загальній складності сім стандартних класів контейнерів першого класу і три класи адаптерів контейнерів і лише сім файлів заголовків, які забезпечують доступ до цих контейнерів або адаптерів контейнерів.

• **Контейнери послідовності** (**Sequence containers**) - реалізують структури даних, до яких можна звертатися послідовно тобто за номером елементу:

- vector
- list
- deque
- arrays (C++11)

- forward_list(C++11)

• **Контейнери адаптери (Container Adaptors)** - реалізують інтерфейс для контейнерів баз даних, найбільш прості структури даних, що дозволяють додавати та видаляти елементи без забезпечення послідовного доступу:

- queue
- priority_queue
- stack

• **Асоціативні контейнери (Associative Containers)** – структури даних що зберігають сортовані структури даних, що дозволяють швидкий пошук (зі складністю O(log n)):

- set
- multiset
- map
- multimap

• З C++11 до стандартних контейнерів додали також **невідсортовані асоціативні контейнери (Unordered Associative Containers)** які складаються з невідсортованих структур даних- хеш таблиць та відображень:

- unordered_set (C++11)
- unordered_multiset (C++11)
- unordered_map (C++11)
- unordered_multimap (C++11)

Контейнери адаптори

Стек (Stack)

Стек – це шаблон класу який реалізує концепцію доступу до даних LIFO (Last In First Out – останній зайшов, перший вийшов), в який елемент додається до верхівки структури та відповідно опускає вниз той елемент, що до цього знаходився на верхівці стеку. Коли ж відбувається операція видалення елементу – видаляється верхівка стеку, та той елемент, що знаходився нижче (якщо він був) повертається на верхівку. Нижче нової вершини так саме знаходиться попередник нового елементу вершини і так далі.
Одна з можливих реалізацій шаблону була розглянута в попередньому розділі. В стандартній бібліотеці, зрозуміло, більш якісна версія класу шаблону, що дозволяє виділяти пам’ять під кожен новий елемент та кидає виключення у випадку некоректного видалення елементу.
Для використання цього контейнеру потрібно *підключити заголовочний файл <stack>*.

Методи цього класу стеку працюють дуже швидко (час виконання – O(1)), а саме:

- empty() – булева функція, що повертає true, якщо стек порожній;
- size() – повертає розмір стеку;
- top() – повертає вказівник на останній елемент(верхівку) стеку;
- push(T g) –додає елемент ‘g’ у верхівку стеку;
- pop() – видаляє верхівку стеку.

Також для стеку визначені оператори присвоєння та порівняння (>,>=,==,!=,<,<=) які здійснюють відповідні поелементні порівняння для вмістів двох стеків.

Конструктори визначений за замовченням та аналог копіконструктору, що приймає стек.

Примітка. З Сі++11 додано також методи swap() для обміну значень двох стеків та emplace() - синонім push(), який дозволяє додавати елементи в колекцію по аргументах типу, тобто без зайвого копіювання та переміщення у рамках move-семантики стандарту Сі++11.

```
// C++ програма для демонстрації STL стеку
#include <iostream>
#include <string>
#include <stack>
using namespace std;
// виводить стек у зворотньому порядку
void showstack(const stack <string> & s){
stack <string> copy_s(s); // copy-constructor to create copy of s
// поки стек не порожній – виводить верхівку та видаляє її
while (!copy_s.empty()){
cout<<'\t'<<copy_s.top();
copy_s.pop();
}
cout << '\n';
}

int main(){
stack <string> s;
s.push("sator");
s.push("arepo");
s.push("tenet");
s.push("opera");
s.push("rotas");
cout <<"The stack is : ";
showstack(s);
```

```
//The stack is : rotas opera tenet arepo sator
cout<<"\ns.size():"<<s.size();//5
cout << "\ns.top() : " << s.top();//rotas
cout << "\ns.pop() : "; //
s.pop();
showstack(s); // opera tenet arepo sator
//s.pop() : // tenet arepo sator
}
```

Результат:

```
The stack is :          rotas  opera tenet arepo sator
s.size():5
s.top() : rotas
s.pop() :       opera  tenet arepo sator
```

Черга (Queue) в (STL)

На відміну від стеку, черга - шаблон класу який реалізує концепцію доступу до даних FIFO (First In First Out – перший зайшов, перший вийшов), в який елемент додається до кінця структури та видаляється елемент, що стоїть в початку структури. Тут пряма аналогія з “чергою” у магазині.

Для використання цього контейнеру потрібно підключити заголовочний файл <queue>.

Методи класу працюють зі складністю O(1):

- empty() – булева функція, що повертає true, якщо черга порожня та false в іншому випадку;
- size() – повертає розмір черги;
- push(T g) –додає елемент ‘g’ у кінець черги;
- pop() – видаляє початок черги;
- front() - повертає посилання на елемент початку черги;
- back() - повертає посилання на останній елемент черги.

Також для черги визначені оператори присвоєння та порівняння (>,>=,==,!=,<,<=) які здійснюють відповідні поелементні порівняння для вмісту двох черг.

Конструктори, що є для черги: конструктор за замовченням та аналог копіконструктору.

Примітка. З Ci++11 додано також методи swap() для обміну значень двох черг та emplace() - що має функціонал push().

```
// Queue in Standard Template Library (STL)
#include <iostream>
#include <queue>

using namespace std;
// виводить чергу в порядку заповнення для будь-якого стандартного типу
template <typename T>
void showq(const queue <T> & gq) {
// створюємо копію черги
queue <T> g = gq;
// видаляємо та дивимось всі елементи в неї з початку
while (!g.empty()) {
        cout << '\t' << g.front();
      g.pop();
 }
cout << '\n';
}


int main() {
  queue <int> q1;
  q1.push(10);
  q1.push(20);
  q1.push(30);
  q1.push(15);
  cout << "The queue q1 is : ";
  showq(q1);
  cout << "\nq1.size() : " << q1.size();
  cout << "\nq1.front() : " << q1.front();
  cout << "\nq1.back() : " << q1.back();
  cout << "\nq1.pop() : ";
  q1.pop();
  showq(q1);
}
```

Результат:

```
The queue q1 is :   10      20      30      15

q1.size() : 4
q1.front() : 10
```

```
q1.back() : 15
q1.pop() :    20      30      15
```

Пріоритетна черга (Priority Queue)в STL

Пріоритетна черга відрізняється від звичайної черги тим, що першим елементом на видалення йде не перший елемент, а елемент, що приймає найбільше значення. Тобто пріоритетна черга - це тип контейнеру-адаптеру, спеціально розроблений таким чином, що перший елемент черги є найбільшим із усіх елементів черги, а елементи контейнеру розташовані в порядку неспадання (отже, ми можемо бачити, що кожен елемент черги має пріоритет (фіксований порядок).
Для використання цього контейнеру потрібно підключити заголовочний файл <queue>.
Методи контейнеру:

- empty() – булева функція, що повертає true, якщо черга порожня. Час виконання O(1);
- size() – повертає розмір черги – час виконання O(1);
- top() – повертає вказівник на останній елемент(верхівку) черги. Час виконання O(1);
- push(T g) –додає елемент ‘g’ у кінець черги. Час виконання повинен бути O(log n) ;
- pop() – видаляє початок черги. Час виконання час повинен бути O(log n).

Також для пріоритетної черги визначені оператори присвоєння та порівняння (>,>=,==,!=,<,<=) які здійснюють відповідні поелементні порівняння для вмістів двох черг.
Конструктори для цього класу - конструктор за замовченням та аналог копіконструктору, що приймає цю колекцію.

```
// Queue in Standard Template Library (STL)
#include <iostream>
#include <queue>

using namespace std;
// виводить чергу в порядку пріорітету для стандартних типів
template<class T>
void showpq(const priority_queue <T> & gq) {
  priority_queue <T> g(gq);
```

```
  while (!g.empty()){
   cout << '\t' << g.top();
   g.pop();
   }
}
// виводить чергу в порядку пріорітету для типу Пара від двох цілих – спеціалізація showpq
template<>
void showpq(const priority_queue <pair<int,int> > & gq) {
  //‘>>’ should be ‘> >’ within a nested template argument list before C++11
  priority_queue <pair<int,int> > g(gq);
  while (!g.empty()){
   pair<int,int> tmp = g.top();
   cout << "\t(" << tmp.first<<", "<<tmp.second<<")";
   g.pop();
   }
}
int main() {
  priority_queue <int> q2;
   q2.push(10);
   q2.push(30);
   q2.push(20);
   q2.push(5);
   q2.push(1);
   cout << "The priority queue q2 is : ";
   showpq(q2);
   cout << "\nq2.size() : " << q2.size();
   cout << "\nq2.top() : " << q2.top();
   cout << "\nq2.pop() : ";
   q2.pop();
   showpq(q2);

   priority_queue <pair<int,int> > q3;/* коректно до c++20 – інакше для pair не визначено оператор < */
   q3.push(make_pair(1,2));
```

```
  q3.push(pair<int,int>(3,4));
  q3.push(make_pair(1,4));
  q3.push(make_pair(2,10));
  cout << "The priority queue q3 is : ";
  showpq(q3);
}
```

Результат:

```
The priority queue q2 is :        30     20     10     5      1
q2.size() : 5
q2.top() : 30
q2.pop() :    20     10     5      1
The priority queue q3 is :        (3, 4)   (2, 10) (1, 4) (1, 2)
```

Примітка. Звертаємо увагу, що тут використано контейнер від шаблону класу. До Сі++11 потрібно було завжди в цьому випадку писати визначення типу вигляду:

```
priority_queue <pair<int,int> > q3; // обов'язковий пробіл між двома останніми дужками > >
```

І хоча останні стандарти дозволяють не робити цей пробіл, для сумісності коду краще цей пробіл ставити.

Контейнери послідовності

Контейнери послідовності (Sequential containers) або контейнери послідовного доступу дозволяють зберігати масиви даних таким чином, щоб можна було б, на відміну від контейнерів адаптерів, отримати доступ до довільного елементу контейнеру. Таким чином, в цих контейнерах можна проходити зміст контейнеру за допомогою аналогу вказівника – ітератору. Різниця між даними контейнерами полягає в сутності структур даних та можливостях контейнера (дек – розширений функціонал для контейнерів адаптерах, вектор та масив – для роботи з динамічним та сталого розміру масивами, списки – для даних де потрібно часто додавати/видаляти дані всередині).

Дек (Deque)

Дек або двонаправлена черга є контейнером послідовності з функцією розширення і стиснення на обох кінцях. Це фактично об'єднання структур стек та черга з доданим функціоналом послідовного доступу. Вони схожі на вектори, але більш ефективні у випадку вставки і видалення елементів. На відміну від векторів, для деків послідовне виділення пам'яті не може бути гарантованим. Деки в основному є реалізацією структури даних двонаправлена черга. Структура даних черг дозволяє вставляти тільки в кінці і видаляти з початку. Це схоже на чергу в реальному житті, де люди видаляються з початку черги і додаються назад. Можна сказати, шо дек - це особливий випадок черг, де операції вставки та видалення можливі на обох кінцях.
Методи для цієї структури наступні:

- insert() – вставляє в дек нові елементи та повертає ітератор на початок встановлюваних елементів;
- max_size() – максимальний розмір контейнера;
- assign() – присвоює нові значення контейнеру;
- resize() – змінює розмір деку;
- push_front() – вставляє елементи на початок;
- push_back() – вставляє елемент в кінець;
- pop_front() – видаляє елемент в початок
- pop_back() – видаляє елемент з кінця;
- front() – повертає посилання на перший елемент;
- back() – повертає вказівник на останній елемент;
- clear() – видаляє всі елементи деку;
- erase() – видаляє елементи всередині вказаного діапазону;
- empty() – перевіряє чи порожній дек;
- size() – повертає розмір деку;
- operator[] – повертає елемент заданої позиції;
- at() – повертає елемент заданої позиції;
- operator= – присвоює новий дек.

Також для черги визначені оператори присвоєння та порівняння (>,>=,==,!=,<,<=) які здійснюють спочатку порівняння для розмірів контейнеру, а потім відповідні поелементні порівняння для вмістів двох контейнерів.
Крім того, визначені методи для роботи з ітераторами, які будуть розглянуті в розділі про ітератори.

Шаблон класу дек має також наступні конструктори:

1) Конструктор за замовченням – створює порожній дек.

2) Конструктор заповнення - ініціалізує контейнер з даною кількістю даних елементів:

explicit deque (size_type n, const value_type& val = value_type())// Будує контейнер з *n*елементів. Кожен елемент дорівнює *val*.

Приклад

deque d1(5, 10); // Вміст деку d1 : 10,10,10,10,10

3) Копіконструктор — копіює вміст іншого деку.

4) Конструктор за інтервалом (range constructor) — конструює дек за двома ітераторами іншого деку.

Примітка. З Ci++11 додано також методи swap() для обміну значень двох деків та методи додавання елементів emplace_front(), emplace_back(), а також метод для зміни розміру структури під дані shrink_to_fit(). Також було додано move-конструктори та конструктори за списком ініціалізації.

```
#include <iostream>
#include <deque>
#include <stack>
using namespace std;

// виведення деку від стандартного класу
template<typename T>
void showdq(const deque <T> & g){
 for(size_t i=0; i<g.size();++i)
   cout << '\t'<< g[i];
 cout << '\n';
}
// виведення стеку
template<class T>
void printStack(const stack<T> & st){
 stack<T> st1(st);
 while(!st1.empty()){
   cout << ", "<< st1.top();
```

```
   st1.pop();
 }
 cout << ";\t";
}
// виведення деку від стеку
template<typename T>
void showdq(const deque<stack<T> > & g){
  for(size_t i=0; i<g.size();++i) {
    cout << "stack "<< i<<":";
    printStack(g[i]);
 }
 cout << '\n';
}

int main(){
// дек від рядків Ci
deque <char*> deq1;
deq1.push_back((char*)"cadabra");
deq1.push_front((char*)"abra");
deq1.push_back((char*)"bums");
deq1.push_front((char*)"\n");
cout << "The deque deq1 is : ";
showdq(deq1);
cout << "\ndeq1.size() : "<< deq1.size();
cout << "\ndeq1.max_size() : "<< deq1.max_size();
cout << "\ndeq1.at(2) : "<< deq1.at(2);
cout << "\ndeq1.front() : "<< deq1.front();
cout << "\ndeq1.back() : "<< deq1.back();
cout << "\ndeq1.pop_front() : ";
deq1.pop_front();
showdq(deq1);
cout << "\ndeq1.pop_back() : ";
deq1.pop_back();
showdq(deq1);
// дек від стеків
```

```
deque<stack<int> > deq2;
stack<int> a1;
a1.push(1);a1.push(2);
deq2.push_back(a1);

stack<int> a2(a1);
a2.push(3);
deq2.push_back(a2);

deq2.push_front(a1);
a1.pop();
deq2.push_back(a1);
showdq(deq2);
}
```

Результат:

```
The deque deq1 is :
        abra   cadabra       bums

deq1.size() : 4
deq1.max_size() : 2305843009213693951
deq1.at(2) : cadabra
deq1.front() :

deq1.back() : bums
deq1.pop_front() :          abra   cadabra       bums

deq1.pop_back() : abra   cadabra
stack 0:, 2, 1;         stack 1:, 2, 1;         stack 2:, 3, 2, 1;      stack 3:, 1;
```

Вектор

Вектор є, мабуть, найбільш популярним та застосовуваним контейнером STL. Вектори є реалізацією динамічних масивів з можливістю автоматично змінювати розмір, коли елемент вставляється або видаляється, а їх зберігання автоматично обробляється контейнером. Елементи вектору розміщуються в контейнері, так що до них можна отримати доступ і пройти всі елементи за допомогою ітераторів. У векторах дані вставляються в кінець. Вставка в кінець займає змінний час, оскільки іноді може виникнути потреба у розширенні масиву. Видалення

останнього елемента займає лише сталий час (O(1)), оскільки зміна розміру не відбувається. Вставка та видалення елементу на початку або всередині є лінійними по часу O(n).

Методи, які містить клас вектор можна класифікувати наступним чином:

- Модифікатори даних (Modifiers);
- Ідентифікатори та модифікатори обсягу (Capacity);
- Методи доступу до елементів (Access);
- Робота з ітераторами (Iterators).

До Сі++11 шаблон класу множина мав наступні конструктори
1) конструктор за замовченням (default constructor), що створює порожній вектор
2) Копіконструктор — копіює вміст іншого вектору
3) Конструктор за інтервалом (range constructor) — конструює вектор за двома ітераторами іншої колекції.
4) Конструктор заповнення - ініціалізує контейнер з даною кількістю даних елементів:

explicit deque (size_type n, const value_type& val = value_type())// Будує контейнер з *n* елементів. Кожен елемент дорівнює *val*.

Приклад.

vector v1(3, 'A'); // Вміст v1 : A,A,A

Модіфікатори даних (Modifiers):

- assign(n, v) – присвоюємо (ініціалізуємо) значення у векторі — n значень v присвоюється у вектор;
- push_back(g) – додає елемент g у кінець вектору;
- pop_back() – видаляє елемент з кінкця вектору;
- insert(g, it_beg) – додає елемент(елементи) у вказану ітератором позицію;
- erase(it_begin, it_end) – видаляє елементи з вектору по вказаному ітераторами інтервалу;
- clear() – видаляє всі елементи з вектору;
- emplace(g) – розширює вектор вставляючи нові елементи на позицію вказану ітератором (C++11);
- emplace_back(g) – додає нові елементи в кінець вектору(C++11);
- swap(v) – міняє значення векторів з одного в інший. Розміри векторів (але не тип) можуть відрізнятись(C++11).

// C++ program to illustrate the methods Modifiers in vector

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
// Assign vector
vector<int> v;
// fill the array with 10 five times
v.assign(5, 10);

cout << "The vector elements are: ";
for(int i = 0; i < v.size(); i++)
cout << v[i] << " ";

// inserts 15 to the last position
v.push_back(15);
int n = v.size();
cout << "\nThe last element is: "<< v[n - 1];

// removes last element
v.pop_back();

for(int i = 0; i < v.size(); i++)
 cout << v.at(i) << " ";
// erases the vector
v.clear();
cout << "\nVector size after erase(): "<< v.size();


// prints the vector
cout << "\nThe vector elements are: ";
for(int i = 0; i < v.size(); i++)
 cout << v.at(i) << " ";

// inserts 5 at the beginning
v.insert(v.begin(), 5);

cout << "\nThe first element is: "<< v[0];

// removes the first element
v.erase(v.begin());
```

```
cout << "\nThe first element is: "<< v[0];
}
```
Результат:
```
The vector elements are: 10 10 10 10 10
The last element is: 1510 10 10 10 10
Vector size after erase(): 0
The vector elements are:
The first element is: 5
The first element is: 5Size : 5
```

Ідентифікатори та модифікатори обсягу **(Capacity)**

- size() – кількість елементів вектору;
- max_size() – максимальна кількість елементів вектору;
- capacity() – розмір алокатору, виділеного під цій контейнер, тобто справжній розмір даного вектору;
- resize(n) – змінює розмір вектору на заданий;
- empty() – Повертає true, якщо контейнер порожній;
- reserve(n) –виділяє пам'ять для зберігання рівно n елементів;
- shrink_to_fit() – Зменшує розмір контейнеру видаляючи неініціалізовані елементи (C++11).

Приклад:
```
vector<int> g1;
for(int i = 1; i <= 5; i++)
  g1.push_back(i);

cout << "Size : "<< g1.size();
cout << "\nCapacity : "<< g1.capacity();
cout << "\nMax_Size : "<< g1.max_size();

// resizes the vector size to 4
g1.resize(4);

// prints the vector size after resize()
cout << "\nSize : "<< g1.size();

// checks if the vector is empty or not
if(g1.empty() == false)
cout << "\nVector is not empty";
else
```

```
cout << "\nVector is empty";

for(vector<int>::iterator it = g1.begin(); it != g1.end(); it++)
  cout << *it << " ";
```

Результат:
Capacity : 8
Max_Size : 4611686018427387903
Size : 4
Vector is not empty1 2 3 4

Методи доступу до елементів**:**

- at(n) – повертає посилання на ‘n’-ий елемент вектору;
- operator [n] – перевантажений оператор для доступу до ‘n’-го елементу вектору;
- front() – повертає посилання на перший елемент контейнеру;
- back() – повертає посилання на останній елемент контейнеру;
- data() – повертає вказівник на місце де зберігаються дані.

Приклад.

```
string massiv[] = {"first", "second", "third", "forth", "fifth"};
vector<string> g2(massiv, massiv + 5);

cout << "\nReference operator [g] : g1[2] = "<< g2[2];
cout << "\nat : g1.at(4) = "<< g2.at(4);
cout << "\nfront() : g1.front() = "<< g2.front();
cout << "\nback() : g1.back() = "<< g2.back();
// pointer to the first element
string* pos = g2.data();
cout << "\nThe first element is "<< *pos;
```

Результат:
Reference operator [g] : g1[2] = third
at : g1.at(4) = fifth
front() : g1.front() = first
back() : g1.back() = fifth
The first element is first

Різницю між оператором [] та методом at() полягає в обробці ситуації з невірним індексом вектору. При використанні функції at() при спробі звернення по неприпустимому індексу буде генеруватись виключення out_of_range, в той яас як при використанні квадратних дужок поведінка компілятора невизначена:

```
int mas[] = { 1, 2, 3, 4, 5};
std::vector<int> numbers(mas,mas+5);
try
{
   int n = numbers.at(5); //oops...
   std::cout<<"n="<<n;
}
catch (std::out_of_range e)
{
   std::cout << "Caught: Incorrect index 1" << std::endl;
}

std::vector<int> numbers2(mas,mas+5);
try
{
   int n = numbers2[5]; // ??? - поведінка невизначена
   std::cout<<"n="<<n; // n=0 or maybe n=502012792....
}
catch (std::out_of_range e)
{
   std::cout << "Shouldnt caught: Incorrect index 2" << std::endl;
}
```

Результат:
Incorrect index 1
n=502012792

Робота з ітераторами:

- begin() – повертає ітератор на перший обєкт вектору
- end() – повертає ітератор на останній обєкт вектору

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

- rbegin() – повертає ітератор на останній обєкт вектору як на початковий (reverse beginning). Рухається з останнього елементу до першого;
- rend() – повертає ітератор на перший обєкт вектору як на останній (reverse beginning). Рухається він з останнього елементу до першого;
- cbegin() – повертає константний ітератор на преший елемент;
- cend() – повертає константний ітератор на останній елемент;
- crbegin() – повертає константний реверсивний оператор на початок;

- crend() – повертає константний реверсивний оператор на кінець вектору.

Список (list) у STL

Списки є контейнерами послідовностей, які дозволяють як і асоціативні контейнери не виділяти цілком однорідну пам'ять під масиви, а зберігати її “порціями” разом з посиланнями на попередній та наступний об’єкти. У порівнянні з вектором список має повільну обробку, але коли знайдена позиція, вставка і видалення є швидкими. Зазвичай, коли ми говоримо список, ми говоримо про подвійно пов'язаний список. Для реалізації однозв'язного списку потрібно використовувати forward_list (C++11).

- Модифікатори даних (Modifiers)
- Ідентифікатори та модифікатори обсягу (Capacity)
- Методи доступу до елементів (Access)
- Робота з ітераторами (Iterators)

Методи класу List:

Ідентифікатори та модифікатори обсягу:

- size() – повертає кількість елементів;
- resize()- перевизначає розмір контейнеру;
- empty() – повертає чи порожній список (true) чи ні (false);
- max_size()–максимальна кількість елементів списку.

Методи доступу до елементів (тут значно менше методів ніж у векторів):

- front() – перший елемент списку;
- back() – останній елемент списку;

Модифікатори даних (а ось ці методи значно різноманітні ніж у векторі):

- assign() – визначає нові елементи шляхом заміни їх новими значеннями;
- push_front(g) – додає елемент ‘g’ у початок списку;
- push_back(g) – додає елемент ‘g’ в кінець списку;
- pop_front() – видаляє перший елемент списку, зменшує розмір списку на 1;
- pop_back() – видаляє останній елемент списку, зменшує розмір списку на одиницю;
- insert() – додає елемент перед поточним елементом списку;
- clear() – видаляє елементи списку, його розмір стає нульовим;
- erase() – видаляє елементи між двома вказаними вказівниками (ітераторами);

- merge() – зливає два списки в один;
- remove() – видаляє всі елементи списку, що є рівними даному;

- remove_if() – видаляє елементи, що задовольняють деякій булевій умові;
  - reverse() – інвертує список;
  - sort() – сортує елементи по неспаданню;
  - splice()– передає елементи з одного списку в інший;
  - unique() – видаляє дублюючи елементи списку.

Починаючи з C++11, додано наступні методи:

- emplace() – розширює список, додаючи в нього ще елементи;
- emplace_front() – вставляє елемент на початок списку;
- emplace_back() – вставляє елемент в кінець списку;
- swap()– обмінює значення двох списків, якщо в них співпадають розміри та тип.

Ітератори

- begin() - повертає ітератор що вказує на початок списку;
- end() - повертає ітератор, що вказує на кінець списку;
- rbegin() - повертає реверсивний ітератор на кінець списку;
- rend() - повертає реверсивний ітератор на початок;
- cbegin() - константний ітератор на початок;
- cend() – повертає константний ітератор на кінець;
- crbegin() – повертає константний реверсивний ітератор на початок;
- crend() – повертає константний оператор на кінець списку.

```
#include <iostream>
#include <list>
#include <iterator> // advance
using namespace std;

//function for printing the elements in a list
template <typename T>
void showlist(const list <T> & g){
  // to get iterator of template type - write template before assignment
  typename list<T>::const_iterator it = g.begin();
  cout << *it;
  it++;
  for(; it != g.end(); ++it)
    cout << ','<< *it ;
 cout << '\n';
```

```
}

bool is_even(int n){ return n%2 == 0;}

int main(){

list <int> gqlist1, gqlist2;

for(int i = 0; i < 8; ++i){
  gqlist1.push_back(i * 2);
}
int mas[] = {1,2,5,3,4};
gqlist2.assign(mas,mas+5);

cout << "\nList 1 (gqlist1) is : ";
showlist(gqlist1);

cout << "\nList 2 (gqlist2) is : ";
showlist(gqlist2);

cout << "\ngqlist1.front() : "<< gqlist1.front();
cout << "\ngqlist1.back() : "<< gqlist1.back();

cout << "\ngqlist1.pop_front() : ";
gqlist1.pop_front();
showlist(gqlist1);

gqlist1.insert(gqlist2.begin(), 2, 10); // insert 2 tens
//gqlist1.insert(gqlist1.begin()+5, 100);  // iterator can't do it
list <int>::iterator it = gqlist1.begin();
advance(it,5);
gqlist1.insert(it, 100);
showlist(gqlist1);

cout << "\ngqlist2.pop_back() : ";
gqlist2.pop_back();
showlist(gqlist2);

cout << "\ngqlist1.reverse() : ";
gqlist1.reverse();
showlist(gqlist1);
```

```
cout << "\ngqlist2.sort(): ";
gqlist2.sort();
showlist(gqlist2);

cout << "\ngqlist1.splice with list2: ";
list <int>::iterator ind = gqlist2.begin();
advance(ind,3);
gqlist1.splice(gqlist1.begin(),gqlist2, gqlist2.begin(), ind);
showlist(gqlist1);
cout << "\n list2after splice: ";
showlist(gqlist2);

cout << "\ngqlist1.unique: ";
gqlist1.unique(); // delete repeated values
showlist(gqlist1);

gqlist1.sort();
gqlist1.unique();
for(list<int>::iterator it = gqlist1.begin();it!=gqlist1.end();++it){ (*it)++;}
cout<<"qlist1 update: ";
showlist(gqlist1);

gqlist1.merge(gqlist2);
cout<<"qlist1 & qlist2 merged: ";
showlist(gqlist1);

gqlist1.remove_if(is_even);
cout<<"removed even elements";
showlist(gqlist1);
}
```

Результат:

```
List 1 (gqlist1) is : 0,2,4,6,8,10,12,14
List 2 (gqlist2) is : 1,2,5,3,4
gqlist1.front() : 0
gqlist1.back() : 14
gqlist1.pop_front() : 2,4,6,8,10,12,14
2,4,6,8,10,100,12,14
gqlist2.pop_back() : 10,10,1,2,5,3
gqlist1.reverse() : 14,12,100,10,8,6,4,2
```

gqlist2.sort(): 1,2,3,5,10,10
gqlist1.splice with list2: 1,2,3,14,12,100,10,8,6,4,2
list2after splice: 5,10,10
gqlist1.unique: 1,2,3,14,12,100,10,8,6,4,2
qlist1 update: 2,3,4,5,7,9,11,13,15,101
qlist1 & qlist2 merged: 2,3,4,5,5,7,9,10,10,11,13,15,101
removed even elements3,5,5,7,9,11,13,15,101