# STL for newbies

## MỤC LỤC

# Lời nói đầu

- "C++ được đánh giá là ngôn ngữ mạnh vì tính mềm dẻo, gần gũi với ngôn ngữ máy. Ngoài ra, với khả năng lập trình theo mẫu ( template ), C++ đã khiến ngôn ngữ lập trình trở thành khái quát, không cụ thể và chi tiết như nhiều ngôn ngữ khác. Sức mạnh của C++ đến từ STL, viết tắt của Standard Template Library - một thư viện template cho C++ với những cấu trúc dữ liệu cũng như giải thuật được xây dựng tổng quát mà vẫn tận dụng được hiệu năng và tốc độ của C. Với khái niệm template, những người lập trình đã đề ra khái niệm lập trình khái lược (generic programming), C++ được cung cấp kèm với bộ thư viện chuẩn STL.

  Bộ thư viện này thực hiện toàn bộ các công việc vào ra dữ liệu (iostream), quản lý mảng (vector), thực hiện hầu hết các tính năng của các cấu trúc dữ liệu cơ bản (stack, queue, map, set...). Ngoài ra, STL còn bao gồm các thuật toán cơ bản: tìm min, max, tính tổng, sắp xếp (với nhiều thuật toán khác nhau), thay thế các phần tử, tìm kiếm (tìm kiếm thường và tìm kiếm nhị phân), trộn. Toàn bộ các tính năng nêu trên đều được cung cấp dưới dạng template nên việc lập trình luôn thể hiện tính khái quát hóa cao. Nhờ vậy, STL làm cho ngôn ngữ C++ trở nên trong sáng hơn nhiều."

  Trích **"Tổng quan về thư viện chuẩn STL"**
- STL khá là rộng nên tài liệu này mình chỉ viết để định hướng về cách sử dụng STL cơ bản để các bạn ứng dụng trong việc giải các bài toán tin học đòi hỏi đến cấu trúc dữ liệu và giải thuật.
- Mình chủ yếu sử dụng các ví dụ, cũng như nguồn tài liệu từ trang web www.cplusplus.com , các bạn có thể tham khảo chi tiết ở đó nữa.
- Để có thể hiểu được những gì mình trình bày trong này, các bạn cần có những kiến thức về các cấu trúc dữ liệu, cũng như một số thuật toán như sắp xếp, tìm kiếm...
- Việc sử dụng thành thạo STL sẽ là rất quan trọng nếu các bạn có ý định tham gia các kì thi như Olympic Tin Học, hay ACM. "STL sẽ nối dài khả năng lập trình của các bạn" (trích lời thầy Lê Minh Hoàng).
- Mọi ý kiến đóng góp xin gửi về địa chỉ: manhdjeu@gmail.com

- Thư viện mẫu chuẩn STL trong C++ chia làm 4 thành phần là:
    - Containers Library : chứa các cấu trúc dữ liệu mẫu (template)
        - Sequence containers
            - Vector
            - Deque
            - List
        - Containers adpators
            - Stack
            - Queue
            - Priority_queue
        - Associative containers
            - Set
            - Multiset
            - Map
            - Multimap
            - Bitset
    - Algorithms Library: một số thuật toán để thao tác trên dữ liệu
    - Iterator Library: giống như con trỏ, dùng để truy cập đến các phần tử dữ liệu của container.
    - Numeric library:

- Để sử dụng STL, bạn cần khai báo từ khóa "using namespace std;" sau các khai báo thư viện (các "#include", hay "#define",...)
- Ví dụ:

```
#include <iostream>
#include <stack>  //khai báo sử dụng container stack
#define n 100
using namespace std;  //khai báo sử dụng STL
main() {
      ....
}
```

- Việc sử dụng các hàm trong STL tương tự như việc sử dụng các hàm như trong class. Các bạn đọc qua một vài ví dụ là có thể thấy được quy luật.

# I. ITERATOR (BIẾN LẶP):

- Trong C++, một biến lặp là một đối tượng bất kì, trỏ tới một số phần tử trong 1 phạm vi của các phần tử (như mảng hoặc container), có khả năng để lặp các phần tử trong phạm vi bằng cách sử dụng một tập các toán tử (operators) (như so sánh, tăng (++), ...)
- Dạng rõ ràng nhất của iterator là một con trỏ: Một con trỏ có thể trỏ tới các phần tử trong mảng, và có thể lặp thông qua sử dụng toán tử tăng (++). Tuy nhiên, cũng có

các dạng khác của iterator. Ví dụ: mỗi loại container (chẳng hạn như vector) có một loại iterator được thiết kế để lặp các phần tử của nó một cách hiệu quả.

- Iterator có các toán tử như:
  - So sánh: "==" , "!=" giữa 2 iterator.
  - Gán: "=" giữa 2 iterator.
  - Cộng trừ: "+","-" với hằng số và "++","—".
  - Lấy giá trị: "*".

## II. CONTAINERS (THƯ VIỆN LƯU TRỮ)

- Một container là một đối tượng cụ thể lưu trữ một tập các đối tượng khác (các phần tử của nó). Nó được thực hiện như các lớp mẫu ( class templates).
- Container quản lý không gian lưu trữ cho các phần tử của nó và cung cấp các hàm thành viên (member function) để truy cập tới chúng, hoặc trực tiếp hoặc thông qua các biến lặp (iterator – giống như con trỏ).
- Container xây dựng các cấu trúc thường sử dụng trong lập trình như: mảng động - dynamic arrays (vector), hàng đợi – queues (queue), hàng đợi ưu tiên – heaps (priority queue), danh sách kiên kết – linked list (list), cây – trees (set), mảng ánh xạ - associative arrays (map),...
- Nhiều container chứa một số hàm thành viên giống nhau. Quyết định sử dụng loại container nào cho nhu cầu cụ thể nói chung không chỉ phụ thuộc vào các hàm được cung cấp mà còn phải dựa vào hiệu quả của các hàm thành viên của nó (độ phức tạp (từ giờ mình sẽ viết tắt là ĐPT) của các hàm). Điều này đặc biệt đúng với container dãy (sequence containers), mà trong đó có sự khác nhau về độ phức tạp đối với các thao tác chèn/xóa phần tử hay truy cập vào phần tử.

### 1. Iterator:

Tất cả các container ở 2 loại: Sequence container và Associative container đều hỗ trợ các iterator như sau (ví dụ với vector, những loại khác có chức năng cũng vậy).

```
/*khai báo iterator "it"*/
vector <int> :: iterator it;
/* trỏ đến vị trí phần tử đầu tiên của vector */
it=vector.begin();
/*trỏ đến vị trí kết thúc (không phải phần tử cuối cùng nhé) của vector) */
it=vector.end();
/* khai báo iterator ngược "rit" */
vector <int> :: reverse_iterator rit; rit = vector.rbegin();
/* trỏ đến vị trí kết thúc của vector theo chiều ngược (không phải phần tử đầu tiên nhé*/
rit = vector.rend();
```

Tất cả các hàm iterator này đều có độ phức tạp O(1).

## 2. Vector (Mảng động):

**Khai báo vector:**

```
#include <vector>
...
/* Vector 1 chiều */

/* tạo vector rỗng kiểu dữ liệu int */
vector <int> first;

//tạo vector với 4 phần tử là 100
vector <int> second (4,100);

// lấy từ đầu đến cuối vector second
vector <int> third (second.begin(),second.end())

//copy từ vector third
vector <int> four (third)

/*Vector 2 chiều*/

/* Tạo vector 2 chiều rỗng */
vector < vector <int> > v;

/* khai báo vector 5×10 */
vector < vector <int> > v (5, 10) ;

/* khai báo 5 vector 1 chiều rỗng  */
vector < vector <int> > v (5) ;

//khai báo vector 5*10 với các phần tử khởi tạo giá trị là 1
vector < vector <int> > v (5, vector <int> (10,1) ) ;
```

Các bạn chú ý 2 dấu "ngoặc" không được viết liền nhau.
Ví dụ như sau là **sai**:

```
/*Khai báo vector 2 chiều SAI*/
vector <vector <int>> v;
```

**Các hàm thành viên:**

**Capacity:**

- size : trả về số lượng phần tử của vector. ĐPT O(1).
- empty : trả về true(1) nếu vector rỗng, ngược lại là false(0). ĐPT O(1).

**Truy cập tới phần tử**:

- operator [] : trả về giá trị phần tử thứ []. ĐPT O(1).
- at : tương tự như trên. ĐPT O(1).

- front: trả về giá trị phần tử đầu tiên. ĐPT O(1).
- back: trả về giá trị phần tử cuối cùng. ĐPT O(1).

**Chỉnh sửa:**

- push_back : thêm vào ở cuối vector. ĐPT O(1).
- pop_back : loại bỏ phần tử ở cuối vector. ĐPT O(1).
- insert (iterator,x): chèn "x" vào trước vị trí "iterator" ( x có thể là phần tử hay iterator của 1 đoạn phần tử…). ĐPT O(n).
- erase : xóa phần tử ở vị trí iterator. ĐPT O(n).
- swap : đổi 2 vector cho nhau (ví dụ: first.swap(second);). ĐPT O(1).
- clear: xóa vector. ĐPT O(n).

**Nhận xét:**

- Sử dụng vector sẽ tốt khi:
  - Truy cập đến phần tử riêng lẻ thông qua vị trí của nó O(1)
  - Chèn hay xóa ở vị trí cuối cùng O(1).
- Vector làm việc giống như một "mảng động".

Ví dụ 1: Ví dụ này chủ yếu để làm quen sử dụng các hàm chứ không có đề bài cụ thể.

```
#include <iostream>
#include <vector>
using namespace std;
vector <int> v; //Khai báo vector
vector <int>::iterator it;  //Khai báo iterator
vector <int>::reverse_iterator rit; //Khai báo iterator ngược
int i;
main() {
	for (i=1;i<=5;i++) v.push_back(i); // v={1,2,3,4,5}
	cout << v.front() << endl;         // In ra 1
	cout << v.back() << endl;          // In ra 5

	cout << v.size() << endl;          // In ra 5

	v.push_back(9);                    // v={1,2,3,4,5,9}
	cout << v.size() << endl;          // In ra 6

	v.clear();                         // v={}
	cout << v.empty() << endl;         // In ra 1 (vector rỗng)

	for (i=1;i<=5;i++) v.push_back(i); // v={1,2,3,4,5}
	v.pop_back();                      // v={1,2,3,4}
	cout << v.size() << endl;          // In ra 4

	v.erase(v.begin()+1);              // Xóa ptử thứ 1 v={1,3,4}
	v.erase(v.begin(),v.begin()+2);    // v={4}
	v.insert(v.begin(),100);           // v={100,4}
	v.insert(v.end(),5);               // v={100,4,5}

	/*Duyệt theo chỉ số phần tử*/
	for (i=0;i<v.size();i++) cout << v[i] << " "; // 100 4 5
	cout << endl;
```

```
        /*Chú ý: Không nên viết
         for (i=0;i<=v.size()-1;i++) ...
         Vì nếu vector v rỗng thì sẽ dẫn đến sai khi duyệt !!!
         */

         /*Duyệt theo iterator*/
         for (it=v.begin();it!=v.end();it++)
             cout << *it << " " ;
         //In ra giá trị ma iterator đang trỏ tới "100 4 5"
         cout << endl;

         /*Duyệt iterator ngược*/
         for (rit=v.rbegin();rit!=v.rend();rit++)
             cout << *rit << " "; // 5 4 100
         cout << endl;

         system("pause");
}
```

Ví dụ 2: Cho đồ thị vô hướng G có n đỉnh (các đỉnh đánh số từ 1 đến n) và m cạnh và không có khuyên (đường đi từ 1 đỉnh tới chính đỉnh đó).
Cài đặt đồ thị bằng danh sách kề và in ra các cạnh kề đối với mỗi cạnh của đồ thị.
Dữ liệu vào:

- Dòng đầu chứa n và m cách nhau bởi dấu cách
- M dòng sau, mỗi dòng chứa u và v cho biết có đường đi từ u tới v. Không có cặp đỉnh u,v nào chỉ cùng 1 đường đi.

Dữ liệu ra:

- M dòng: Dòng thứ i chứa các đỉnh kề cạnh i theo thứ tự tăng dần và cách nhau bởi dấu cách.

Giới hạn: 1 <= n,m <= 10000
Ví dụ:

| INPUT | OUTPUT |
|---|---|
| 6 7 | 2 3 5 6 |
| 1 2 | 1 3 6 |
| 1 3 | 1 2 5 |
| 1 5 | |
| 2 3 | 1 3 |
| 2 6 | 1 2 |
| 3 5 | |
| 6 1 | |

Chương trình mẫu:

```
#include <iostream>
#include <vector>
using namespace std;
vector < vector <int> > a (10001);
```

```
//Khai báo vector 2 chiều với 10001 vector 1 chiều rỗng
int m,n,i,j,u,v;
main() {
	/*Input data*/
	cin >> n >> m;
	for (i=1;i<=m;i++) {
		cin >> u >> v;
		a[u].push_back(v);
		a[v].push_back(u);
	}
	/*Sort cạnh kề*/
	for (i=1;i<=m;i++)
		sort(a[i].begin(),a[i].end());
	/*Print Result*/
	for (i=1;i<=m;i++) {
		for (j=0;j<a[i].size();j++) cout << a[i][j] << " ";
		cout << endl;
	}
	system("pause");
}
```

### 3. Deque (Hàng đợi hai đầu):

- Deque (thường được phát âm giống như "deck") là từ viết tắt của double-ended queue (hàng đợi hai đầu).
- Deque có các ưu điểm như:
  - Các phần tử có thể truy cập thông cua chỉ số vị trí của nó. O(1)
  - Chèn hoặc xóa phần tử ở cuối hoặc đầu của dãy. O(1)

**Khai báo:** `#include <deque>`

**Capacity:**

- size : trả về số lượng phần tử của deque. ĐPT O(1).
- empty : trả về true(1) nếu deque rỗng, ngược lại là false(0). ĐPT O(1).

**Truy cập phần tử**:

- operator [] : trả về giá trị phần tử thứ []. ĐPT O(1).
- at : tương tự như trên. ĐPT O(1).
- front: trả về giá trị phần tử đầu tiên. ĐPT O(1).
- back: trả về giá trị phần tử cuối cùng. ĐPT O(1).

**Chỉnh sửa:**

- push_back : thêm phần tử vào ở cuối deque. ĐPT O(1).
- push_front : thêm phần tử vào đầu deque. ĐPT O(1).
- pop_back : loại bỏ phần tử ở cuối deque. ĐPT O(1).
- pop_front : loại bỏ phần tử ở đầu deque. ĐPT O(1).
- insert (iterator,x): chèn "x" vào trước vị trí "iterator" ( x có thể là phần tử hay iterator của 1 đoạn phần tử…). ĐPT O(n).
- erase : xóa phần tử ở vị trí iterator. ĐPT O(n).
- swap : đổi 2 deque cho nhau (ví dụ: first.swap(second);). ĐPT O(n).

clear: xóa vector. ĐPT O(1).

## 4. List (Danh sách liên kết):

- List được thực hiện như danh sách nối kép (doubly-linked list). Mỗi phần tử trong danh sách nối kép có liên kết đến một phần tử trước đó và một phần tử sau nó.
- Do đó, list có các ưu điểm như sau:
  - Chèn và loại bỏ phần tử ở bất cứ vị trí nào trong container. O(1).
- Điểm yếu của list là khả năng truy cập tới phần tử thông qua vị trí. O(n).
- Khai báo: #include <list>

**Các hàm thành viên:**

**Capacity:**
- size : trả về số lượng phần tử của list. ĐPT O(1).
- empty : trả về true(1) nếu list rỗng, ngược lại là false(0). ĐPT O(1).

**Truy cập phần tử**:
- front: trả về giá trị phần tử đầu tiên. ĐPT O(1).
- back: trả về giá trị phần tử cuối cùng. ĐPT O(1).

**Chỉnh sửa:**
- push_back : thêm phần tử vào ở cuối list. ĐPT O(1).
- push_front : thêm phần tử vào đầu list. ĐPT O(1).
- pop_back : loại bỏ phần tử ở cuối list. ĐPT O(1).
- pop_front : loại bỏ phần tử ở đầu list. ĐPT O(1).
- insert (iterator,x): chèn "x" vào trước vị trí "iterator" ( x có thể là phần tử hay iterator của 1 đoạn phần tử…). ĐPT là số phần tử thêm vào.
- erase : xóa phần tử ở vị trí iterator. ĐPT là số phần tử bị xóa đi.
- swap : đổi 2 list cho nhau (ví dụ: first.swap(second);). ĐPT O(1).
- clear: xóa list. ĐPT O(n).

**Operations:**
- splice : di chuyển phần tử từ list này sang list khác. ĐPT O(n).
- remove (const) : loại bỏ tất cả phần tử trong list bằng const. ĐPT O(n).
- remove_if (function) : loại bỏ tất các phần tử trong list nếu hàm function return true . ĐPT O(n).
- unique : loại bỏ các phần tử bị trùng lặp hoặc thỏa mãn hàm nào đó. ĐPT O(n). Lưu ý: Các phần tử trong list phải được sắp xếp.
- sort : sắp xếp các phần tử của list. O(NlogN)
- reverse : đảo ngược lại các phần tử của list. O(n).

## 5. Stack (Ngăn xếp):

- Stack là một loại container adaptor, được thiết kế để hoạt động theo kiểu LIFO (Last - in first - out) (vào sau ra trước), tức là một kiểu danh sách mà việc bổ sung và loại bỏ một phần tử được thực hiển ở cuối danh sách. Vị trí cuối cùng của stack gọi là đỉnh (top) của ngăn xếp.

Khai báo: `#include <stack>`

**Các hàm thành viên:**

- size : trả về kích thước hiện tại của stack. ĐPT O(1).
- empty : true stack nếu rỗng, và ngược lại. ĐPT O(1).
- push : đẩy phần tử vào stack. ĐPT O(1).
- pop : loại bỏ phần tử ở đỉnh của stack. ĐPT O(1).
- top : truy cập tới phần tử ở đỉnh stack. ĐPT O(1).

**Chương trình demo:**

```
#include <iostream>
#include <stack>
using namespace std;
stack <int> s;
int i;
main() {
	for (i=1;i<=5;i++) s.push(i); // s={1,2,3,4,5}
	s.push(100);                  // s={1,2,3,4,5,100}
	cout << s.top() << endl;      // In ra 100
	s.pop();                      // s={1,2,3,4,5}
	cout << s.empty() << endl;    // In ra 0
	cout << s.size() << endl;     // In ra 5
	system("pause");
}
```

## 6. Queue (Hàng đợi):

- Queue là một loại container adaptor, được thiết kế để hoạt động theo kiểu FIFO (First - in first - out) (vào trước ra trước), tức là một kiểu danh sách mà việc bổ sung được thực hiển ở cuối danh sách và loại bỏ ở đầu danh sách.
- Trong queue, có hai vị trí quan trọng là vị trí đầu danh sách (front), nơi phần tử được lấy ra, và vị trí cuối danh sách (back), nơi phần tử cuối cùng được thêm vào.

**Khai báo:** `#include <queue>`

**Các hàm thành viên:**

- size : trả về kích thước hiện tại của queue. ĐPT O(1).
- empty : true nếu queue rỗng, và ngược lại. ĐPT O(1).
- push : đẩy vào cuối queue. ĐPT O(1).
- pop: loại bỏ phần tử (ở đầu). ĐPT O(1).
- front : trả về phần tử ở đầu. ĐPT O(1).
- back: trả về phần tử ở cuối. ĐPT O(1).

**Chương trình demo:**

```
#include <iostream>
#include <queue>
using namespace std;
queue <int> q;
int i;
main() {
	for (i=1;i<=5;i++) q.push(i); // q={1,2,3,4,5}
	q.push(100);                  // q={1,2,3,4,5,100}
	cout << q.front() << endl;    // In ra 1
	q.pop();                      // q={2,3,4,5,100}
	cout << q.back() << endl;     // In ra 100
```

```
        cout << q.empty() << endl;     // In ra 0
        cout << q.size() << endl;      // In ra 5
        system("pause");
}
```

## 7. Priority Queue (Hàng đợi ưu tiên):

- Priority queue là một loại container adaptor, được thiết kế đặc biệt để phần tử ở đầu luôn luôn lớn nhất (theo một quy ước về độ ưu tiên nào đó) so với các phần tử khác.
- Nó giống như một heap, mà ở đây là heap max, tức là phần tử có độ ưu tiên lớn nhất có thể được lấy ra và các phần tử khác được chèn vào bất kì.
- Phép toán so sánh mặc định khi sử dụng priority queue là phép toán less (Xem thêm ở thư viện **functional**)
- Để sử dụng priority queue một cách hiệu quả, các bạn nên học cách viết hàm so sánh để sử dụng cho linh hoạt cho từng bài toán.
- Khai báo: `#include <queue>`

```
/*Dạng 1 (sử dụng phép toán mặc định là less)*/
priority_queue <int> pq;

/* Dạng 2 (sử dụng phép toán khác) */
priority_queue <int,vector<int>,greater<int> > q; //phép toán greater
```

Phép toán khác cũng có thể do người dùng tự định nghĩa. Ví dụ:
Cách khai báo ở dạng 1 tương đương với:

```
/* Dạng sử dụng class so sánh tự định nghĩa */
struct cmp{
       bool operator() (int a,int b) {return a<b;}
};


main() {
       …
       priority_queue <int,vector<int>,cmp > q;
}
```

**Các hàm thành viên:**

- size : trả về kích thước hiện tại của priority queue. ĐPT O(1)
- empty : true nếu priority queue rỗng, và ngược lại. ĐPT O(1).
- push : đẩy vào priority queue. ĐPT O(logN).
- pop: loại bỏ phần tử ở đỉnh priority queue. ĐPT O(logN).
- top : trả về phần tử ở đỉnh priority queue. ĐPT O(1).

**Chương trình Demo 1:**

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

main() {
```

```
        priority_queue <int> p;  // p={}
        p.push(1);               // p={1}
        p.push(5);               // p={1,5}
        cout << p.top() << endl; // In ra 5
        p.pop();                 // p={1}
        cout << p.top() << endl; // In ra 1
        p.push(9);               // p={1,9}
        cout << p.top() << endl; // In ra 9
        system("pause");
}
```

**Chương trình Demo 2:**

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

main() {
        priority_queue < int , vector <int> , greater <int> > p;  // p={}
        p.push(1);               // p={1}
        p.push(5);               // p={1,5}
        cout << p.top() << endl; // In ra 1
        p.pop();                 // p={5}
        cout << p.top() << endl; // In ra 5
        p.push(9);               // p={5,9}
        cout << p.top() << endl; // In ra 5
        system("pause");
}
```

**Chương trình Demo 3:**

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

struct cmp{
        bool operator() (int a,int b) {return a<b;}
};

main() {
        priority_queue < int , vector <int> , cmp > p;  // p={}
        p.push(1);               // p={1}
        p.push(5);               // p={1,5}
        cout << p.top() << endl; // In ra 1
        p.pop();                 // p={5}
        cout << p.top() << endl; // In ra 5
        p.push(9);               // p={5,9}
        cout << p.top() << endl; // In ra 5
        system("pause");
}
```

## 8. Set (Tập hợp):

- Set là một loại associative containers để lưu trữ các phần tử không bị trùng lặp (unique elements), và các phần tử này chính là các khóa (keys).

- Khi duyệt set theo iterator từ begin đến end, các phần tử của set sẽ tăng dần theo phép toán so sánh.
- Mặc định của set là sử dụng phép toán less, bạn cũng có thể viết lại hàm so sánh theo ý mình.
- Set được thực hiện giống như cây tìm kiếm nhị phân (Binary search tree).

**Khai báo:**

```
#include <set>
set <int> s;
set <int, greater<int> > s;
```

Hoặc viết class so sánh theo ý mình:

```
struct cmp{
	bool operator() (int a,int b) {return a<b;}
};
set <int,cmp > myset ;
```

**Capacity:**

- size : trả về kích thước hiện tại của set. ĐPT O(1)
- empty : true nếu set rỗng, và ngược lại. ĐPT O(1).

**Modifiers:**

- insert : Chèn phần tử vào set. ĐPT O(logN).
- erase : có 2 kiểu xóa: xóa theo iterator, hoặc là xóa theo khóa. ĐPT O(logN).
- clear : xóa tất cả set. ĐPT O(n).
- swap : đổi 2 set cho nhau. ĐPT O(n).

**Operations**:

- find : trả về itarator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy itarator trỏ về "end" của set. ĐPT O(logN).
- lower_bound : trả về iterator đến vị trí phần tử bé nhất mà không bé hơn (lớn hơn hoặc bằng) khóa (dĩ nhiên là theo phép so sánh), nếu không tìm thấy trả về vị trí "end" của set. ĐPT O(logN).
- upper_bound: trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về vị trí "end" của set.. ĐPT O(logN).
- count : trả về số lần xuất hiện của khóa trong container. Nhưng trong set, các phần tử chỉ xuất hiện một lần, nên hàm này có ý nghĩa là sẽ return 1 nếu khóa có trong container, và 0 nếu không có. ĐPT O(logN).

**Chương trình Demo 1:**

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

using namespace std;

main() {
	set <int> s;
	set <int> ::iterator it;
	s.insert(9);                // s={9}
	s.insert(5);                // s={5,9}
	cout << *s.begin() << endl; //In ra 5
	s.insert(1);                // s={1,5,9}
	cout << *s.begin() << endl; // In ra 1
```

```
        it=s.find(5);
        if (it==s.end()) cout << "Khong co trong container" << endl;
        else  cout << "Co trong container" << endl;

        s.erase(it);                // s={1,9}
        s.erase(1);                 // s={9}

        s.insert(3);                // s={3,9}
        s.insert(4);                // s={3,4,9}

        it=s.lower_bound(4);
        if (it==s.end()) cout << "Khong co phan tu nao trong set khong be hon 4" << endl;
        else cout << "Phan tu be nhat khong be hon 4 la " << *it << endl;  // In ra 4

        it=s.lower_bound(10);
        if (it==s.end()) cout << "Khong co phan tu nao trong set khong be hon 10" << endl;
        else cout << "Phan tu be nhat khong be hon 10 la " << *it << endl; // Khong co ptu nao

        it=s.upper_bound(4);
        if (it==s.end()) cout << "Khong co phan tu nao trong set lon hon 4" << endl;
        else cout << "Phan tu be nhat lon hon 4 la " << *it << endl;       // In ra 9

        /* Duyet set */

        for (it=s.begin();it!=s.end();it++) {
            cout << *it <<  " ";
        }
        // In ra 3 4 9

        cout << endl;
        system("pause");
}
```

**Lưu ý:** Nếu bạn muốn sử dụng hàm lower_bound hay upper_bound để tìm phần tử lớn nhất "bé hơn hoặc bằng" hoặc "bé hơn" bạn có thể thay đổi cách so sánh của set để tìm kiếm. Mời bạn xem chương trình sau để rõ hơn:

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

using namespace std;

main() {
        set <int, greater <int> > s;
        set <int, greater <int> > :: iterator it; // Phép toán so sánh là greater

        s.insert(1);                    // s={1}
        s.insert(2);                    // s={2,1}
        s.insert(4);                    // s={4,2,1}
        s.insert(9);                    // s={9,4,2,1}

        /* Tim phần tử lớn nhất bé hơn hoặc bằng 5 */
        it=s.lower_bound(5);
        cout << *it << endl;            // In ra 4

        /* Tim phần tử lớn nhất bé hơn 4 */
        it=s.upper_bound(4);
        cout << *it << endl;            // In ra 2

        system("pause");
}
```

## 9. Mutiset (Tập hợp):

- Multiset giống như Set nhưng có thể chứa các khóa có giá trị giống nhau.
- Khai báo : giống như set.
- **Các hàm thành viên:**

**Capacity**:

- size : trả về kích thước hiện tại của multiset. ĐPT O(1)
- empty : true nếu multiset rỗng, và ngược lại. ĐPT O(1).

**Chỉnh sửa:**

- insert : Chèn phần tử vào set. ĐPT O(logN).
- erase :
  - xóa theo iterator ĐPT O(logN)
  - xóa theo khóa: xóa tất cả các phần tử bằng khóa trong multiset ĐPT: O(logN) + số phần tử bị xóa.
- clear : xóa tất cả set. ĐPT O(n).
- swap : đổi 2 set cho nhau. ĐPT O(n).

**Operations**:

- find : trả về itarator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy itarator trỏ về "end" của set. ĐPT O(logN). Dù trong multiset có nhiều phần tử bằng khóa thì nó cũng chỉ iterator đến một phần tử.
- lower_bound : trả về iterator đến vị trí phần tử bé nhất mà không bé hơn (lớn hơn hoặc bằng) khóa (dĩ nhiên là theo phép so sánh), nếu không tìm thấy trả về vị trí "end" của set. ĐPT O(logN).
- upper_bound: trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về vị trí "end" của set.. ĐPT O(logN).
- count : trả về số lần xuất hiện của khóa trong multiset. ĐPT O(logN) + số phần tử tìm được.

**Chương trình Demo:**

```
#include <iostream>
#include <set>
using namespace std;
main() {
	multiset <int> s;
	multiset <int> :: iterator it;
	int i;
	for (i=1;i<=5;i++) s.insert(i*10); // s={10,20,30,40,50}
	s.insert(30);                      // s={10,20,30,30,40,50}
	cout << s.count(30) << endl;       // In ra 2
	cout << s.count(20) << endl;       // In ra 1
	s.erase(30);                       // s={10,20,40,50}

	/* Duyet set */

	for (it=s.begin();it!=s.end();it++) {
	    cout << *it <<  " ";
	}
	// In ra 10 20 40 50
	cout << endl;
	system("pause");
}
```

## 10. Map (Ánh xạ):

- Map là một loại associative container. Mỗi phần tử của map là sự kết hợp của khóa (key value) và ánh xạ của nó (mapped value). Cũng giống như set, trong map không chứa các khóa mang giá trị giống nhau.
- Trong map, các khóa được sử dụng để xác định giá trị các phần tử. Kiểu của khóa và ánh xạ có thể khác nhau.
- Và cũng giống như set, các phần tử trong map được sắp xếp theo một trình tự nào đó theo cách so sánh.
- Map được cài đặt bằng red-black tree (cây đỏ đen) – một loại cây tìm kiếm nhị phân tự cân bằng. Mỗi phần tử của map lại được cài đặt theo kiểu pair (xem thêm ở thư viện **utility**).

**Khai báo:**

```
#include <map>
...
map <kiểu_dữ_liệu_1,kiểu_dữ_liệu_2>
// kiểu dữ liệu 1 là khóa, kiểu dữ liệu 2 là giá trị của khóa.
```

Sử dụng class so sánh:

Dạng 1:

```
struct cmp{
	bool operator() (char a,char b) {return a<b;}
};
.....
map <char,int,cmp> m;
```

- Truy cập đến giá trị của các phần tử trong map khi sử dụng iterator:

Ví dụ ta đang có một iterator là it khai báo cho map thì:

```
(*it).first;    // Lấy giá trị của khóa, kiểu_dữ_liệu_1
(*it).second;   // Lấy giá trị của giá trị của khóa, kiểu_dữ_liệu_2
(*it)          // Lấy giá trị của phần tử mà iterator đang trỏ đến, kiểu pair

it->first;      // giống như (*it).first
it->second;     // giống như (*it).second
```

**Capacity**:

- size : trả về kích thước hiện tại của map. ĐPT O(1)
- empty : true nếu map rỗng, và ngược lại. ĐPT O(1).

**Truy cập tới phần tử:**

- operator [khóa]: Nếu khóa đã có trong map, thì hàm này sẽ trả về giá trị mà khóa ánh xạ đến. Ngược lại, nếu khóa chưa có trong map, thì khi gọi [] nó sẽ thêm vào map khóa đó. ĐPT O(logN)

**Chỉnh sửa**

- insert : Chèn phần tử vào map. Chú ý: phần tử chèn vào phải ở kiểu "pair". ĐPT O(logN).
- erase :
  - xóa theo iterator ĐPT O(logN)
  - xóa theo khóa: xóa khóa trong map. ĐPT: O(logN).
- clear : xóa tất cả set. ĐPT O(n).

- swap : đổi 2 set cho nhau. ĐPT O(n).

**Operations**:

- find : trả về itarator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy iterator trỏ về "end" của map. ĐPT O(logN).
- lower_bound : trả về iterator đến vị trí phần tử bé nhất mà không bé hơn (lớn hơn hoặc bằng) khóa (dĩ nhiên là theo phép so sánh), nếu không tìm thấy trả về vị trí "end" của map. ĐPT O(logN).
- upper_bound: trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về vị trí "end" của map. ĐPT O(logN).
- count : trả về số lần xuất hiện của khóa trong multiset. ĐPT O(logN)

**Chương trình demo:**

```
#include <iostream>
#include <map>
#include <vector>
using namespace std;
main() {
	map <char,int> m;
	map <char,int> :: iterator it;

	m['a']=1;                              // m={{'a',1}}
	m.insert(make_pair('b',2));            // m={{'a',1};{'b',2}}
	m.insert(pair<char,int>('c',3) );      // m={{'a',1};{'b',2};{'c',3}}

	cout << m['b'] << endl;                // In ra 2
	m['b']++;                              // m={{'a',1};{'b',3};{'c',3}}

	it=m.find('c');                        // it point to key 'c'

	cout << it->first << endl;             // In ra 'c'
	cout << it->second << endl;            // In ra 3

	m['e']=100;                           //m={{'a',1};{'b',3};{'c',3};{'e',100}}

	it=m.lower_bound('d');                 // it point to 'e'
	cout << it->first << endl;             // In ra 'e'
	cout << it->second << endl;            // In ra 100

	system("pause");
}
```

## 11. Multi Map (Ánh xạ):

Giống như map nhưng có thể chứa các phần tử có khóa giống nhau, do đó nó khác map ở chỗ không có opearator[].

# III. STL ALGORITHMS (THƯ VIỆN THUẬT TOÁN):

- Khai báo sử dụng: `#include <algorithm>`
- Các hàm trong STL Algorithm khá nhiều nên mình chỉ giới thiệu sơ qua về một số hàm hay sử dụng trong các bài toán.
- Có một lưu ý nhỏ cho các bạn là khi sử dụng các hàm mà thực hiện trong một đoạn phần tử liên tiếp nào đó thì các hàm trong c++ thường có tác dụng trên nửa đoạn [..). Ví dụ như: bạn muốn hàm f có tác dụng trong đoạn từ 1→n thì các bạn phải gọi hàm trong đoạn từ 1 → n+1.

## 1. Min, max:

**1.1.** **min:** trả về giá trị bé hơn theo phép so sánh (mặc định là phép toán less):

Ví dụ: min('a','b') sẽ return 'a';
min(3,1) sẽ return 1;

**1.2.** **max** thì ngược lại với hàm min:

Ví dụ: max('a','b') sẽ return 'b'
max(3,1) sẽ return 1.

**1.3.** **next_permutation:** hoán vị tiếp theo. Hàm này sẽ return 1 nếu có hoán vị tiếp theo, 0 nếu không có hoán vị tiếp theo.

Ví dụ:

```
// next_permutation
#include <iostream>
#include <algorithm>
using namespace std;

int main () {
  int myints[] = {1,2,3};

  cout << "The 3! possible permutations with 3 elements:\n";

  do {
    cout << myints[0] << " " << myints[1] << " " << myints[2] << endl;
  } while ( next_permutation (myints,myints+3) );

  return 0;
}
```

Output:

```
The 3! possible permutations with 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3  1 2
4  2 1
```

**1.4.** **prev_permution:** ngược lại với next_permutation

## 2. Sắp xếp:

- sort: sắp xếp đoạn phần tử theo một trình tự nào đó. Mặc định của sort là sử dụng operator <.
- Bạn có thể sử dụng hàm so sánh, hay class so sánh tự định nghĩa để sắp xếp cho linh hoạt.
- Chương trình mẫu:

```
// sort algorithm example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i>j);}
} myobject;

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};

  // using default comparison (operator <):
  sort (myints, myints+4);                          //(12 32 45 71)26 80 53 33

  // using function as comp
  sort (myints+4, myints+8, myfunction);            // 12 32 45 71(26 33 53 80)

  for (int i=0;i<8;i++) cout << myints[i] << " ";
  cout << endl;

  // using object as comp
  sort (myints, myints+8, myobject);                //(80 71 53 45 33 32 26 12)

  for (int i=0;i<8;i++) cout << myints[i] << " ";
  cout << endl;

  system("pause");

  return 0;
}
```

## 3. Tìm kiếm nhị phân (các hàm đối với đoạn đã sắp xếp):

### 3.1. binary_search:

- tìm kiếm xem khóa có trong đoạn cần tìm không. Lưu ý: đoạn tìm kiếm phải được sắp xếp theo một trật tự nhất đinh. Nếu tìm được sẽ return true, ngược lại return false.
- Dạng 1: binary_search(vị trí bắt đầu, vị trí kết thúc, khóa);
- Dạng 2: binary_search(vị trí bắt đầu, vị trí kết thúc, khóa, phép so sánh);

```
// binary_search example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool myfunction (int i,int j) { return (i<j); }

int main () {
  int myints[] = {1,2,3,4,5,4,3,2,1};
  vector<int> v(myints,myints+9);                         // 1 2 3 4 5 4 3 2 1

  // Sử dụng toán tử so sánh mặc định
  sort (v.begin(), v.end());

  cout << "looking for a 3... ";
  if (binary_search (v.begin(), v.end(), 3))
    cout << "found!\n"; else cout << "not found.\n";

  // sử dụng hàm so sánh tự định nghĩa:
  sort (v.begin(), v.end(), myfunction);

  cout << "looking for a 6... ";
  if (binary_search (v.begin(), v.end(), 6, myfunction))
    cout << "found!\n"; else cout << "not found.\n";

  return 0;
}
```

### 3.2. lower_bound:

- Hàm lower_bound và upper_bound tương tự như ở trong container set hay map.
- Trả về iterator đến phần tử đầu tiên trong nửa đoạn [first,last] mà không bé hơn khóa tìm kiếm.
- Dạng 1: lower_bound( đầu , cuối , khóa );
- Dạng 2: lower_bound( đầu , cuối , khóa , phép toán so sánh của đoạn)

### 3.3. upper_bound

- Trả về iterator đến phần tử đầu tiên trong nửa đoạn [first,last] mà lớn hơn khóa tìm kiếm.
- Dạng 1: upper_bound ( đầu , cuối , khóa );
- Dạng 2: upper_bound ( đầu , cuối , khóa, phép toán so sánh của đoạn)
- Chương trình demo:

```
// lower_bound/upper_bound example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20
  vector<int>::iterator low,up;
```

```
  sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30

  low=lower_bound (v.begin(), v.end(), 20); //          ^
  up= upper_bound (v.begin(), v.end(), 20); //                   ^

  cout << "lower_bound at position " << int(low- v.begin()) << endl;
  cout << "upper_bound at position " << int(up - v.begin()) << endl;

  return 0;
}
```

**Output:**

```
lower_bound at position 3
upper_bound at position 6
```

## IV. THƯ VIỆN STRING C++:

- String là một kiểu đặc biệt của container, thiết kế đặc để hoạt động với các chuỗi kí tự.
- Khai báo : `#include <string>`
- Iterator: Tương tự như trong container, string cũng hỗ trợ các iterator như begin, end, rbegin, rend với ý nghĩa như trước.
- Nhập, xuất string:
  - Sử dụng toán tử >> : tương tự như trong C. Nhập đến khi gặp dấu cách.
  - Sử dụng getline: giống như gets trong C. Nhập cả một dòng kí tự (chứa cả dấu cách nếu có) cho string.
  - Xuất string: sử dụng toán tử << như bình thường.

Các hàm thành viên:

- Trong string bạn có thể sử dụng các toán tử "=" để gán giá trị cho string, hay toán tử "+" để nối hai string với nhau, hay toán tử "==","!=" để so sánh. Chức năng này khá giống với xâu ở trong ngôn ngữ pascal.
- Capacity:
  - size: trả về độ dài của string
  - length: trả về độ dài của string
  - clear : xóa string
  - empty: return true nếu string rỗng, false nếu ngược lại
- Truy cập đến phần tử:
  - operator [chỉ_số]: lấy kí tự vị trí chỉ_số của string
- Chỉnh sửa:
  - push_back: chèn kí tự vào sau string
  - insert (n,x): chèn x vào string ở trước vị trí thứ n. x có thể là string, char,...
  - erase (pos,n): xóa khỏi string "n" kí tự bắt đầu từ vị trí thứ "pos".
  - erase (iterator): xóa khỏi string phần tử ở vị trí iterator.
  - replace (pos, size, s1) : thay thế string từ vị trí "pos", số phần tử thay thế là "size" và thay bằng xâu s1.

- swap (string_cần_đổi): đổi giá trị 2 xâu cho nhau.

- String operations:
  - c_str : chuyển xâu từ dạng string trong C++ sang string trong C.
  - substr (pos,length): return string được trích ra từ vị trí thứ “pos”, và trích ra “length” kí tự.