Standard Template Library là một tập các lớp (**classes**) cung cấp cho lập trình viên những khuôn mẫu về tổ chức dữ liệu, thuật toán, bộ lặp. Nó cung cấp rất nhiều thuật toán cơ bản và cấu trúc dữ liệu cơ bản trong ngành khoa học máy tính.

STL có thể chia thành 3 phần:

- Container (Bộ chứa)
- Iterator (Bộ lặp)
- Algorithm (Thuật toán)

Mỗi thành phần của STL đảm nhiệm một số chức năng có liên quan đến nhau. Các thành phần này đều được xây dựng sẵn, chúng ta không cần định nghĩa lại mà chỉ cần sử dụng trong chương trình.

1. Container (Bộ chứa)

1.1 Đặt vấn đề

Thư viện khuôn hình chuẩn STL đưa ra các lớp bộ chứa (container) với vai trò như các cấu trúc dữ liệu được sử dụng để tổ chức lưu trữ dữ liệu cho chương trình. Nó cho phép lưu trữ một tập các phần tử theo một số hình thức khác nhau như lưu trữ tuần tự, lưu trữ dựa trên giá trị khóa.
Giả sử cần phải lưu trữ một tập các số nguyên, theo cách làm quen thuộc, ta khai báo như sau:

```
#define MAX 100
int int_array[MAX];
```

hoặc:

```
int* int_array;
```

Cách lưu trữ thứ nhất, kích thước mảng bị giới hạn, ngoài ra nó còn gây lãng phí bộ nhớ. Theo cách thứ hai, người dùng phải tự lo việc cấp phát và giải phóng bộ nhớ mỗi khi một phần tử được thêm vào hay lấy ra khỏi mảng. Điều này không hề đơn giản bởi lẽ việc cấp phát hay giải phóng không tốt có thể gây

lỗi chương trình hoặc "xả rác" trong bộ nhớ. Khi sử dụng bộ chứa STL, người dùng không cần bận tâm tới điều này. Ví dụ:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
 vector<int> int_array;
 int_array.push_back(1); // Thêm 1 vào vector
 int a = int_array[0]; // Gán cho a giá trị đầu tiên trong array
}
```

Các lớp bộ chứa được phân thành 2 nhóm chính là **các bộ chứa toàn tự (sequence container)** và **các bộ chứa liên kết (associative container)**. Các bộ chứa tuần tự bao gồm `vector, deque, list`, còn các bộ chứa liên kết tiêu biểu bao gồm `set, map, multiset, multimap`. Ngoài ra thư viện STL cũng cung cấp một số cấu trúc khác mà có thể xem là các bộ chứa như `string, bit_set`.

1.2 Lớp các bộ chứa tuần tự

1.2.1 vector

Lớp vector là một bộ chứa được dùng khá phổ biến. Nó được sử dụng để lưu trữ một dãy các phần tử. Lớp vector không chỉ cho phép người dùng truy xuất tới các phần tử dựa trên chỉ số như đối với cấu trúc mảng mà nó còn có khả năng tự động mở rộng khi nhu cầu người dùng vượt quá kích thước tối đa hiện tại. Điều này thuận tiện hơn khi sử dụng mảng.

Khai báo vector:

```
/* Vector 1 chiều */
/* Tạo vector rỗng kiểu int */
vector<int> first;
/* Tạo vector với 4 phần tử là 100 */
vector<int> second(4,100);
/* Tạo vector lấy từ phần tử đầu đến cuối của vector second */
vector<int> third(second.begin(), second.end());
/* Tạo vector copy từ vector third */
```

```
 vector<int> four(third);

 /* Vector 2 chiều */
 /* Tạo vector 2 chiều rỗng */
 vector<vector<int>> v1;
 /* Khai báo 5 vector 1 chiều rỗng */
 vector<vector<int>> v2(5);
 /* Tạo vector 5x10 với giá trị khởi tạo là 1 */
 vector<vector<int>> v3(5, vector<int>(10,1));
}
```

Capacity:

- `size`: Trả về số lượng phần tử vector.
- `max_size`: Trả về sức chứa tối đa của vector. Là kích thước tiềm năng tối đa mà vùng chứa có thể đạt.
- `resize`: Thay đổi kích thước vector.
- `capacity`: Trả về dung lượng lưu trữ được phân bổ.
- `empty`: Trả về True nếu vector rỗng, ngược lại trả về false.
- `reverse`: Cấp phát đủ vùng nhớ cho n phần tử.

Ví dụ:

```
// vector::reserve
#include <iostream>
#include <vector>

int main ()
{
 std::vector<int>::size_type sz;

 std::vector<int> foo;
 sz = foo.capacity();
 std::cout << "making foo grow:\n";
 for (int i=0; i<100; ++i) {
   foo.push_back(i);
   if (sz!=foo.capacity()) {
     sz = foo.capacity();
     std::cout << "capacity changed: " << sz << '\n';
   }
```

```
 }

 std::vector<int> bar;
 sz = bar.capacity();
 bar.reserve(100);   // this is the only difference with foo above
 std::cout << "making bar grow:\n";
 for (int i=0; i<100; ++i) {
   bar.push_back(i);
   if (sz!=bar.capacity()) {
     sz = bar.capacity();
     std::cout << "capacity changed: " << sz << '\n';
   }
 }
 return 0;
}
```

Truy cập tới phần tử:

- `[ ]`: Trả về giá trị phần tử thứ i trong vector (như truy xuất mảng).
- `at`: Tương tự như trên.
- `front`: Trả về giá trị phần tử đầu tiên.
- `back`: Trả về giá trị phần tử cuối cùng.

Các hàm bổ trợ:

- `push_back`: Thêm phần tử vào cuối vector.
- `pop_back`: Loại bỏ phần tử ở cuối vector.
- `swap`: Hoán đổi 2 vector cho nhau. Ví dụ first.swap(second)
- `clear`: Xóa vector.
- `insert(iterator, x)`: Chèn x vào trước vị trí iterator (x có thể là 1 phần tử hay iterator của 1 đoạn phần tử).
- `erase(iterator)`: Xóa phần tử ở vị trí iterator.

Ví dụ:

```
#include <iostream>
#include <vector>
#include <iterator>
```

```
using namespace std;
int main ()
{
 std::vector<int> vecInt32;
 std::vector<int>::iterator iter;

 //iter point to the begin of vecInt32
 iter = vecInt32.begin();
 //insert new element into vecInt32's container
 vecInt32.insert(iter, 10);
 //container = { 10 }

 iter = vecInt32.end();
 vecInt32.insert(iter, 20);
 //container = { 10, 20 }
 for(int i=0;i<vecInt32.size();i++)
   cout<<vecInt32[i]<<endl;

 iter = vecInt32.begin() + 1;
 vecInt32.insert(iter, 15);
 //container = { 10, 15, 20 }
 for(iter=vecInt32.begin();iter!=vecInt32.end();iter++)
   cout<<*iter<<endl;

  iter = vecInt32.begin() + 1;
 vecInt32.erase(iter);
 //container = { 10, 20 }
 copy(vecInt32.begin(), vecInt32.end(),
ostream_iterator<int>(cout,"\n") );
}
```

Nhận xét:

vector được thiết kế nhằm tối thiểu thời gian truy cập ngẫu nhiên. Do vậy các phần tử của vector được lưu trữ trong cùng một vùng nhớ theo trật tự tuyến tính. Tổ chức lưu trữ theo cách này đảm bảo việc truy nhập ngẫu nhiên với chi phí thời gian thấp nhất. Tuy nhiên, khi có yêu cầu cấp phát lại bộ nhớ, chương trình phải thực hiện khác nhiều thủ tuc. Các thủ tục này bao gồm:

1. Cấp phát lại một vùng nhớ rộng hơn. Thông thường vùng nhớ mới có kích thước gấp hai lần vùng nhớ hiện tại.

2. Sao chép toàn bộ các phần tử từ vùng nhớ cũ sang vùng nhớ mới nhờ cấu từ sao chép.

3. Xóa bỏ các đối tượng trên vùng nhớ cũ bằng cách gọi các hàm hủy từ.

4. Giải phóng vùng nhớ cũ.

Do phải thực hiện các thủ tục trên nên việc cấp phát lại của vector rất mất thời gian, đặc biệt là khi phần tử được lưu trữ phức tạp và số lượng phần tử của vector lớn. Chính vì lý do này, vector không được sử dụng trong các ứng dụng đòi hỏi nhiều thao tác chèn hay xóa các phần tử trong bộ chứa. Người dùng có thể tránh được điều này bằng cách sử dụng hàm reserve() để xin cấp phát trước một vùng nhớ nào đó. Tuy nhiên, không phải lúc nào ta cũng biết trước số phần tử sẽ được lưu trong bộ chứa.

vector cũng tạo ra một số vấn đề có liên quan tới việc cấp phát lại bộ nhớ. Do vector tổ chức lưu trữ các phần tử trong bộ nhớ theo trật tự tuyến tính nên bộ duyệt của vector chỉ đơn thuần là một con trỏ. Mặc dù các con trỏ này tối ưu tốc độ truy nhập nhưng nó cũng là nguyên nhân gây ra lỗi về truy nhập khi vector thực hiện cấp phát lại. Giả sử khởi ta khởi tạo một bộ duyệt của vector và để bộ duyệt này trỏ tới một phần tử nào đó, sau đó thực hiện việc cấp phát lại bộ nhớ. Khi đó, nếu lại sử dụng bộ duyệt trên để truy nhập tới vector thì rất có thể gây ra lỗi vì khi này các phần tử của vector đã được lưu trữ trên một vùng nhớ khác.

Ví dụ:

```
#include <iostream>
#include <vector>
#include <iterator>

using namespace std;
int main ()
{
 vector<int> v1(10, 9);
 ostream_iterator<int> out(cout, " ");
 copy(v1.begin(), v1.end(), out);
```

```
 vector<int>::iterator i=v1.begin();
 cout<<"\n i:"<< *i <<endl;
 *i = 47;
 copy(v1.begin(), v1.end(), out);
 // Bắt chương trình cấp phát lại bộ nhớ
 v1.resize(v1.capacity()+1);
 cout<<"\n i:"<< *i <<endl;

 *i=48;
 cout<<"\n i:"<< *i <<endl;
 copy(v1.begin(), v1.end(), out);
}
```

1.2.2 deque

deque là viết tắt của từ double-end queue nghĩa là một hàng đợi cho phép lấy và bổ sung dữ liệu từ hai đầu. Lớp deque trong STL không chỉ cho phép cập nhật các các phần tử của dữ liệu ở hai đầu mà còn cho phép bổ sung hay xóa đi một hoặc nhiều phần tử ở bất cứ vị trí nào. Tuy nhiên, các thao tác này đòi hỏi thời gian thực hiện tăng tuyến tính theo số phần tử, trong khi đó thời gian thực hiện cho các thao tác lấy và bổ sung dữ liệu hai đầu luôn là một hằng số.

Khai báo: `#include <deque>`

Capacity:

- `size`: trả về số lượng phần tử của deque.
- `empty`: trả về true nếu deque rỗng, ngược lại trả về false.

Truy cập phần tử:

- `[ ]`: trả về giá trị phần tử thứ [ ].
- `at`: tương tự như trên.
- `front`: trả về giá trị phần tử đầu tiên.
- `back`: trả về giá trị phần tử cuối cùng.

Chỉnh sửa:

- `push_back`: Thêm phần tử vào cuối deque.
- `push_front`: Thêm phần tử vào đầu deque.
- `pop_back`: Loại bỏ phần tử ở cuối deque,
- `pop_front`: Loại bỏ phần tử ở đầu deque.

- `insert(iterator, x)`: chèn x vào trước vị trí iterator (x có thể là phần tử hay iterator của 1 đoạn phần tử)
- `erase`: xóa phần tử ở vị trí iterator.
- `swap`: Đổi 2 deque cho nhau.

1.2.3 list

list là lớp của STL mô hình hóa cấu trúc dữ liệu danh sách liên kết hai chiều. Nó được đưa ra nhằm cải thiện tốc độ các ứng dụng sử dụng bộ chứa tuần tự có nhiều thao tác chèn, xóa các phần tử ở giữa bộ chứa.

list không được dùng cho các ứng dụng đòi hỏi các yêu cầu truy cập ngẫu nhiên. Việc truy nhập các phần tử trên list được các ứng dụng thực hiện theo cách duyệt từ đầu đến cuối danh sách hoặc ngược lại. Mặc dù thời gian để duyệt qua toàn bộ list lâu hơn nhiều so với vector hay deque có cùng số phần tử nhưng nó cũng gây không gây ra hiện tượng "thắt nút cổ chai" khi thực hiện chương trình nếu như bạn không thực hiện thủ tục này quá nhiều trong chương trình, Việc sử dụng list cũng như một bộ chứa tuần tự không có gì khác so với vector hay deque.

Khai báo: `#include <list>`

Capacity:

- `size`: Trả về số lượng phần tử của list.
- empty: Trả về true nếu list rỗng, ngược lại là false.

Truy cập phần tử:

- `push_back`: Thêm phần tử vào cuối list.
- `push_front`: Thêm phần tử vào đầu list.
- `pop_back`: Loại bỏ phần tử ở cuối list.
- `pop_front`: Loại bỏ phần tử ở đầu list.
- `insert(iterator, x)`: chèn x vào trước vị trí iterator (x có thể là phần tử hay iterator của 1 đoạn phần tử).
- `erase`: xóa phần tử ở vị trí iterator.
- `swap`: Đổi 2 list cho nhau.
- `clear`: Xóa list.

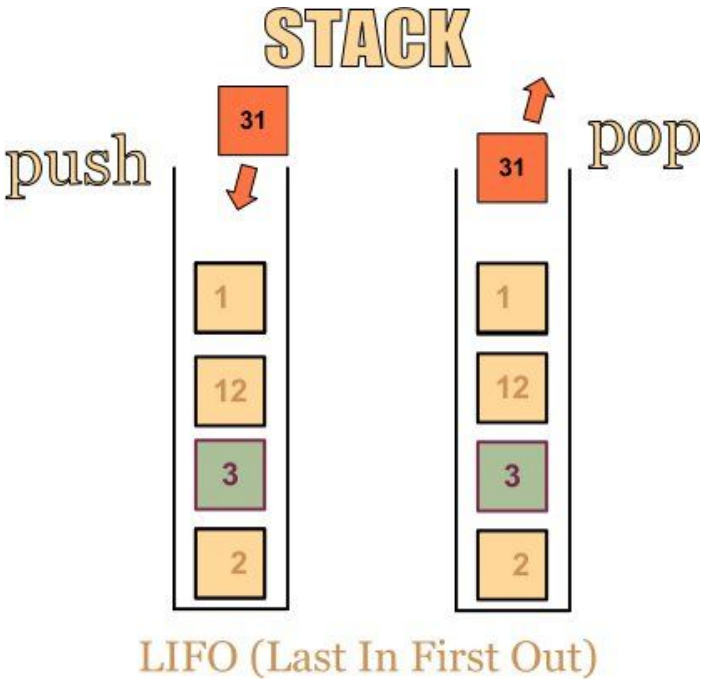
Operations:

- `splice`: Di chuyển phần tử từ list này sang list khác.
- `remove(const)`: Loại bỏ tất cả phần tử trong list bằng const.

- `remove_if(function)`: Loại bỏ tất cả phần tử trong list nếu hàm function return true.
- `unique`: Loại bỏ các phần tử bị trùng lặp hoặc thỏa mãn hàm nào đó. Lưu ý các phần tử trong list phải được sắp xếp.
- `sort`: Sắp xếp các phần tử trong list.
- `reverse`: Đảo ngược lại các phần tử của list.

1.2.4 stack

Stack là kiểu dữ liệu được thiết kế 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 hiện ở cuối danh sách. Vị trí cuối cùng của stack được gọi là đỉnh (top) của ngăn xếp.



Khai báo: `#include <stack>`

Các hàm thành viên:

- `size`: Trả về kích thước hiện tại của stack.
- `empty`: Trả về true nếu stack rỗng, và ngược lại.
- `push`: đẩy phần tử vào stack.
- `pop`: Loại bỏ phần tử ở đỉnh của stack.
- `top`: Truy cập tới phần tử ở đỉnh stack.

Ví dụ:

```
#include <iostream>
#include <stack>
using namespace std;
stack <int> s;
int i;
int main() {
 for (i=1;i<=5;i++) s.push(i); // Stack s = {1,2,3,4,5}
 s.push(100); // Đưa 100 và stack, 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; // False = 0
 cout << s.size() << endl; //  In ra 5
}
```

1.2.5 queue (Hàng đợi)

Queue được thiết kế theo kiểu FIFO (First in first out), tức là một kiểu danh sách mà việc bổ sung được thực hiện ở cuối danh sách và loại bỏ ở đầu danh sách. Trong queue, có 2 vị trí quan trọng là đầu danh sách (front), nơi phần tử được lấy ra, và vị trí cuối danh sách (back), nơi phần tử được thêm vào.

Khai báo: `#include <queue>`

Các hàm thành viên:

- `size`: Trả về kích thước hiện tại của queue.
- `empty`: Trả về true nếu queue rỗng, ngược lại trả về false.
- `push`: đẩy vào cuối queue.
- `pop`: Loại bỏ phần tử (ở đầu hàng đợi).
- `front`: Trả về phần tử ở đầu.
- `back`: Trả về phần tử ở cuối.

Ví dụ:

```
#include <iostream>
#include <queue>
using namespace std;
queue<int> q;
```

```
int main()
{
 for(int 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
}
```

1.2.6 priority queue (Hàng đợi ưu tiên)

- Được thiết kế đặc biệt để phần tử ở đầu luôn luôn lớn nhất (theo một quy ước nào đó) so với các phần tử khác.
- Phép toán mặc định khi sử dụng hàng đợi ưu tiên là phép toán less (thư viện functional)

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 cũng có thể do người dùng tự định nghĩa
```

Các hàm thành viên:

- `size`: Trả về kích thước hiện tại của priority queue.
- `empty`: True nếu priority queue rỗng, ngược lại trả về false.
- `push`: đẩy phần tử vào priority.
- `pop`: Loại bỏ phần tử ở đỉnh.
- `top`: Trả về phần tử ở đỉnh.

Ví dụ 1:

```
#include <iostream>
#include <queue>
```

```
using namespace std;
int 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);
 cout << p.top() << endl; // in ra 9
}
```

Ví dụ 2:

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
int 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}
 p.push(9); // p={5,9}
 cout << p.top() << endl; // In ra 5
}
```

Ví dụ 3:

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
struct cmp{
 bool operator() (int a, int b)
 {
   return a<b;
 }
```

```
};
int main() {
 priority_queue<int, vector<int>, cmp> p;
 p.push(1);
 p.push(5);
 cout << p.top() <<endl;
 p.pop();
 p.push(9);
 cout << p.top() << endl;
}
```

1.3 Lớp các bộ chứa liên kết

Các bộ chứa liên kết cũng được sử dụng để tổ chức lưu trữ dữ liệu. Ta sẽ sử dụng các bộ chứa liên kết để lưu trữ dữ liệu khi các thao tác truy xuất dữ liệu thường được thực hiện dựa trên giá trị khóa. Trong khi đó, ta sử dụng bộ chứa tuần tự để lưu trữ dữ liệu với các yêu cầu truy xuất được thực hiện tuần tự dựa trên thứ tự tuyến tính. Do vậy, dữ liệu được lưu trữ trong bộ chứa liên kết ở dạng đặc biệt, mỗi phần tử là một cặp gồm khóa và dữ liệu. Theo quan điểm toán học, mỗi đối tượng của các lớp bộ chứa liên kết sẽ biểu diễn một quan hệ giữa hai tập phần tử. Ý nghĩa của quan hệ này do người dùng quy định.

Thư viện STL cugn cấp bốn lớp bộ chứa liên kết cơ bản bao gồm: `set, map, multiset, multimap`. Trong đó, multiset và multimap là những mở rộng của hai lớp set và map.

1.3.1 Lớp set

- Set là một loại container để lưu trữ các phần tử không bị trùng lặp (unique element)m và các phần từ này chính là các khóa (key).
- 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, chúng ta 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> s1;
set<int, greater<int>> s2;
// Viết class so sánh theo ý mình
struct cmp{
 bool operator() (int a, int b) {
   return a<b;
 }
};
```

Capacity:

- `size`: Trả về kích thước hiện tại của set.
- `empty`: True nếu set rỗng, và ngược lại.

Modifiers:

- `insert`: Chèn phần tử vào set.
- `erase`: Xóa phần tử, xóa theo iterator hoặc xóa theo khóa.
- `clear`: Xóa tất cả set.
- `swap`: Hoán đổi 2 set cho nhau.

Operations:

- `find`: Trả về iterator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy iterator trỏ về "end" của set.
- `lower_bound`: Trả về iterator đến vị trí phần tử đầu tiên trong set mà lớn hơn hoặc bằng khóa. Nếu không tìm thấy trả về end của set.
- `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.
- `count`: Trả về số lần xuất 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.

Ví dụ 1:

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

```
using namespace std;
int main()
{
 set<int> s;
 set<int>::iterator it;
 s.insert(9); // s = {9}
 s.insert(5); // s = {5,9}
 copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "));// in ra
5 9
 s.insert(1); // s = {1,5,9}
 cout << *s.begin() <<endl; // in ra 1
 it = s.find(5);
 if(it==s.end())
   cout<<"Khong tim thay"<<endl;
 else
   cout<< "Co trong set"<<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 be hon 4"<<endl;
 else
   cout<<"Phan tu lon hon hoac bang 4 trong set la "<<*it<<endl; //
in ra 4
  it = s.lower_bound(10);
 if(it == s.end())
   cout<<"Khong co phan tu nao trong set lon hon 10" << endl;
 else
   cout<<"Phan tu trong set lon hon hoac bang 10 la "<< *it<<endl;

 it = s.upper_bound(4);
 if(it == s.end())
   cout << "Khong co phan tu trong set nao ma lon hon 4"<<endl;
 else
```

```
   cout<<"Phan tu be nhat lon hon 4 la: "<<*it<<endl;
  /* Duyet set */
 for(it=s.begin();it!=s.end();it++)
   cout<< *it << " ";

 cout<<endl;
}
```

Ví dụ 2: Nếu chúng ta muốn sử dụng hàm lower_bound hay upper_bound để tìm phần tử lớn nhất mà "bé hơn hoặc bằng" hoặc "bé hơn" ta có thể thay đổi cách so sánh của set để tìm kiếm.

```
#include <iostream>
#include <set>
#include <iterator>
using namespace std;
int main()
{
 set<int , greater<int>> s;
 set<int, greater<int>>::iterator it;

 s.insert(1); // s={1}
 s.insert(2); // s={2,1}
 s.insert(4); // s={4,2,1}
 s.insert(9); // s={5,4,2,1}

 /* Phan tu lon nhat be hon hoac bang 5*/
 it = s.lower_bound(5);
 cout<< *it <<endl;

 /* Phan tu lon nhat be hon 4*/
 it = s.upper_bound(4);
 cout<< *it <<endl;
}
```

1.3.2 Lớp map

- Mỗi phần tử của map là sự kết hợp của khóa (key value) và ánh xạ 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.

Khai báo: `#include <map>`

```
map<type_name_1, type_name2>
```

Trong đó:

- `type_name_1`: Kiểu dữ liệu của khóa.
- `type_name_2`: Kiểu dữ liệu giá trị của khóa.

Có thể sử dụng class so sánh:

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

Truy cập map khi sử dụng iterator:

```
(*it).first; // Lấy giá trị của khóa
it->first;
(*it).second; // Lấy giá trị của giá trị của khóa
it->second;
(*it) // Lấy giá trị phần tử mà iterator đang trỏ đến. Kiểu pair
```

Capacity:

- `size`: Trả về kích thước hiện tại của map.
- `empty`: Trả về true nếu map rỗng, và ngược lại.

Truy cập tới phần tử:

- `operator[khóa]`: Nếu khóa đã có trong map, thì hàm này 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.
- `at`: tương tự như toán tử [ ]

Modifiers:

- `insert`: Chèn phần tử vào map, phần tử chèn phải có dạng pair.
- `erase`: Xóa phần tử, xóa theo iterator hoặc xóa theo khóa.
- `clear`: Xóa tất cả map.
- `swap`: Hoán đổi 2 map cho nhau.

Operations:

- `find`: Trả về iterator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy thì iterator trả về "end" của map.
- `lower_bound`: Trả về iterator trỏ đến vị trí phần tử bé nhất mà lớn hơn hoặc bằng khóa (theo kiểu so sánh), nếu không tìm thấy trả về "end" của map.
- `upper_bound`: Trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về "end" của map.
- `count`: Trả về số lần xuất hiện của khóa.
-

Ví dụ:

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

int 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');

 cout<<it->first<<endl;
 cout<<it->second<<endl;

 m['e'] = 100;

 it = m.lower_bound('d');
 cout<<it->first<< endl; // in ra 'e'
 cout<<it->second<<endl; // in ra 100
}
```

1.3.3 Lớp multiset và multimap

- Multiset giống như set nhưng có thể chứa các khóa có giá trị giống nhau. Khai báo giống như set.
- Tương tự với multimap, tuy nhiên multimap không có toán tử [ ]

Dưới đây giới thiệu về các hàm multiset, với multimap cũng tương tự.

Capacity:

- `size`: Trả về kích thước hiện tại của multiset.
- `empty`: Trả về true nếu multiset rỗng và ngược lại.

Modifiers:

- `insert`: Chèn phần tử vào multiset.
- `erase`: Xóa phần tử khỏi multiset, có thể xóa theo iterator hoặc xóa theo khóa.
- `clear`: Xóa tất cả multiset.
- `swap`: Hoán đổi 2 multiset cho nhau.

Operations:

- `find`: Trả về iterator trỏ đến phần tử cần tìm kiếm. Nếu không tìm thấy thì iterator trả về "end" của multiset.
- `lower_bound`: Trả về iterator trỏ đến vị trí phần tử bé nhất mà lớn hơn hoặc bằng khóa (theo kiểu so sánh), nếu không tìm thấy trả về "end" của multiset.
- `upper_bound`: Trả về iterator đến vị trí phần tử bé nhất mà lớn hơn khóa, nếu không tìm thấy trả về "end" của multiset.
- `count`: Trả về số lần xuất hiện của khóa.

Ví dụ:

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

int main()
{
 multiset<int> s;
 multiset<int>::iterator it;
 int i;
 for(int 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;
 cout<<s.count(20)<<endl;

 for(it=s.begin();it!=s.end();it++)
   cout<< *it <<" ";
}
```

2. Iterator (Bộ duyệt)

Một iterator là một đối tượng có thể đi qua (iterate over) một container class mà không cần biết trật tự các phần tử bên trong của container. Iterator là một cách để truy cập dữ liệu bên trong các container.
Có thể hình dung iterator giống như một con trỏ trỏ đến một phần tử nào đó bên trong container với một số toán tử được định nghĩa:

- operator*: Trả về giá trị bên trong container tại vị trí iterator được đặt.
- operator++: di chuyển iterator đến phần tử tiếp theo trong container.
- operator--: ngược lại với operator++.
- operator== và operator!=: dùng để so sánh vị trí tương đối 2 phần tử đang được trỏ đến bởi 2 iterator.
- operator: dùng để gán vị trí iterator trỏ đến.

Khai báo một iterator

Với mỗi container class, chúng ta sẽ có một kiểu iterator tương ứng,

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

int main()
{
  std::vector<int> vec;

  std::vector<int>::iterator iter;

  return 0;
}
```

Các phương thức trả về kiểu iterator của các container class

Một giá trị muốn được gán cho một biến (hoặc một đối tượng nào đó) thì giá trị và biến đó phải cùng kiểu dữ liệu với nhau. Như vậy, muốn gán địa chỉ (vị trí) của một phần tử trong container cho một iterator thì chúng ta cũng cần có những phương thức trả về giá trị kiểu iterator tương ứng.

Mỗi container class trong STL (ngoại trừ các container đặc biệt như std::stack và std::queue) đều chứa định nghĩa của một iterator bên trong.
Những container có chứa định nghĩa class iterator sẽ có những phương thức trả về kiểu iterator tương ứng:

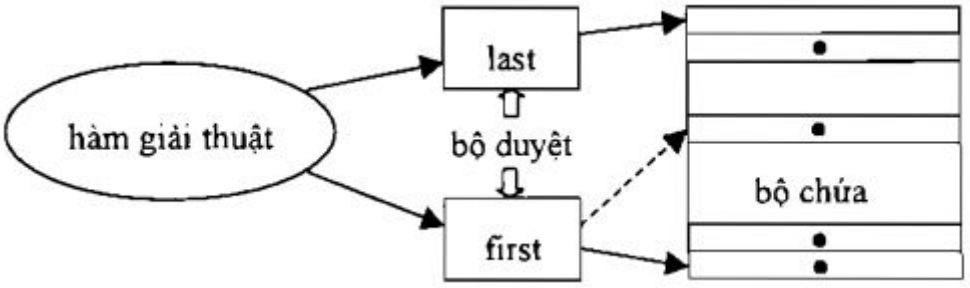
- `begin()`: Trả về một iterator đại diện cho vị trí của phần tử đầu tiên trong container.
- `end()`: Trả về một iterator đại diện cho vị trí đứng ngay sau phần tử cuối cùng trong container.
- `cbegin()`: Trả về một hằng (read-only) iterator đại diện cho vị trí của phần tử đầu tiên trong container.
- `cend()`: Trả về một hằng (read-only) iterator đại diện cho vị trí đứng ngay sau phần tử cuối cùng trong container.

Ví dụ:
Dùng iterator để duyệt tất cả phần tử của map.

```
#include <iostream>
#include <map>
#include <string>
int main()
{
   std::map<int, std::string> mymap;
   mymap.insert(std::make_pair(4, "apple"));
   mymap.insert(std::make_pair(2, "orange"));
   mymap.insert(std::make_pair(1, "banana"));
   mymap.insert(std::make_pair(3, "grapes"));
   mymap.insert(std::make_pair(6, "mango"));
   mymap.insert(std::make_pair(5, "peach"));
   std::map<int,std::string>::const_iterator it; // declare an
iterator
   it = mymap.begin(); // assign it to the start of the vector
   while (it != mymap.end()) // while it hasn't reach the end
   {
       std::cout << it->first << "=" << it->second << " "; // print
the value of the element it points to
       ++it; // and iterate to the next element
   }
}
```

3. Algorithm (Thuật toán)

Khuôn hình giải thuật làm việc với các bộ chứa thông qua các bộ duyệt. Hầu hết các khuôn hình giải thuật đều có dạng sau: "Duyệt từ vị trí này của bộ chứa đến vị trí kia của bộ chứa và thực hiện thao tác ...". Như vậy đầu vào của khuôn hình giải thuật thường là hai bộ duyệt trỏ đến vị trí đầu và cuối của quá trình duyệt trên bộ chứa. Để thực hiện thao tác, khuôn hình giải thuật cần phải truy cập lên các phần tử của bộ chứa bằng cách lấy tham chiếu của bộ duyệt duyệt.



Khai báo: `#include <algorithm>`

3.1 Các giải thuật không làm đổi biến

3.1.1 for_each

Duyệt qua bộ chứa.
Ví dụ:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void myFunction(int i)
{
 cout << " " << i;
}
struct myclass{
 void operator() (int i){
   cout<<" "<<i;
 }
};
```

```
int main()
{
 vector<int> myvector;
 myvector.push_back(10);
 myvector.push_back(20);
 myvector.push_back(30);

 for_each(myvector.begin(), myvector.end(), myFunction);
 cout<<endl;

 // or
 myclass myobject;
 for_each(myvector.begin(), myvector.end(), myobject);
 cout<<endl;
}
```

3.1.2 find

Trả về iterator trỏ đến phần tử đầu tiên trong khoảng [first, last) mà bằng value, nếu không tìm thấy, trỏ đến phần tử last.

Ví dụ:

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

int main()
{
 // sử dụng find tìm kiếm trong mảng
 int myints[] = {10,20,30,40};
 int *p;
 p=find(myints, myints+4, 30);
 if(p!=myints+4)
   cout<<"Find: "<<*p<<endl;
 else
   cout<<"Element not found"<<endl;
```

```
 // sử dụng find tìm kiếm trong vector
 vector<int> myvector(myints, myints+4);
 vector<int>::iterator it;
 it=find(myvector.begin(),myvector.end(), 30);
 if(it!=myvector.end())
   cout<<"Find: "<< *it<<endl;
 else
   cout<<"Element not found"<<endl;
}
```

3.1.3 find_if

Tương tự như find, nhưng có điều kiện nào đó.

Ví dụ:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
bool IsOdd(int i){
 return (i%2==1);
}
int main()
{
 vector<int> myvector;
 myvector.push_back(10);
 myvector.push_back(25);
 myvector.push_back(40);
 myvector.push_back(55);

 vector<int>::iterator it=find_if(myvector.begin(), myvector.end(),
IsOdd);
 cout<<*it<<endl;
}
```

3.1.4 find_first_of

Tương tự như find, thì find_first_of cũng thực hiện tìm kiếm tuyến tính trên một dãy các đối tượng. Tuy nhiên, khuôn hình giải thuật find chỉ tìm kiếm một giá trị đặc biệt nào đó trong khi find_first_of tìm kiếm một giá trị bất kỳ trong khoảng định trước.

Ví dụ 1:

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

int main()
{
 int a1[7] = {10,13,25,3,9,44,7};
 int a2[4] = {1,2,3,4};

 list<int> L1;
 L1.insert(L1.begin(), a1, a1+7);
 list<int> L2;
 L2.insert(L2.begin(), a2, a2+4);

 list<int>::iterator it=find_first_of(L1.begin(), L1.end(),
L2.begin(), L2.end());
 cout<<*it<<endl;
}
```

Ví dụ với dãy `a1[7] = {10,13,25,3,9,44,7}` và `a2[4] = {1,2,3,4}` khi áp dụng find_first_of sẽ nhận được kết quả là bộ duyệt trỏ vào phần tử thứ tư của a1, vì trong dãy a2 có phần tử thứ ba cũng có giá trị là 3.

Ví dụ 2:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::find_first_of
#include <vector>       // std::vector
#include <cctype>       // std::tolower

bool comp_case_insensitive (char c1, char c2) {
 return (std::tolower(c1)==std::tolower(c2));
}

int main () {
 int mychars[] = {'a','b','c','A','B','C'};
 std::vector<char> haystack (mychars,mychars+6);
 std::vector<char>::iterator it;

 int needle[] = {'A','B','C'};

 // using default comparison:
 it = find_first_of (haystack.begin(), haystack.end(), needle,
needle+3);

 if (it!=haystack.end())
   std::cout << "The first match is: " << *it << '\n';

 // using predicate comparison:
 it = find_first_of (haystack.begin(), haystack.end(),
                     needle, needle+3, comp_case_insensitive);

 if (it!=haystack.end())
   std::cout << "The first match is: " << *it << '\n';

 return 0;
}
```

3.1.5 adjacent_find

Sử dụng khi muốn tìm hai số liên tiếp có cùng tính chất nào đó. Mặc định là tính chất là phép so sánh bằng (operator==). Nếu không tìm thấy, hàm trả về bộ duyệt trỏ tới vị trí cuối.
Ví dụ:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::adjacent_find
#include <vector>       // std::vector

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

int main () {
 int myints[] = {5,20,5,30,30,20,10,10,20};
 std::vector<int> myvector (myints,myints+8);
 std::vector<int>::iterator it;

 // using default comparison:
 it = std::adjacent_find (myvector.begin(), myvector.end());

 if (it!=myvector.end())
   std::cout << "the first pair of repeated elements are: " << *it <<
'\n';

 //using predicate comparison:
 it = std::adjacent_find (++it, myvector.end(), myfunction);

 if (it!=myvector.end())
   std::cout << "the second pair of repeated elements are: " << *it
<< '\n';

 return 0;
}
```

3.1.6 count

Đếm xem trong một khoảng [first, last) có bao nhiêu phần tử có giá trị bằng value.
Ví dụ:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::count
#include <vector>       // std::vector

int main () {
 // counting elements in array:
 int myints[] = {10,20,30,30,20,10,10,20};   // 8 elements
 int mycount = std::count (myints, myints+8, 10);
 std::cout << "10 appears " << mycount << " times.\n";

 // counting elements in container:
 std::vector<int> myvector (myints, myints+8);
 mycount = std::count (myvector.begin(), myvector.end(), 20);
 std::cout << "20 appears " << mycount  << " times.\n";

 return 0;
}
```

3.1.7 count_if

Các khuôn hình giải thuật có _if là mở rộng của các khuôn hình giải thuật cùng tên. count_if đếm phần tử theo điều kiện nào đó (từ hàm).
Ví dụ:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::count_if
#include <vector>       // std::vector

bool IsOdd (int i) { return ((i%2)==1); }

int main () {
```

```
 std::vector<int> myvector;
 for (int i=1; i<10; i++) myvector.push_back(i); // myvector: 1 2 3 4
5 6 7 8 9

 int mycount = count_if (myvector.begin(), myvector.end(), IsOdd);
 std::cout << "myvector contains " << mycount  << " odd values.\n";

 return 0;
}
```

3.1.8 mismatch

Tìm kiếm sự khác nhau đầu tiên giữa 2 dãy.

Ví dụ:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::mismatch
#include <vector>       // std::vector
#include <utility>      // std::pair

bool mypredicate (int i, int j) {
 return (i==j);
}

int main () {
 std::vector<int> myvector;
 for (int i=1; i<6; i++) myvector.push_back (i*10); // myvector: 10
20 30 40 50

 int myints[] = {10,20,80,320,1024};                //   myints: 10
20 80 320 1024

 std::pair<std::vector<int>::iterator,int*> mypair;

 // using default comparison:
 mypair = std::mismatch (myvector.begin(), myvector.end(), myints);
 std::cout << "First mismatching elements: " << *mypair.first;
 std::cout << " and " << *mypair.second << '\n';
```

```
 ++mypair.first; ++mypair.second;

 // using predicate comparison:
 mypair = std::mismatch (mypair.first, myvector.end(), mypair.second,
mypredicate);
 std::cout << "Second mismatching elements: " << *mypair.first;
 std::cout << " and " << *mypair.second << '\n';

 return 0;
}
```

3.1.9 equal

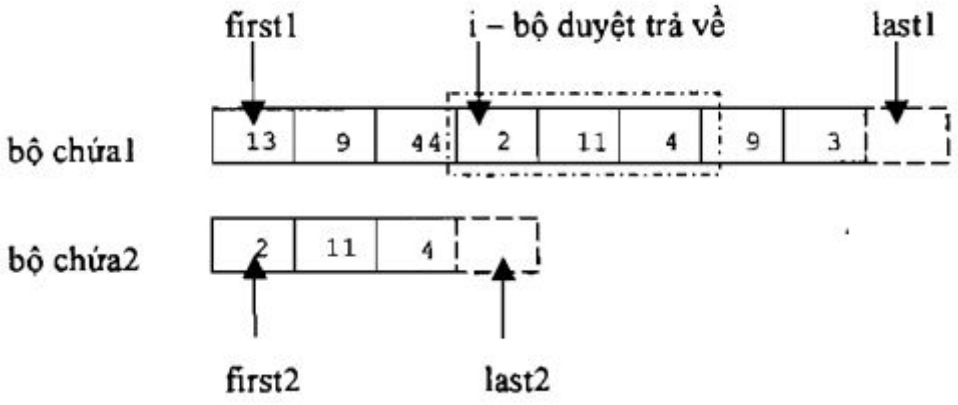
Kiểm tra xem 2 dãy có giống nhau hay không?
Ví dụ:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::equal
#include <vector>       // std::vector

bool mypredicate (int i, int j) {
 return (i==j);
}

int main () {
 int myints[] = {20,40,60,80,100};               //   myints: 20 40
60 80 100
 std::vector<int>myvector (myints,myints+5);     // myvector: 20 40
60 80 100

 // using default comparison:
 if ( std::equal (myvector.begin(), myvector.end(), myints) )
   std::cout << "The contents of both sequences are equal.\n";
 else
   std::cout << "The contents of both sequences differ.\n";

 myvector[3]=81;  // myvector: 20 40 60 81 100
```

```
 // using predicate comparison:
 if ( std::equal (myvector.begin(), myvector.end(), myints,
mypredicate) )
   std::cout << "The contents of both sequences are equal.\n";
 else
   std::cout << "The contents of both sequences differ.\n";

 return 0;
}
```

3.1.10 search

Khuôn hình giải thuật search tìm trong dãy [first1, last1) dãy con giống với dãy [first2, last2). Nếu tìm thấy, search trả về bộ duyệt it đầu tiên của dãy con. Nếu không tìm thấy dãy con nào thỏa mãn, search trả về last1.



Khuôn hình giải thuật search không quy định số phần tử của dãy thứ nhất phải nhiều hơn số phần tử của dãy thứ hai.
Ví dụ:

```
#include <iostream>      // std::cout
#include <algorithm>     // std::search
#include <vector>        // std::vector

bool mypredicate (int i, int j) {
 return (i==j);
}
```

```
int main () {
 std::vector<int> haystack;

 // set some values:         haystack: 10 20 30 40 50 60 70 80 90
 for (int i=1; i<10; i++) haystack.push_back(i*10);

 // using default comparison:
 int needle1[] = {40,50,60,70};
 std::vector<int>::iterator it;
 it = std::search (haystack.begin(), haystack.end(), needle1,
needle1+4);

 if (it!=haystack.end())
   std::cout << "needle1 found at position " << (it-haystack.begin())
<< '\n';
 else
   std::cout << "needle1 not found\n";

 // using predicate comparison:
 int needle2[] = {20,30,50};
 it = std::search (haystack.begin(), haystack.end(), needle2,
needle2+3, mypredicate);

 if (it!=haystack.end())
   std::cout << "needle2 found at position " << (it-haystack.begin())
<< '\n';
 else
   std::cout << "needle2 not found\n";

 return 0;
}
```

3.1.11 search_n

Tìm trong khoảng [first, last) dãy con gồm count phần tử trong đó tất cả các phần tử đều có giá trị bằng value. Nếu tìm thấy, search_n trả về bộ duyệt đầu tiên của dãy con. Nếu không trả về last.

Ví dụ:

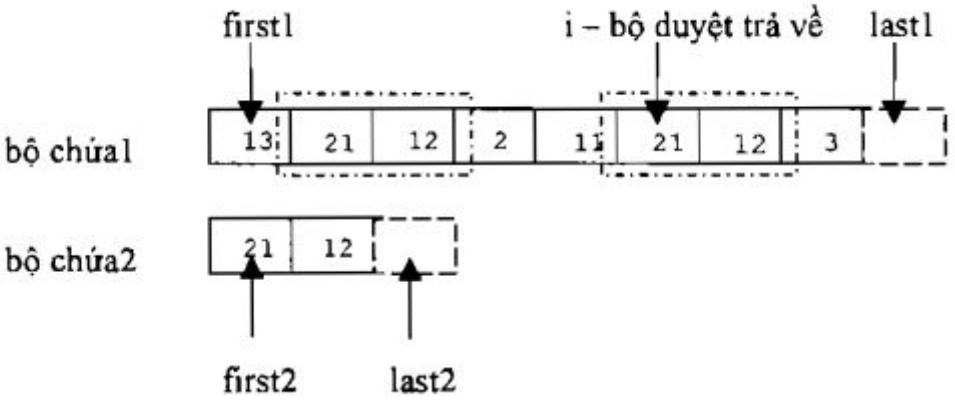
```
#include <iostream>     // std::cout
#include <algorithm>    // std::search_n
#include <vector>       // std::vector

bool mypredicate (int i, int j) {
 return (i==j);
}

int main () {
 int myints[]={10,20,30,30,20,10,10,20};
 std::vector<int> myvector (myints,myints+8);

 std::vector<int>::iterator it;

 // using default comparison:
 it = std::search_n (myvector.begin(), myvector.end(), 2, 30);

 if (it!=myvector.end())
   std::cout << "two 30s found at position " << (it-myvector.begin())
<< '\n';
 else
   std::cout << "match not found\n";

 // using predicate comparison:
 it = std::search_n (myvector.begin(), myvector.end(), 2, 10,
```

```
mypredicate);

 if (it!=myvector.end())
   std::cout << "two 10s found at position " <<
int(it-myvector.begin()) << '\n';
 else
   std::cout << "match not found\n";

 return 0;
}
```

3.1.12 find_end

Đúng ra, khuôn hình giải thuật find_end phải có tên là search_end vì find_end giống với các khuôn hình giải thuật search hơn là giống với các khuôn hình giải thuật find.

Tương tự như search, find_end cũng tìm dãy con trong khoảng [first1, last1) sao cho dãy này giống với dãy con [first2, last2), nhưng khác với search, find_end tìm dãy con cuối cùng thỏa mãn tính chất đó. Nếu tìm thấy, find_end trả về bộ duyệt đầu tiên của dãy con, nếu không tìm thấy, find_end trả về last1.



Ví dụ:

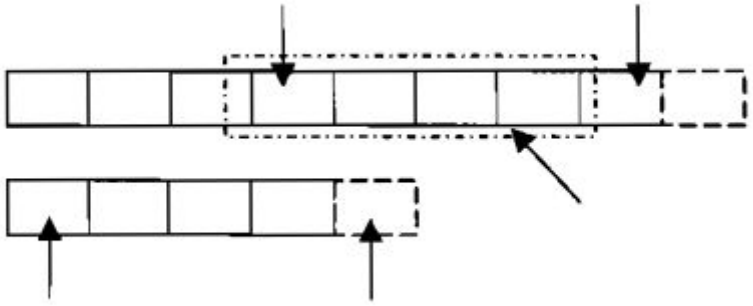
```
#include <iostream>      // std::cout
#include <algorithm>     // std::find_end
#include <vector>        // std::vector

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

```
int main () {
 int myints[] = {1,2,3,4,5,1,2,3,4,5};
 std::vector<int> haystack (myints,myints+10);

 int needle1[] = {1,2,3};

 // using default comparison:
 std::vector<int>::iterator it;
 it = std::find_end (haystack.begin(), haystack.end(), needle1,
needle1+3);
 if (it!=haystack.end())
   std::cout << "needle1 last found at position " <<
(it-haystack.begin()) << '\n';

 int needle2[] = {4,5,1};

 // using predicate comparison:
 it = std::find_end (haystack.begin(), haystack.end(), needle2,
needle2+3, myfunction);

 if (it!=haystack.end())
   std::cout << "needle2 last found at position " <<
(it-haystack.begin()) << '\n';

 return 0;
}
```

3.2 Các giải thuật làm đổi biến

Các giải thuật làm đổi biến khi thực hiện trên các bộ chứa sẽ gây biến đổi lên các đối tượng lưu trữ trong các bộ chứa đó. Các giải thuật làm đổi biến bao gồm các hàm sao chép, hoán vị, thay thế, khai triển, …

3.2.1 copy



Ví dụ:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::copy
#include <vector>       // std::vector

int main () {
 int myints[]={10,20,30,40,50,60,70};
 std::vector<int> myvector (7);

 std::copy ( myints, myints+7, myvector.begin() );

 std::cout << "myvector contains:";
 for (std::vector<int>::iterator it = myvector.begin();
it!=myvector.end(); ++it)
   std::cout << ' ' << *it;

 std::cout << '\n';

 return 0;
}
```

3.1.2 copy_backward

Tương tự như copy nhưng là sao chép ngược.
Ví dụ:

```
#include <iostream>     // std::cout
```

```
#include <algorithm>     // std::copy_backward
#include <vector>        // std::vector

int main () {
 std::vector<int> myvector;

 // set some values:
 for (int i=1; i<=5; i++)
   myvector.push_back(i*10);          // myvector: 10 20 30 40 50

 myvector.resize(myvector.size()+3);  // allocate space for 3 more
elements

 std::copy_backward ( myvector.begin(), myvector.begin()+5,
myvector.end() );

 std::cout << "myvector contains:";
 for (std::vector<int>::iterator it=myvector.begin();
it!=myvector.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 return 0;
}
```

3.2.3 swap

Hoán đổi 2 kiểu dữ liệu bất kỳ.

3.2.4 swap_range

Hoán đổi mỗi phần tử trong khoảng [first, last) tương ứng với một phần tử trong khoảng [first2, first2 + (last1-first1)).

Ví dụ:

```
#include <iostream>      // std::cout
#include <algorithm>     // std::swap_ranges
#include <vector>        // std::vector

int main () {
 std::vector<int> foo (5,10);         // foo: 10 10 10 10 10
```

```
 std::vector<int> bar (5,33);         // bar: 33 33 33 33 33

 std::swap_ranges(foo.begin()+1, foo.end()-1, bar.begin());

 // print out results of swap:
 std::cout << "foo contains:";
 for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 std::cout << "bar contains:";
 for (std::vector<int>::iterator it=bar.begin(); it!=bar.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 return 0;
}
```

3.2.5 transform

Ví dụ:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;
int main () {
 vector<int> v(5), y(8);
 // Khởi tạo v với giá trị 5
 fill(v.begin(), v.end(), 5); // v={5,5,5,5,5}
 // Khởi tạo nửa đầu cảu y với giá trị 2
 fill(y.begin(), y.begin()+4, 2);
 // Khởi tạo nửa cuối của y với giá trị -2
 transform(y.begin(), y.begin()+4, y.begin()+4, negate<int>());
// y={2,2,2,2,-2,-2,-2,-2}
 ostream_iterator<int> oi(cout, " ");
 cout <<"\nv: ";
 copy(v.begin(), v.end(), oi);
```

```
 cout<<"\ny: ";
 copy(y.begin(), y.end(), oi);
  cout<<"\nv+y: ";
 // Cộng 2 vector bằng transform và hiển thị
// 7 7 7 7 3
 transform(v.begin(), v.end(), y.begin(), oi, plus<int>());
}
```

3.2.6 replace và replace_if

Khuôn hình giải thuật replace thay thế mọi phần tử trong khoảng [first, last) có giá trị old_value bằng giá trị mới new_value.replace_if tương tự.

```
replace (myvector.begin(), myvector.end(), 20, 99);
replace_if (myvector.begin(), myvector.end(), IsOdd, 0);
```

3.2.7 replace_copy và replace_copy_if

Vừa thực hiện sao chép vừa thực hiện thay thế giá trị.

```
replace_copy (myints, myints+8, myvector.begin(), 20, 99);
replace_copy_if (foo.begin(), foo.end(), bar.begin(), IsOdd, 0);
```

3.2.8 fill và fill_n

Khuôn hình giải thuật fill gán giá trị value cho mọi phần tử trong khoảng [first, last). Còn fill_n thực hiện gán n phần tử.

```
fill (myvector.begin(),myvector.begin()+4,5);
fill_n (myvector.begin(),4,20);
```

3.2.9 generate và generate_n

Khuôn hình giải thuật generate cũng là một trong các giải thuật sinh số như fill nhưng tiện lợi hơn. Trong khi fill chỉ sinh các số giống hệt nhau, generate sử dụng đối tượng hàm có thể sinh với các số ngẫu nhiên hoặc các số theo một quy tắc nào đó. Tương tự với generate_n.

Ví dụ:

```
// generate algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::generate
#include <vector>       // std::vector
#include <ctime>        // std::time
#include <cstdlib>      // std::rand, std::srand

// function generator:
int RandomNumber () { return (std::rand()%100); }

// class generator:
struct c_unique {
 int current;
 c_unique() {current=0;}
 int operator()() {return ++current;}
} UniqueNumber;

int main () {
 std::srand ( unsigned ( std::time(0) ) );

 std::vector<int> myvector (8);

 std::generate (myvector.begin(), myvector.end(), RandomNumber);

 std::cout << "myvector contains:";
 for (std::vector<int>::iterator it=myvector.begin();
it!=myvector.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 std::generate (myvector.begin(), myvector.end(), UniqueNumber);

 std::cout << "myvector contains:";
 for (std::vector<int>::iterator it=myvector.begin();
it!=myvector.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';
```

```
 return 0;
}
```

3.2.10 remove, remove_if, remove_copy, remove_copy_if

Tương tự như các phương thức của replace, nhưng đây là xóa.

3.2.11 unique và unique_copy

unique xóa khỏi dãy [first, last) các phần tử giống nhau liên tiếp, trừ phần tử đầu tiên.
Nghĩa là dãy {1,1,1,2,2,1,1} trở thành {1,2,1}.
unique_copy tương tự unique nhưng thực hiện copy.
Ví dụ 1:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::unique, std::distance
#include <vector>       // std::vector

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

int main () {
 int myints[] = {10,20,20,20,30,30,20,20,10};            // 10 20 20
20 30 30 20 20 10
 std::vector<int> myvector (myints,myints+9);

 // using default comparison:
 std::vector<int>::iterator it;
 it = std::unique (myvector.begin(), myvector.end());   // 10 20 30
20 10 ?  ?  ?  ?
                                                        //
^

 myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30
20 10

 // using predicate comparison:
```

```
 std::unique (myvector.begin(), myvector.end(), myfunction);   // (no
changes)

 // print out content:
 std::cout << "myvector contains:";
 for (it=myvector.begin(); it!=myvector.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 return 0;
}
```

Ví dụ 2:

```
// unique_copy example
#include <iostream>     // std::cout
#include <algorithm>    // std::unique_copy, std::sort, std::distance
#include <vector>       // std::vector

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

int main () {
 int myints[] = {10,20,20,20,30,30,20,20,10};
 std::vector<int> myvector (9);                          // 0  0  0  0  0  0  0  0  0

 // using default comparison:
 std::vector<int>::iterator it;
 it=std::unique_copy (myints,myints+9,myvector.begin());  // 10 20 30 20 10 0  0  0  0
                                                          //                ^

 std::sort (myvector.begin(),it);                         // 10 10 20 20 30 0  0  0  0
                                                          //                ^

 // using predicate comparison:
 it=std::unique_copy (myvector.begin(), it, myvector.begin(), myfunction);
                                                          // 10 20 30 20 30 0  0  0  0
                                                          //          ^

 myvector.resize( std::distance(myvector.begin(),it) );   // 10 20 30

 // print out content:
 std::cout << "myvector contains:";
 for (it=myvector.begin(); it!=myvector.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';
```

```
 return 0;
}
```

3.2.12 reverse và reverse_copy

Đảo ngược thứ tự dãy.

3.2.13 rotate và rotate_copy

Hoán chuyển vị trí các phần tử trong [first, last) sao cho [first, middle) thế chỗ [middle, last) và ngược lại.
Ví dụ:

```
// rotate algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::rotate
#include <vector>        // std::vector

int main () {
 std::vector<int> myvector;

 // set some values:
 for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

 std::rotate(myvector.begin(),myvector.begin()+3,myvector.end());
                                                 // 4 5 6 7 8 9 1 2 3
 // print out content:
 std::cout << "myvector contains:";
 for (std::vector<int>::iterator it=myvector.begin();
it!=myvector.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 return 0;
}
```

3.2.14 random_suffle

Tạo bộ hoán vị ngẫu nhiên giữa các phần tử của dãy [first, last).

```
random_shuffle ( myvector.begin(), myvector.end() );
```

3.2.15 partition

partition thay đổi lại thứ tự các phần tử trong khoảng [first, last) dưa trên đối tượng hàm mệnh đề pred sao cho các phần tử thỏa mãn pred sẽ đứng trước các phần tử không thỏa mãn pred.
Ví dụ:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::partition
#include <vector>       // std::vector

bool IsOdd (int i) { return (i%2)==1; }

int main () {
 std::vector<int> myvector;

 // set some values:
 for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

 std::vector<int>::iterator bound;
 bound = std::partition (myvector.begin(), myvector.end(), IsOdd);

 // print out content:
 std::cout << "odd elements:";
 for (std::vector<int>::iterator it=myvector.begin(); it!=bound;
++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 std::cout << "even elements:";
 for (std::vector<int>::iterator it=bound; it!=myvector.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 return 0;
}
```

3.3 Các giải thuật sắp xếp

Các giải thuật không xét đến trong phần này (tự tìm hiểu): `partial_sort, partial_sort_copy, nth_element, lower_bound, upper_bound, equal_range, merge, inplace_merge.`

3.3.1 sort và stable_sort

Khuôn hình giải thuật sort sắp xếp các phần tử trong dãy [first, last) sao cho phần tử đứng sau không nhỏ hơn phần tử đứng trước. Ta cũng có thể thay đổi quy tắc sắp xếp.
Ví dụ:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector

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};
 std::vector<int> myvector (myints, myints+8);               // 32 71
12 45 26 80 53 33

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

 // using function as comp
 std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32
45 71(26 33 53 80)

 // using object as comp
 std::sort (myvector.begin(), myvector.end(), myobject);     //(12 26
32 33 45 53 71 80)
```

```
 // print out content:
 std::cout << "myvector contains:";
 for (std::vector<int>::iterator it=myvector.begin();
it!=myvector.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 return 0;
}
```

Khuôn hình giải thuật stable_sort tương tự như sort nhưng giữ nguyên thứ tự tương đối của các phần tử. Nghĩa là nếu x đứng trước y, x không nhỏ hơn y và y cũng không nhỏ hơn x, sau khi sắp xếp x vẫn đứng trước y.
Ví dụ:

```
// stable_sort example
#include <iostream>      // std::cout
#include <algorithm>     // std::stable_sort
#include <vector>        // std::vector

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

int main () {
 double mydoubles[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62,
2.58};

 std::vector<double> myvector;

 myvector.assign(mydoubles,mydoubles+8);

 std::cout << "using default comparison:";
 std::stable_sort (myvector.begin(), myvector.end());
 for (std::vector<double>::iterator it=myvector.begin();
it!=myvector.end(); ++it)
```

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

 myvector.assign(mydoubles,mydoubles+8);

 std::cout << "using 'compare_as_ints' :";
 std::stable_sort (myvector.begin(), myvector.end(),
compare_as_ints);
 for (std::vector<double>::iterator it=myvector.begin();
it!=myvector.end(); ++it)
   std::cout << ' ' << *it;
 std::cout << '\n';

 return 0;
}
```

3.3.2 min, min_element, max, max_element

min trả về đối tượng nhỏ hơn trong 2 đối tượng a và b.
min_element trả về đối tượng nhỏ nhất trong dãy [first, last). Tương tự với các khuôn hình giải thuật max và max_element.

3.4 Các giải thuật trên tập hợp

3.4.1 includes

Khuôn hình giải thuật includes kiểm tra xem dãy đã sắp xếp [first1, last1) có chứa dãy sắp xếp [first2, last2) hay không (4 tham số). Yêu cầu của includes nói riêng và các giải thuật trên tập hợp nói chung là các dãy đều phải là các dãy đã sắp xếp tăng dần.

3.4.2 set_union

Khuôn hình giải thuật set_union tạo ra mỗt dãy sắp xếp là hợp của hai dãy sắp xếp [first1, last1) và [first2, last2). Kết quả trả về con trỏ đến vị trí sau cuối của dãy kết quả.
Ví dụ:

```
// set_union example
#include <iostream>     // std::cout
#include <algorithm>    // std::set_union, std::sort
```

```
#include <vector>       // std::vector

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);   // 0  0  0  0  0  0  0  0  0  0
  std::vector<int>::iterator it;

  std::sort (first,first+5);     //  5 10 15 20 25
  std::sort (second,second+5);   // 10 20 30 40 50

  it=std::set_union (first, first+5, second, second+5, v.begin());
 // 5 10 15 20 25 30 40 50  0  0
  v.resize(it-v.begin());   // 5 10 15 20 25 30 40 50

  std::cout << "The union has " << (v.size()) << " elements:\n";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

3.4.3 set_intersection

Thực hiện phép giao giữa 2 tập hợp giữa 2 khoảng [first1, last1) và [first2, last2).

Ví dụ:

```
// set_intersection example
#include <iostream>     // std::cout
#include <algorithm>    // std::set_intersection, std::sort
#include <vector>       // std::vector

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);                      // 0  0  0  0  0  0  0
0  0  0
  std::vector<int>::iterator it;

  std::sort (first,first+5);     //  5 10 15 20 25
  std::sort (second,second+5);   // 10 20 30 40 50

  it=std::set_intersection (first, first+5, second, second+5,
v.begin());
                                               // 10 20 0  0  0  0  0
0  0  0
  v.resize(it-v.begin());                      // 10 20
```

```
  std::cout << "The intersection has " << (v.size()) << "
elements:\n";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

3.4.4 set_difference

Sự khác biệt của 2 tập hợp, xem trong tập hợp thứ nhất những phần tử nào khác biệt so với tập hợp thứ hai.
Ví dụ:

```
// set_difference example
#include <iostream>     // std::cout
#include <algorithm>    // std::set_difference, std::sort
#include <vector>       // std::vector

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);                      // 0  0  0  0  0  0  0
0  0  0
  std::vector<int>::iterator it;

  std::sort (first,first+5);     //  5 10 15 20 25
  std::sort (second,second+5);   // 10 20 30 40 50

  it=std::set_difference (first, first+5, second, second+5,
v.begin());
                                               //  5 15 25  0  0  0
0  0  0  0
  v.resize(it-v.begin());                      //  5 15 25

  std::cout << "The difference has " << (v.size()) << " elements:\n";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

3.4.5 set_symmetric_difference

Tìm sự khác biệt đối xứng giữa 2 tập hợp.

Ví dụ:

```
// set_symmetric_difference example
#include <iostream>     // std::cout
#include <algorithm>    // std::set_symmetric_difference, std::sort
#include <vector>       // std::vector

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);                      // 0  0  0  0  0  0  0
0  0  0
  std::vector<int>::iterator it;

  std::sort (first,first+5);     //  5 10 15 20 25
  std::sort (second,second+5);   // 10 20 30 40 50

  it=std::set_symmetric_difference (first, first+5, second, second+5,
v.begin());
                                               //  5 15 25 30 40 50
0  0  0  0
  v.resize(it-v.begin());                      //  5 15 25 30 40 50

  std::cout << "The symmetric difference has " << (v.size()) << "
elements:\n";
  for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```