TỔNG QUAN VỀ THƯ VIỆN CHUẨN STL

I. GIỚI THIỆU THƯ VIỆN CHUẨN STL

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.

STL gồm các thành phần chính:

- Container (các bộ lưu trữ dữ liệu) là các cấu trúc dữ liệu phổ biến đã template hóa dùng để lưu trữ các kiểu dữ liệu khác nhau. Các container chia làm 2 loại:
  - Sequential container (các ctdl tuần tự) bao gồm list, vector và deque
  - Asociative container (các ctdl liên kết) bao gồm map, multimap, set và multiset
- Iterator (biến lặp) giống như con trỏ, tích hợp bên trong container
- Algorithm (các thuật toán) là các hàm phổ biến để làm việc với các bộ lưu trữ như thêm, xóa, sửa, truy xuất, tìm kiếm, sắp xếp ...
- Function object (functor): Một kiểu đối tượng có thể gọi như 1 hàm, đúng ra đây là 1 kỹ thuật nhưng trong STL nó được nâng cao và kết hợp với các algorithm
- Các adapter (bộ tương thích) , chia làm 3 loại:
  - container adapter (các bộ tương thích lưu trữ) bao gồm stack, queue và priority_queue
  - iterator adapter (các bộ tương thích con trỏ)
  - function adapter (các bộ tương thích hàm)

Những thành phần này làm việc chung với các thành phần khác để cung cấp các giải pháp cho các vấn đề khác nhau của chương trình.

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.

Đặc điểm thư viện STL là được hỗ trợ trên các trình biên dịch ở cả hai môi trường WINDOWS lẫn UNIX, vì vậy nên khi sử dụng thư viện này trong xử lý thuận tiện cho việc chia sẽ mã nguồn với cộng đồng phát triển.

Vì thư viện chuẩn được thiết kế bởi những chuyện gia hàng đầu và đã được chứng minh tính hiệu quả trong lịch sử tồn tại của nó, các thành phần của thư viện này được khuyến cáo sử dụng thay vì dùng những phần viết tay bên ngoài hay những phương tiện cấp thấp khác. Thí dụ, dùng std::vector hay std::string thay vì dùng kiểu mảng đơn thuần là một cách hữu hiệu để viết phần mềm được an toàn và linh hoạt hơn.

Các chức năng của thư viện chuẩn C++ được khai báo trong namespace std;

Dưới đây ta sẽ tìm hiểu từng thành phần của STL

II. NHẬP XUẤT VỚI IOSTREAM

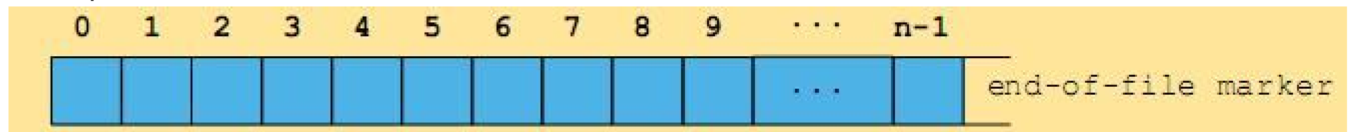
Như chúng ta sẽ thấy, C++ sử dụng nhập/xuất kiểu an toàn (type safe). Việc nhập/xuất được thực hiện một cách tự động theo lối nhạy cảm về kiểu dữ liệu. Mỗi thao tác nhập xuất có được định nghĩa thích hợp để xử lý một kiểu dữ liệu cụ thể thì hàm đó được gọi để xử lý kiểu dữ liệu đó. Nếu không có đối sánh giữa kiểu của dữ liệu hiện tại và một hàm cho việc xử lý kiểu dữ liệu đó, một chỉ dẫn lỗi biên dịch được thiết lập. Vì thế dữ liệu không thích hợp không thể "lách" qua hệ thống.

Các đặc tính nhập xuất mô tả theo hướng đối tượng. Người dùng có thể chỉ định nhập/xuất của các kiểu dữ liệu tự định nghĩa cũng như các kiểu dữ liệu chuẩn. Khả năng mở rộng này là một trong các đặc tính quan trọng của C++.

1.CÁC LỚP STREAM

C++ sử dụng khái niệm dòng tin (stream) và đưa ra các lớp dòng tin để tổ chức việc nhập xuất. Dòng tin có thể xem như một dẫy các byte. Thao tác nhập là lấy (đọc) các byte từ dòng tin (khi đó gọi là dòng nhập - input) vào bộ nhớ. Thao tác xuất là đưa các byte từ bộ nhớ ra dòng tin (khi đó gọi là dong xuất - output). Các thao tác này là độc lập thiết bị. Để thực hiện việc nhập, xuất lên một thiết bị cụ thể, chúng ta chỉ cần gắn dòng tin với thiết bị này.

Khái nệm stream:



–chuỗi byte, kết thúc bởi ký hiệu end_of_file

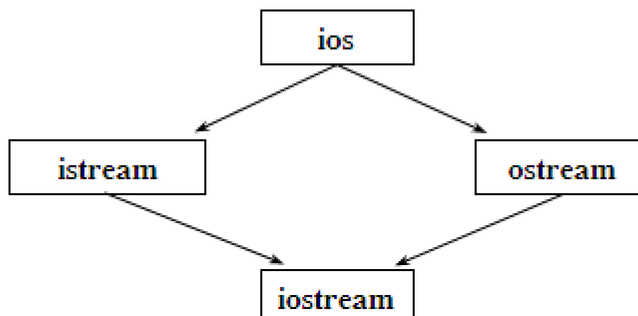
– Input: từ bàn phím, đĩa... vào bộ nhớ

– Output: từ bộ nhớ ra màn hình, máy in...

– file cũng được coi là một dòng

Lớp streambuf là cơ sở cho tất cả các thao tác vào ra bằng toán tử, nó định nghĩa các đặc trưng cơ bản của các vùng đệ m lưu trữ các ký tự để xuất hayn hập. Lớp ios là lớp dẫn xuất từ streambuf , ios đị nh nghĩa các dạng cơ bản và khả năng kiểm tra lỗi dùng cho streambuf . ios là lớp cơ sở ảo cho các lớp istream và ostream. Mỗi lớp này có định nghĩa chồng toán tử “ << ” và “ >> ” cho các kiểu dữ liệ u cơ sở khác nhau.

Có 4 lớp quan trọng cần nhớ là:
+ Lớp cơ sở ios
+ Từ lớp ios dẫn xuất đến 2 lớp istream và ostream
+ Hai lớp istream và ostream lại dẫn xuất tới lớp iostream
Sơ đồ kế thừa giữa các lớp như sau:



- Lớp ios
  + Thuộc tính của lớp: Trong lớp ios định nghĩa các thuộc tính được sử dụng làm các cờ định dạng cho việc nhập xuất và các cờ kiểm tra lỗi (xem bên dưới).
  + Các phương thức: Lớp ios cung cấp một số phương thức phục vụ việc định dạng dữ liệu nhập xuất, kiểm tra lỗi (xem bên dưới).

- Lớp istream
  Lớp này cung cấp toán tử nhập >> và nhiều phương thức nhập khác (xem bên dưới) như các phương thức: get, getline, read, ignore, peek, seekg, tellg,...

- Lớp ostream
  Lớp này cung cấp toán tử xuất << và nhiều phương thức xuất khác (xem bên dưới) như các phương thức: put, write, flush, seekp, tellp,...

- Lớp iostream
  Lớp này thừa kế các phương thức nhập xuất của các lớp istream và ostream.

Thư viện iostream của C++ cung cấp hàng trăm khả năng của nhập/xuất. Một vài tập tin header chứa các phần của giao diện thư viện:

- Phần lớn chương trình C++ thường include tập tin header <iostream> mà chứa các thông tin cơ bản đòi hỏi tất cả các thao tác dòng nhập/xuất:
  - dòng nhập chuẩn nối với thiết bị nhập chuẩn – Standard input (cin)
  - dòng xuất chuẩn nối với thiết bị xuất chuẩn – Standard output (cout)
  - dòng báo lỗi - nối với thiết bị báo lỗi chuẩn:

    Không có bộ nhớ đệm (unbuffered error) cerr

    Có dùng bộ nhớ đệm (buffered error) clog

- Header <iomanip> chứa thông tin hữu ích cho việc thực hiện nhập/xuất định dạng với tên gọi là các bộ xử lý dòng biểu hiện bằng tham số (parameterized stream manipulators).

- Header <fstream> chứa các thông tin quan trọng cho các thao tác xử lý file do người dùng kiểm soát.

- Header <strstream> chứa các thông tin quan trọng cho việc thực hiện các định dạng trong bộ nhớ. Điều này tương tự xử lý file, nhưng các thao tác nhập/xuất tới và từ mảng các ký tự hơn là file.

- Header <stdiostream.h> kết hợp kiểu nhập/xuất cũ của C với C++ theo hướng đối tượng.

2. NHẬP XUẤT CƠ BẢN VỚI TOÁN TỬ >> VÀ <<

3. NHẬP KÝ TỰ VÀ CHUỖI KÝ TỰ

Chúng ta nhận thấy toán tử nhập >> chỉ tiện lợi khi dùng để nhập các giá trị số (nguyên, thực). Để nhập ký tự và chuỗi ký tự nên dùng các phương thức sau (định nghĩa trong lớp istream):

```
istream::get();
istream::getline();
istream::ignore();
```

3.1. Phương thức get

Có 3 dạng (thực chất có 3 phương thức cùng có tên get):

Dạng 1: int istream::get() ;
Cách thức đọc của get() có thể minh hoạ qua ví dụ sau:

```
char ch;
ch = cin.get();
```

+ Nếu gõ ABC<Enter>
thì biến ch nhận mã ký tự A, các ký tự BC<Enter> còn lại trên dòng vào.
+ Nếu gõ A<Enter>
thì biến ch nhận mã ký tự A, ký tự <Enter> còn lại trên dòng vào.
+ Nếu gõ <Enter>
thì biến ch nhận mã ký tự <Enter> (bằng 10) và dòng vào rỗng.

Dạng 2: istream& istream::get(char &ch) ;

char được tham chiếu bởi ch.

Chú ý:
+ Cách thức đọc của get dạng 2 cũng giống như dạng 1
+ Do get() dạng 2 trả về tham chiếu tới istream, nên có thể sử dụng các phương thức get() dạng 2 nối đuôi nhau và cũng có thể kết hợp với toán tử >>. Ví dụ:

```
cin.get(ch1); cin.get(ch2); cin.get(ch3);
```

có thể viết chung trên một câu lệnh sau: cin.get(ch1).get(ch2) >> ch3;

Dạng 3: istream& istream::get(char *str, int n, char delim = \n);

Dùng để đọc một dẫy ký tự (kể cả khoảng trắng) và đưa vào vùng nhớ do str trỏ tới. Quá trình đọc kết thúc khi xẩy ra một trong 2 tình huống sau:
+ Gặp ký tự giới hạn (cho trong delim). Ký tự giới hạn mặc định là \n (Enter)
+ Đã nhận đủ (n-1) ký tự

Chú ý:
+ Ký tự kết thúc chuỗi \0 được bổ sung vào dẫy ký tự nhận được
+ ký tự giới hạn vẫn còn lại trên dòng nhập để dành cho các lệnh nhập tiếp theo.
+ Cũng giống như get() dạng 2, có thể viết các phương thức get() dạng 3 nối đuôi nhau trên một dòng lệnh, và cũng có thể kết hợp với toán tử >>
+ Ký tự <Enter> còn lại trên dòng nhập có thể làm trôi phương thức get() dạng 3. Ví dụ xét đoạn chương trình:

```
char ht[25], qq[20], cq[30];
cout << "\nHọ tên: " ;
cin.get(ht,25);
cout << "\nQuê quán: " ;
cin.get(qq,20);
cout << "\nCơ quan: " ;
cin.get(cq,30);
cout <<"\n" <<ht<<" "<<qq<<" "<<cq;
```

Đoạn chương trình dùng để nhập họ tên, quê quán và cơ quan. Nếu gõ:

Pham Thu Huong<Enter>

thì câu lệnh get() đầu tiên sẽ nhận được chuỗi "Pham Thu Huong" cất vào mảng ht. Ký tự <Enter> còn lại sẽ làm trôi 2 câu lệnh get tiếp theo. Do đó câu lệnh cuối cùng sẽ chỉ in ra Pham Thu Huong.
Để khắc phục tình trạng trên, có thể dùng một trong các cách sau:

+ Dùng phương thức get() dạng 1 hoặc dạng 2 để lấy ra ký tự <Enter> trên dòng nhập trước khi dùng get (dạng 3).
+ Dùng phương thức ignore để lấy ra một số ký tự không cần thiết trên dòng nhập trước khi dùng get dạng 3.

cin.ignore(n) ; // Lấy ra (loại ra hay bỏ qua) n ký tự trên dòng nhập.

Như vậy để có thể nhập được cả quê quán và cơ quan, cần sửa lại đoạn chương trình trên như sau:

```
char ht[25], qq[20], cq[30];
cout << "\nHọ tên: " ;
cin.get(ht,25);
cin.get();  // Nhận <Enter>
cout << "\nQuê quán: " ;
cin.get(qq,20);
cin.ignore(1);  // Bỏ qua <Enter>
cout << "\nCơ quan: " ;
cin.get(cq,30);
cout <<"\n" <<ht<<" "<<qq<<" "<<cq;
```

3.2. Phương thức getline

Tương tự như get dạng 3, có thể dùng getline để nhập một dẫy ký tự từ bàn phím. Phương thức này được mô tả như sau:

```
istream& istream::getline(char *str, int n, char delim = \n);
```

Phương thức đầu tiên làm việc như get dạng 3, sau đó nó loại <Enter> ra khỏi dòng nhập (ký tự <Enter> không đưa vào dẫy ký tự nhận được). Như vậy có thể dùng getline để nhập nhiều chuối ký tự (mà không lo ngại các câu lệnh nhập tiếp theo bị trôi).

Ví dụ đoạn chương trình nhập họ tên, quê quán và cơ quan bên trên có thể viết như sau (bằng cách dùng getline):

```
char ht[25], qq[20], cq[30];
cout << "\nHọ tên: " ;
cin.getline(ht,25);
cout << "\nQuê quán: " ;
cin.getline(qq,20);
cout << "\nCơ quan: " ;
cin.get(cq,30);
cout <<"\n" <<ht<<" "<<qq<<" "<<cq;
```

Chú ý: Cũng giống như get() dạng 2 và get() dạng 3, có thể viết các phương thức getline() nối đuôi nhau trên một dòng lệnh hoặc kết hợp với toán tử >>

3.3. Nhập đồng thời giá trị số và ký tự

Như đã nói trong 2, toán tử nhập >> bao giờ cũng để lại ký tự <Enter> trên dòng nhập. Ký tự <Enter> này sẽ làm trôi các lệnh nhập ký tự hoặc chuỗi ký tự bên dưới. Do vậy cần dùng:

hoặc ignore();
hoặc get() dạng 1
hoặc get() dạng 2

để loại bỏ ký tự <Enter> còn lại ra khỏi dòng nhập trước khi thực hiện việc nhập ký tự hoặc chuỗi ký tự.

4.CÁC HÀM THÀNH VIÊN KHÁC

Các hàm thành viên khác của istream

- Hàm ignore dùng để bỏ qua (loại bỏ) một số ký tự trên dòng nhập.

  istream& ignore(int n = 1, int delim = EOF);//bỏ qua đến n ký tự hoặc đến lúc bắt gặp eof.

- Hàm putback():
  istream& putback(char ch);

  Đặt một ký tự ngược lại dòng nhập

- Hàm peek():
  int peek();
  Hàm trả về ký tự kế tiếp mà không trích nó từ dòng.

Các hàm thành viên khác của ostream

- Xuất ký tự bằng hàm thành viên put

  ostream& put(char ch);

Có thể gọi liền ví dụ cout.put('A').put('\n'); khi đó toán tử dấu chấm(.) được tính từ trái sang phải

- Đồng bộ dòng nhập và dòng xuất

Mặc định cin và cout được đồng bộ: std ::cin.tie (&std ::cout); do đó ta thấy những gì được nhập vào bàn phím không cần qua bộ đệm mà hiện ngay lên màn hình.Để đồng bộ các cặp IO khác ta cũng dùng cú pháp instream.tie(&outstream);(instream và outstream là tên stream)

Để bỏ đồng bộ: instream.tie(0);

Nhập xuất không định dạng

Nhập/xuất mức thấp (nghĩa là nhập/xuất không định dạng) chỉ định cụ thể số byte nào đó phải được di chuyển hoàn toàn từ thiết bị tới bộ nhớ hoặc từ bộ nhớ tới thiết bị. Vì không có các xử lý trung gian nên cung cấp tốc độ và dung lượng cao, nhưng cách này không tiện lợi lắm cho lập trình viên.

Nhập/xuất không định dạng được thực hiện với các hàm thành viên istream::read() và ostream::write().

- Hàm istream::read():

  istream& read(unsigned char* puch, int nCount);

  Trích các byte từ dòng cho đến khi giới hạn nCount đạt đến hoặc cho đến khi end- of-file đạt đến. Hàm này có ích cho dòng nhập nhị phân.

- Hàm ostream::write():

ostream& write(const unsigned char* puch, int nCount);

Chèn nCount byte vào từ vùng đệm (được trỏ bởi puch và psch) vào dòng. Nếu file được mở ở chế độ text, các ký tự CR có thể được chèn vào. Hàm này có ích cho dòng xuất nhị phân. Chẳng hạn:

```
char Buff[]="HAPPY BIRTHDAY";
cout.write(Buff,10);
```

- Hàm int istream::gcount() trả về số ký tự đã trích bởi hàm nhập không định dạng cuối cùng.

5.CÁC TRẠNG THÁI DÒNG

Khái niệm cờ: chứa trong một bit, có 2 trạng thái:
Bật (on) có giá trị 1
Tắt (off) có giá trị 0

Mỗi stream lưu giữ những cờ trạng thái cho ta biết thao tác nhập, xuất có thành công hay không, và nguyên nhân gây lỗi. Các cờ này, cũng như cờ định dạng, thực chất là các phần tử của 1 vector bit (1 số nguyên). Chúng bao gồm.

- goodbit: bật khi không có lỗi xảy ra và các cờ khác đều tắt.
- eofbit: bật khi gặp end-of-file.
- failbit: bật khi việc nhập trở nên không chính xác nhưng stream vẫn ổn. Ví dụ như thay vì nhập số nguyên thì người dùng lại nhập ký tự.
- badbit: bật khi bằng cách nào đó stream bị hỏng và mất dữ liệu.

Các cờ trên có thể được truy xuất thông qua các hàm tương ứng: good(), eof(), fail() và bad()
Bạn có thể lấy toàn bộ các cờ bằng hàm ios::iostate rdstate();
Xem ví dụ bên dưới:

```
int x;
cout << "Enter an integer: ";
cin >> x;
// The state of the stream can be gotten with rdstate.
ios::iostate flags = cin.rdstate();
// We can test for which bits are set as follows.
// Note the use of the bitwise & operator.
// It's usually easier to test the bits directly:
if (flags & ios::failbit)
cout << "failbit set." << endl;
else    cout << "failbit not set." << endl;
if (flags & ios::badbit)
   cout << "badbit set." << endl;
else    cout << "badbit not set." << endl;
   if (flags & ios::eofbit)
cout << "eofbit set." << endl;
else    cout << "eofbit not set." << endl;
if (cin.good())
   cout << "Stream state is good." << endl;
else    cout << "Stream state is not good." << endl;
if (cin.fail())
```

```
    cout << "Are you sure you entered an integer?" << endl;
else    cout << "You entered: " << x << endl;
```

Bạn có thể đặt lại các cờ trạng thái bằng phương thức clear():

```
void clear(ios::iostate flags = ios::goodbit );
```

Phương thức này sẽ reset toàn bộ các bit về 0 và bật cờ flags.

VD: cin.clear() sẽ đưa trạng thái dòng về OK, cin.clear(ios::failbit) sẽ bật failbit (xóa những cái khác) còn cin.clear(ios::failbit | ios::badbit) sẽ bật failbit và badbit.

Để bật 1 cờ mà không làm ảnh hưởng đến cờ khác, ta dùng toán tử | với chính vector bit:

```
cin.clear( ios::badbit | cin.rdstate());
```

Hoặc hàm setstate:

```
void setstate( ios::iostate states)
```

Ví dụ:

```
cin.setstate( ios::failbit | ios::badbit)
```

6.ĐỊNH DẠNG XUẤT

6.1. Định dạng giá trị xuất

Định dạng là xác định các thông số:
- Độ rộng quy định
- Độ chính xác
- Ký tự độn
- Và các thông số khác

+ Độ rộng thực tế của giá trị xuất: Như đã nói ở trên, C++ sẽ biến đổi giá trị cần xuất thành một chuỗi ký tự rồi đưa chuỗi này ra màn hình. Ta sẽ gọi số ký tự của chuỗi này là độ rộng thực tế của giá trị xuất.

Ví dụ:

```
int n=4567, m=-23 ;
float  x = -3.1416 ;
char ht[] = “Tran Van Thong” ;
```

Độ rộng thực tế của n là 4, của m là 3, của x là 7, của ht là 14.
+ Độ rộng quy đinh là số vị trí tối thiểu trên màn hình dành để in giá trị. Theo mặc định, độ rộng quy định bằng 0. Chúng ta có thể dùng phương thức cout.width() để thiết lập rộng này. Ví dụ:
cout.width(8);
sẽ thiết lập độ rộng quy định là 8..
+ Mối quan hệ giữa độ rộng thực tế và độ rộng quy định
- Nếu độ rộng thực tế lớn hơn hoặc bằng độ rộng quy định thì số vị trí trên màn hình chứa giá trị xuất sẽ bằng độ rộng thực tế.
- Nếu độ rộng thực tế nhỏ hơn độ rộng quy định thì số vị trí trên màn hình chứa giá trị xuất sẽ bằng độ rộng quy định. Khi đó sẽ có một số vị trí dư thừa. Các vị trí dư thừa sẽ được độn (lấp đầy) bằng khoảng trống.
+ Xác định ký tự độn: Ký tự độn mặc định là dấu cách (khoảng trống). Tuy nhiên có thể dùng phương thức cout.fill() để chọn một ký tự độn khác. Ví dụ:

```
int n=123;    // Độ rộng thực tế là 3
cout.fill(*); // Ký tự độn là *
```

```
cout.width(5); // Độ rộng quy định là 5
cout << n ;
```

thì kết quả in ra là: **123
+ Độ chính xác là số vị trí dành cho phần phân (khi in số thực). Độ chính xác mặc định là 6. Tuy nhiên có thể dùng phương thức cout.precision() để chọn độ chính xác. Ví dụ:

```
float  x = 34.455 ;        // Độ rộng thực tế 6
cout.precision(2) ;        // Độ chính xác 2
cout.width(8);             // Độ rộng quy ước 8
cout.fill(0) ;             // Ký tự độn là số 0
cout << x ;
```

thì kết quả in ra là: 0034.46

6.2. Các phương thức định dạng

6.2.1. Phương thức int cout.width() cho biết độ rộng quy định hiện tại.
6.2.2. Phương thức int cout.width(int n)

Thiết lập độ rộng quy định mới là n và trả về độ rộng quy định trước đó.độ rộng quy định n chỉ có tác dụng cho một giá trị xuất. Sau đó C++ thiết lập lại bằng 0.
Ví dụ:

```
int m=1234, n=56;
cout << "\nAB"
cout.width(6);
cout << m ;
cout << n ;
```

thì kết quả in ra là: B 123456
(giữa B và số 1 có 2 dấu cách).
6.2.3. Phương thức int cout.precision()
Cho biết độ chính xác hiện tại (đang áp dụng để xuất các giá trị thức).
6.2.4. Phương thức int cout.precision(int n)
Thiết lập độ chính xác sẽ áp dụng là n và cho biết độ chính xác trước đó. Độ chính xác được thiết lập sẽ có hiệu lực cho tới khi gặp một câu lệnh thiết lập độ chính xác mới.
6.2.5. Phương thức char cout.fill()
Cho biết ký tự độn hiện tại đang được áp dụng.
6.2.6. Phương thức char cout.fill(char ch)
Quy định ký tự độn mới sẽ được dùng là ch và cho biết ký tự độn đang dùng trước đó. Ký tự độn được thiết lập sẽ có hiệu lực cho tới khi gặp một câu lệnh chọn ký tự độn mới.
Ví dụ :

```
float x=-3.1551, y=-23.45421;
cout.precision(2);
cout.fill('*');
cout << "\n" ;
cout.width(8);
cout << x;
cout << "\n" ;
cout.width(8);
cout << y;
```

Sau khi thực hiện, chương trình in ra màn hình 2 dòng sau:

***-3.16
**-23.45

6.3. Cờ định dạng

6.3.1. Các cờ định dạng

Có thể chia các cờ thành các nhóm:

Nhóm 1 gồm các cờ định vị (căn lề) :

-ios::left: Khi bật cờ ios:left thì giá trị in ra nằm bên trái vùng quy định, các ký tự độn nằm sau

-ios::right: Khi bật cờ ios:right thì giá trị in ra nằm bên phải vùng quy định, các ký tự độn nằm trước.

-ios::internal: Cờ ios:internal có tác dụng giống như cờ ios::right chỉ khác là dấu (nếu có) in đầu tiên

Mặc định cờ ios::right bật.

Nhóm 2 gồm các cờ định dạng số nguyên:

+ Khi ios::dec bật (mặc định): Số nguyên được in dưới dạng cơ số 10
+ Khi ios::oct bật : Số nguyên được in dưới dạng cơ số 8
+ Khi ios::hex bật : Số nguyên được in dưới dạng cơ số 16
Nhóm 3 gồm các cờ định dạng số thực:
ios::fĩxed ios::scientific ios::showpoint
Mặc định: Cờ ios::fixed bật (on) và cờ ios::showpoint tắt (off).
+ Khi ios::fixed bật và cờ ios::showpoint tắt thì số thực in ra dưới dạng thập phân, số chữ số phần phân (sau dấu chấm) được tính bằng độ chính xác n nhưng khi in thì bỏ đi các chữ số 0 ở cuối.
Ví dụ nếu độ chính xác n = 4 thì:

Số thực -87.1500 được in: -87.15
Số thực 23.45425 được in: 23.4543
Số thực 678.0 được in: 678

+ Khi ios::fixed bật và cờ ios::showpoint bật thì số thực in ra dưới dạng thập phân, số chữ số phần phân (sau dấu chấm) được in ra đúng bằng độ chính xác n.
Ví dụ nếu độ chính xác n = 4 thì:

Số thực -87.1500 được in: -87.1500
Số thực 23.45425 được in: 23.4543
Số thực 678.0 được in: 678.0000

+ Khi ios::scientific bật và cờ ios::showpoint tắt thì số thực in ra dưới dạng mũ (dạng khoa học). Số chữ số phần phân (sau dấu chấm) của phần định trị được tính bằng độ chính xác n nhưng khi in thì bỏ đi các chữ số 0 ở cuối.
Ví dụ nếu độ chính xác n = 4 thì:

Số thực -87.1500 được in: -8.715e+01
Số thực 23.45425 được in: 2.3454e+01
Số thực 678.0 được in: 6.78e+02

+ Khi ios::scientific bật và cờ ios::showpoint bật thì số thực in ra dưới dạng mũ. Số chữ số phần phân (sau dấu chấm) của phần định trị được in đúng bằng độ chính xác n.
Ví dụ nếu độ chính xác n = 4 thì:

Số thực -87.1500 được in: -8.7150e+01
Số thực 23.45425 được in: 2.3454e+01
Số thực 678.0 được in: 6.7800e+01

Nhóm 4 gồm các hiển thị:
ios::showpos ios::showbase ios::uppercase
Cờ ios::showpos
+ Nếu cờ ios::showpos tắt (mặc định) thì dấu cộng không được in trước số dương.
+ Nếu cờ ios::showpos bật thì dấu cộng được in trước số dương.
Cờ ios::showbase
+ Nếu cờ ios::showbase bật thì số nguyên hệ 8 được in bắt đầu bằng ký tự 0 và số nguyên hệ 16 được bắt đầu bằng các ký tự 0x. Ví dụ nếu a = 40 thì:

dạng in hệ 8 là: 050
dạng in hệ 16 là 0x28

+ Nếu cờ ios::showbase tắt (mặc định) thì không in 0 trước số nguyên hệ 8 và không in 0x trước số nguyên hệ 16. Ví dụ nếu a = 40 thì:

dạng in hệ 8 là: 50
dạng in hệ 16 là 28

Cờ ios::uppercase
+ Nếu cờ ios::uppercase bật thì các chữ số hệ 16 (như A, B, C, ...) được in dưới dạng chữ hoa.
+ Nếu cờ ios::uppercase tắt (mặc định) thì các chữ số hệ 16 (như A, B, C, ...) được in dưới dạng chữ thường.

6.3.2. Các phương thức bật tắt cờ

Các phương thức này định nghĩa trong lớp ios.
+ Phương thức
long cout.setf(long f) ;
sẽ bật các cờ liệt kê trong f và trả về một giá trị long biểu thị các cờ đang bật. Thông thường giá trị f được xác định bằng cách tổ hợp các cờ trình bầy trong mục 6.1.
Ví dụ:

cout.setf(ios::showpoint | ios::scientific) ;
sẽ bật các cờ ios::showpoint và ios::scientific.

+ Phương thức
long cout.unsetf(long f) ;
sẽ tắt các cờ liệt kê trong f và trả về một giá trị long biểu thị các cờ đang bật. Thông thường giá trị f được xác định bằng cách tổ hợp các cờ trình bầy trong mục 6.1.
Ví dụ:

cout.unsetf(ios::showpoint | ios::scientific) ;
sẽ tắt các cờ ios::showpoint và ios::scientific.

+ Phương thức
long cout.flags(long f) ;
có tác dụng giống như cout.setf(long). Ví dụ:
cout.flags(ios::showpoint | ios::scientific) ;
sẽ bật các cờ ios::showpoint và ios::scientific.
+ Phương thức
long cout.flags() ;

sẽ trả về một giá trị long biểu thị các cờ đang bật.

6.4.Các bộ phận định dạng và các hàm định dạng

6.4.1. Các bộ phận định dạng (định nghĩa trong <iostream.h>)

Các bộ phận định dạng gồm:

```
dec     //như cờ ios::dec
oct     //như cờ  ios::oct
hex     //như cờ  ios::hex
endl    //xuất ký tự \n  (chuyển dòng)
ends    //xuất ký tự \0 (null)
flush   //đẩy dữ liệu ra thiết bị xuất
```

Chúng có tác dụng như cờ định dạng nhưng được viết nối đuôi trong toán tử xuất nên tiện sử dụng hơn.
Ví dụ :

```
cout.setf(ios::showbase)
cout << "ABC" << endl << hex << 40 << " " << 41;
```

Chương trình sẽ in 2 dòng sau ra màn hình:

```
ABC
0x28 0x29
```

6.4.2. Các hàm định dạng (stream manipulator)

Các hàm định dạng gồm:

```
setw(int n)                // như cout.width(int n)
setpecision(int n)         // như cout.pecision(int n)
setfill(char ch)           // như cout. fill(char ch)
setiosflags(long l)        // như cout.setf(long f)
resetiosflags(long l)      // như cout.unsetf(long f)
```

Các hàm định dạng có tác dụng như các phương thức định dạng nhưng được viết nối đuôi trong toán tử xuất nên tiện sử dụng hơn.

Muốn sử dụng các hàm định dạng cần bổ sung vào đầu chương trình chỉ thị #include <iomanip>
Ví dụ có thể thay phương thức cout.setf(ios::showbase) ;
bằng hàm cout << setiosflags(ios::showbase) << "…";

7.CÁC DÒNG CHUẨN

Có 4 dòng tin (đối tượng của các lớp Stream) đã định nghĩa trước, được cài đặt khi chương trình khởi động.

Hai dòng chuẩn quan trọng nhất là:

cin dòng input chuẩn gắn với bàn phím, giống như stdin của C.
cout dòng output chuẩn gắn với màn hình, giống như stdout của C.

Hai dòng tin chuẩn khác:

cerr dòng output lỗi chuẩn gắn với màn hình, giống như stderr của C.
clog giống cerr nhưng có thêm bộ đệm.

Chú ý 1: Có thể dùng các dòng cerr và clog để xuất ra màn hình như đã dùng đối với cout.
Chú ý 2: Vì clog có thêm bộ đệm, nên dữ liệu được đưa vào bộ đệm. Khi đầy bộ đệm thì đưa dữ liệu từ bộ đệm ra dòng clog. Vì vậy trước khi kết thúc xuất cần dùng phương thức: clog.flush()để đẩy dữ liệu từ bộ đệm ra clog.Clog thường được sủ dụng cho các ứng dụng ưu tiên về tốc độ.

Chương trình sau minh hoạ cách dùng dòng clog. Chúng ta nhận thấy, nếu bỏ câu lệnh clog.flush() thì sẽ không nhìn thấy kết quả xuất ra màn hình khi chương trình tạm dừng bởi câu lệnh cin.get()

```
float x=-87.1500;
clog.setf(ios::scientific);
clog.precision(4);
clog.fill('*');
clog.width(10);
clog << x;
clog.flush();
cin.get();
```

Xuất ra máy in

Trong số 4 dòng tin chuẩn không dòng nào gắn với máy in. Như vậy không thể dùng các dòng này để xuất dữ liệu ra máy in. Để xuất dữ liệu ra máy in (cũng như nhập, xuất trên tệp) cần tạo ra các dòng tin mới và cho nó gắn với thiết bị cụ thể.

Để tạo một dòng xuất và gắn nó với máy in ta có thể dùng một trong các hàm tạo sau:

```
ofstream  Tên_dòng_tin(int fd) ;
ofstream  Tên_dòng_tin(int fd, char *buf, int n) ;
```

Trong đó:
+ Tên_dòng_tin là tên biến đối tượng kiểu ofstream hay gọi là tên dòng xuất do chúng ta tự đặt.
+ fd (file descriptor) là chỉ số tập tin. Chỉ số tập tin định sẵn đối với stdprn (máy in chuẩn) là 4.
+ Các tham số buf và n xác định một vùng nhớ n byte do buf trỏ tới. Vùng nhớ sẽ được dùng làm bộ đệm cho dòng xuất.

```
ofstream  prn(4) ;
```

sẽ tạo dòng tin xuất prn và gắn nó với máy in chuẩn. Dòng prn sẽ có bộ đệm mặc định. Dữ liệu trước hết chuyển vào bộ đệm, khi đầy bộ đệm thì dữ liệu sẽ được đẩy từ bộ đệm ra dòng prn. Để chủ động yêu cầu đẩy dữ liệu từ bộ đệm ra dòng prn có thể sử dụng phương thức flush hoặc bộ phận định dạng flush. Cách viết như sau:

```
prn.flush();  // Phương thức
prn << flush ; // Bộ phận định dạng
```

Dùng chuỗi ký tự làm bộ đệm:

```
char buf[1000] ;
ofstream  prn(4,buf,1000) ;
```

Tạo dòng tin xuất prn và gắn nó với máy in chuẩn. Dòng xuất prn sử dụng 1000 byte của mảng buf làm bộ đệm.
Các câu lệnh dưới đây cũng xuất dữ liệu ra máy in:

```
prn << "\nTong = " <<  (4+9) ;  // Đưa dữ liệu vào bộ đệm
prn << "\nTich =  " << (4*9) ;   // Đưa dữ liệu vào bộ đệm
prn.flush() ;  // Xuất 2 dòng (ở bộ đệm) ra máy in
```

Chú ý: Trước khi kết thúc chương trình, dữ liệu từ bộ đệm sẽ được tự động đẩy ra máy in.

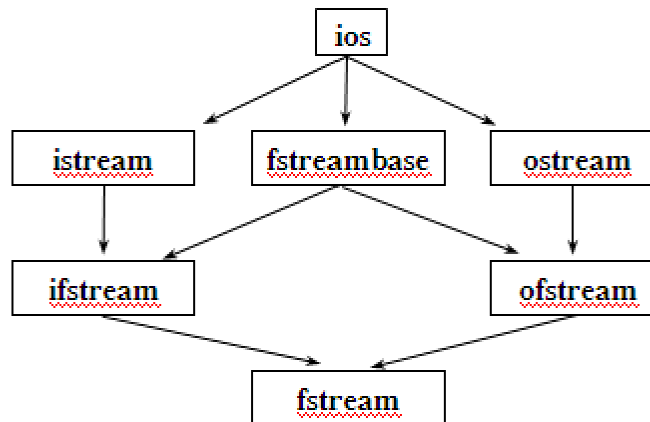
8.THAO TÁC VỚI FILE STREAM

8.1.Các lớp dùng để nhập, xuất dữ liệu lên file

Như đã nói ở trên, C++ cung cấp 4 dòng tin chuẩn để làm việc với bàn phím và màn hình. Muốn nhập xuất lên tệp chúng ta cần tạo các dòng tin mới (khai báo các đối tượng Stream) và gắn chúng với một tệp cụ thể. C++ cung cấp 3 lớp stream để làm điều này, đó là các lớp:

ofstream	dùng để tạo các dòng xuất (ghi tệp)
ifstream	dùng để tạo các dòng nhập (đọc tệp)
fstream	dùng để tạo các dòng nhập, dòng xuất hoặc dòng nhập-xuất

Sơ đồ dẫn xuất các lớp như sau:



8.2.Cách sử dụng fstream :

Để định nghĩa 1 đối tượng file ta dùng cú pháp fstream dataFile;
(ở đây dataFile chỉ là tên do người dùng đặt mà thôi)
Để mở 1 file ta dùng cú pháp sau :

```
dataFile.open("info.txt", ios::out);
```

Hoặc đơn giản truyền tham số vào constructor:

```
fstream dataFile("info.txt", ios::out);
```

Ở đây đòi hỏi 2 đối số : đối thứ nhất là 1 chuỗi tên chứa tên file. Đối thứ 2 là chế độ (mode) mở file và cái này cho ta biết chế độ nào mà chúng ta dùng để mở file. Ở ví dụ trên thì tên file là info.txt còn flag file ở đây là ios::out. Cái này nó nói cho C++ biết chúng ta mở file ở chế độ xuất ra.
Chế độ xuất ra cho phép dữ liệu có thể được ghi vào file.

```
datafile.open("info.txt", ios::in);
```

Còn ở ví dụ này thì tức là ta đang mở file ở chế độ nhập vào, tức là cho phép dữ liệu được đọc vào từ file.
+ Tham số mode có giá trị mặc định là ios::out (mở để ghi). Tham số này có thể là một trong các giá trị sau:

ios::binary	ghi theo kiểu nhị phân (mặc định theo kiểu văn bản)
ios::out	ghi tệp, nếu tệp đã có thì nó bị ghi đè
ios::app	ghi bổ sung vào cuối tệp
ios::ate	chuyển con trỏ tệp tới cuối tệp sau khi mở tệp
ios::trunc	xoá nội dung của tệp nếu nó tồn tại
ios::nocreate	nếu tệp chưa có thì không làm gì (bỏ qua)
ios::noreplace	nếu tệp đã có thì không làm gì (bỏ qua)

Chúng ta thể sử dụng những chế độ trên chung với nhau và chúng sẽ được kết nối với nhau bằng toán tử |.

Ví dụ: fstream dataFile("info.txt", ios::in | ios::out);

Dòng lệnh trên cho phép ta mở file info.txt ở cả 2 chế độ xuất và nhập.
Chú ý : Khi dùng riêng lẻ thì ios::out sẽ xóa nội dung của file nếu file đã được tạo sẵn. Tuy nhiên nếu dùng chung với ios::in, thì nội dung file cũ sẽ được giữ lại. Và nếu file chưa được tạo, nó sẽ tạo ra 1 file mới cho chúng ta luôn.

8.3.Các thap tác cơ bản

–Ghi/đoc file (như cout, cin)
• outClientFile << myVariable
• inClientFile >> myVariable
– Đóng file outClientFile.close();

Bây giờ là 1 ví dụ hoàn chỉnh :

```
// This program uses an fstream object to write data to a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  fstream dataFile;

  cout << "Opening file...\n";
  dataFile.open("demofile.txt", ios::out);   // Mở file để ghi vào
  cout << "Now writing data to the file.\n";
  dataFile << "Jones\n";                     // Ghi dòng thứ 1
  dataFile << "Smith\n";                     // Ghi dòng thứ 2
  dataFile.close();                          // Đóng file
  cout << "Done.\n";
  return 0;
}
```

File Output: J O N E S \n S M I T H H \n <EOF>
Khi file được đóng lại thì kí tự end-of-file sẽ được tự động ghi vào.
Khi file lại được mở ra thì tùy theo mode con trỏ sẽ nằm ở đầu file hoặc vị trí end-of-file đó.

8.4.Kiểm tra file có tồn tại hay không trước khi mở

Đôi khi chúng ta sẽ cần kiểm tra xem file có tồn tại trước khi chúng ta mở nó ra hay không và sau đây là 1 ví dụ:

```
fstream dataFile;
dataFile.open("value.txt", ios::in);
if(dataFile.fail())
{
        //Nếu file không tồn tại, thì tạo ra 1 file mới
         dataFile.open("value.txt", ios::out);
         //...
}
else  dataFile.close();
```

8.5.Cách truyền 1 file vào hàm

Chúng ta khi làm việc với những chương trình thực sự thì đôi khi chúng ta cần phải truyền file stream vào hàm nào đó để tiện cho việc quản lý, nhưng khi truyền phải lưu ý là luôn luôn truyền bằng tham chiếu.

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;
bool OpenFile(fstream &file, char *name);
void ShowContents(fstream &file);

int main()
{
   fstream dataFile;
   if(!OpenFile(dataFile, "demo.txt"))
   {
      cout << "Error !" << endl;
      return 0;
   }
   cout << "Successfully.\n";
   ShowContents(dataFile);
   dataFile.close();

   return 0;
}

bool OpenFile(fstream &file, char *name)
{
   file.open(name, ios::in);
   if(file.fail())
      return false;
   else
      return true;
}

void ShowContents(fstream &file)
{
   string line;
   while(getline(file, line)){
      cout << line << endl;
   }
}
```

8.6. Các hàm định vị cho file tuần tự

• con trỏ vị trí ghi số thứ tự của byte tiếp theo để đọc/ghi
• các hàm đặt lại vị trí của con trỏ:
– seekg (đặt vị trí đọc cho lớp istream)
– seekp (đặt vị trí ghi cho ostream)
– seekg và seekp lấycác đối số là offset và mốc (offset: số byte tương đối kể từ mốc)
•Mốc(ios::beg mặc định)
– ios::beg - đầu file
– ios::cur -vị trí hiện tại
– ios::end -cuối file
• các hàm lấy vị trí hiện tại của con trỏ:
– tellg và tellp
•Ví dụ:

```
fileObject.seekg(0)
//đến đầu file (vị trí 0), mặc định đối số thứ hai là ios::beg
fileObject.seekg(n)
//đến byte thứ n kể từ đầu file
fileObject.seekg(n, ios::cur)
```

```
//tiến n byte
fileObject.seekg(y, ios::end)
//lùi y byte kể từ cuối file
fileObject.seekg(0, ios::cur)
//đến cuối file
//seekp tương tự
location = fileObject.tellg()
//lấy vị trí đọc hiện tại của fileObject
```

8.7.File nhị phân
8.7.1.Định nghĩa:

File nhị phân là file chứa nội dung không nhất thiết phải là ASCII text.

Tất cả những file từ đầu tới giờ chúng ta thao tác đều ở dạng mặc định là text file. Có nghĩa là dữ liệu trong những file này đều đã được định dạng dưới mã ASCII. Thậm chí là số đi chăng nữa khi nó được lưu trong file với toán tử << thì nó đã đc ngầm định chuyển về dạng text. Ví dụ :

```
ofstream file("num.dat");
short x = 1297;
file << x;
```

Dòng lệnh cuối cùng của ví dụ trên sẽ ghi nội dung của x vào file, chúng được lưu vào ở dạng kí tự '1', '2', '9', '7'. Thực sự là con số 1297 được lưu dưới dạng nhị phân, và chiếm 2 byte trong bộ nhớ máy tính.
Vì x kiểu short nó sẽ được lưu như sau :
    00000101 | 00010001
Để ghi trực tiếp các byte chúng ta sẽ dùng mode ios::binary

```
file.open("stuff.dat", ios::out | ios::binary);
```

8.7.2.Hàm write và read :

8.7.1.1.Write

-Hàm write dùng để ghi 1 file stream ở định dạng nhị nhận. Dạng tổng quát của hàm write như sau :

```
fileObject.write(address, size);
```

Ở đây chúng ta có những lưu ý sau :
-fileObject là tên của đối tượng file stream.
-address là địa chỉ đầu tiên của 1 vùng nhớ được ghi vào file. Đối số này có thể là địa chỉ của 1 kí tự hoặc là con trỏ tới kiểu char.
-size là số lượng byte của vùng nhớ mà nó được write. Đối số này bắt buộc phải là kiểu integer(số nguyên dương)
Chúng ta xét ví dụ sau :

```
char letter = 'A';
file.write(&letter, sizeof(letter));
```

-Đối thứ nhất ở đây là địa chỉ của biến letter. Và đối này sẽ nói cho hàm write biết rằng dữ liệu được ghi vào file ở đâu trong vùng nhớ.
-Đối thứ 2 sẽ là kích thước của biến letter, và đối này sẽ nói cho hàm write biết số lượng byte của dữ liệu sẽ ghi vào file. Bởi vì dữ sẽ được lưu khác nhau trên tuỳ hệ thống khác nhau, nên cách tốt nhất là chúng ta dùng toán tử sizeof để quyết định số bytes được ghi. Và sau khi hàm này được thực hiện, nội dung của biến letter sẽ được khi vào file nhị phân của đối tượng "file"đó.

Chúng ta xem tiếp 1 ví dụ sau :

```
char data[] = {'A', 'B', 'C', 'D'};
file.write(data, sizeof(data));
```

Trong ví dụ này thì đối thứ 1 là tên của mảng (data). Vì khi ta truyền tham số là tên của mãng thì tức là ta đã truyền con trỏ trỏ tới vị trí đầu tiên của mãng. Đối thứ 2 cũng có ý nghĩa tượng tự như ví dụ 1. Và sau khi gặp này thực hiện thì nội dung của mãng sẽ được ghi vào file nhị phân tương ứng với đối tượng file.

8.7.1.2.Read

Hàm read thì sẽ dùng đọc vào số dữ liệu nhị phân từ file vào bộ nhớ máy tính. Dạng tổng quát là :
fileObject.read(address, size);
-Ở đây fileObject là tên của đối tượng file stream.
-address là địa chỉ đầu tiên mà vùng nhớ mà dữ liệu được đọc vào được lưu. Và đối này có thể là địa chỉ của 1 kí tự hay 1 con trỏ tới kiểu char.
-size cũng là số lượng byte trong bộ nhớ được đọc vào từ file. Và đối này bắt buộc cũng phải là số kiểu integer (nguyên dương)
.
Tương tự hàm read ta cũng có 2 ví dụ sau :

```
char letter;
file.read(&letter, sizeof(letter));
```

và :

```
char data[4];
file.read(data, sizeof(data));
```

Nếu chúng ta muốn ghi các kiểu khác vào file nhị phân thì ta phải dùng cú pháp có đặt biệt sau đây.

```
reinterpret_cast<dataType>(value)
```

Ở cú pháp trên thì dataType sẽ là kiểu dữ liệu mà chúng ta muốn ép về, và value sẽ là giá trị mà chúng ta muốn ép về dạng byte.
Ví dụ :

```
int x = 65;
file.write(reinterpret_cast<char *>(&x), sizeof(x));
```

Đối với mảng thì :

```
const int SIZE = 10;
int numbers[SIZE] = {1,2,3,4,5,6,7,8,9,10};
file.write(reinterpret_cast<charr *>(numbers), sizeof(numbers));
```

Ví dụ:

```
// This program uses the write and read functions.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  const int SIZE = 10;
  fstream file;
  int numbers[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  // Open the file for output in binary mode.
```

```
        file.open("numbers.dat", ios::out | ios::binary);

        // Write the contents of the array to the file.
        cout << "Writing the data to the file.\n";
        file.write(reinterpret_cast<char *>(numbers), sizeof(numbers));

        // Close the file.
        file.close();

        // Open the file for input in binary mode.
        file.open("numbers.dat", ios::in | ios::binary);

        // Read the contents of the file into the array.
        cout << "Now reading the data back into memory.\n";
        file.read(reinterpret_cast<char *>(numbers), sizeof(numbers));

        // Display the contents of the array.
        for (int count = 0; count < SIZE; count++)
          cout << numbers[count] << " ";
        cout << endl;

        // Close the file.
        file.close();
        return 0;
    }
```

9. ĐỊNH NGHĨA TOÁN TỬ << VÀ >> CỦA BẠN

Như đã nói, nhập xuất trong C++ rất mạnh, nhờ cơ chế đa năng hóa toán tử, C++ cho phép ta định nghĩa nhập xuất đối với các kiểu dữ liệu tự tạo.

Toán tử >> được đa năng hóa có prototype:

```
ostream & operator << (ostream & stream, ClassName Object);
```

Hàm toán tử << trả về tham chiếu chỉ đến dòng xuất ostream. Tham số thứ nhất của hàm toán tử << là một tham chiếu chỉ đến dòng xuất ostream, tham số thứ hai là đối tượng được chèn vào dòng. Khi sử dụng, dòng trao cho toán tử << (tham số thứ nhất) là toán hạng bên trái và đối tượng được đưa vào dòng (tham số thứ hai) là toán hạng bên phải. Ta sẽ sử dụng toán tử này với đối số thứ nhất là ostream nên nó không thể là 1 hàm thành viên.

Còn hàm toán tử của toán tử >> được đa năng hóa có prototype như sau:

```
istream & operator >> (istream & stream, ClassName Object);
```

Hàm toán tử >> trả về tham chiếu chỉ đến dòng nhập istream. Tham số thứ nhất của hàm toán tử này là một tham chiếu chỉ đến dòng nhập istream, tham số thứ hai là đối tượng của lớp đang xét mà chúng ta muốn tạo dựng nhờ vào dữ liệu lấy từ dòng nhập. Khi sử dụng, dòng nhập đóng vai toán hạng bên trái, đối tượng nhận dữ liệu đóng vai toán hạng bên phải. Cũng như trường hợp toán tử <<, hàm toán tử >> không là hàm thành viên của lớp.

Thông thường 2 toán tử này được cấp quyền friend.

Ví dụ với lớp point:

```
#include<iostream>

class point
{
     int x,y;
  public:
     friend ostream & operator << (ostream & Out,const point & P);
```

```
        friend istream & operator >> (istream & In,point & P);
    };
    ostream & operator << (ostream & out,const point & p)
    {
        out << "( "<< p.x << " , " << p.y << " )";
        return out;
        //Cho phép cout << a << b << c;
    }

    istream & operator >> (istream & in,point & p)
    {
        char c;
        in >> p.x >> c >> p.y;
        return in;
    }

    int main()
    {
        point p;
        cin >> p;
        cout << "Toa do: " << p;
        return 0;
    }
```

10.STRING STREAM

Thư viện chuẩn cũng cho phép nhập xuất dữ liệu từ 1 chuỗi ký tự có sẵn trong bộ nhớ.Tính năng này được cung cấp trong header strstream.
Về cách sử dụng, string stream cũng có constructor và các mode như file stream:

```
strstream::strstream( char *, int, ios_base::openmode = ios_base::in|ios_base::out )
```

Đối số thứ nhất là mảng lưu bộ đệm.
Đối số thứ 2 là kích thước tối đa của bộ đệm.
Đối số thứ 3 là mode.

Cũng như file, string stream có 3 loại:

istrstream
ostrstream
strstream

Sau đây là 1 ví dụ về sử dụng osstream để đổi từ 1 số nguyên sang 1 chuỗi:

```
#include<iostream>
#include <strstream>
int main()
{
    long x = 235323429;
    char buf[10];
    ostrstream oss(buf,10,ios::out);
    oss << x << ends;
    //ends đại diện cho ký tự null
    cout << oss.str();//cout << buf;
    cout << "\nSo ky tu la: " << oss.pcount();
    cout << "\nDia chi buffer: " << (void*)oss.rdbuf();
```

```
return 0;
}
```

III. CONTAINER & ITERATOR

1. Tổng quan về container

Container (thùng chứa) là khái niệm chỉ các đối tượng lưu trữ các đối tượng (giá trị) khác. Đối tượng container sẽ cung cấp các phương thức để truy cập các thành phần (element) của nó.

Container nào cũng có các phương thức sau đây:

Phương thức	Mô tả
size()	Số lượng phần tử
empty ()	Trả về 1 nếu container rỗng, 0 nếu ngược lại.
max_size()	Trả về số lượng phần tử tối đa đã được cấp phát
==	Trả về 1 nếu hai container giống nhau
!=	Trả về 1 nếu hai **container** khác nhau
begin()	Trả về **con trỏ đến phần tử** đầu tiên của container
end()	Trả về **con trỏ đến phần tử** cuối cùng của container
front()	Trả về tham chiếu đến phần tử đầu tiên của container
back()	Trả về tham chiếu đến phần tử cuối cùng của container
swap()	Hoán đổi 2 container với nhau (giống việc hoán đổi giá trị của 2 biến)

Các container chia làm 2 loại:

- Sequential container (các ctdl tuần tự) bao gồm list, vector và deque
- Asociative container (các ctdl liên kết) bao gồm map, multimap, set và multiset

Container	Mô tả	Header
Bitset	Một chuỗi bit	**<bitset>**
deque	Hàng đợi	**<queue>**
list	Danh sách tuyến tính	**<list>**
map	Lưu trữ cặp khóa/giá trị mà trong đó mỗi khóa chỉ được kết hợp với 1 giá trị.	**<map>**
multimap	Lưu trữ cặp khóa/giá trị mà trong đó một khóa có thể kết hợp với 2 hay nhiều hơn 2 giá trị.	**<map >**
multiset	Một tập hợp mà trong đó các phần tử có thể giống nhau (theo 1 cách so sánh nào đó)	**<set>**
priority_queue	Một hàng đợi ưu tiên.	**<queue>**
set	Một tập hợp (trong đó mỗi phần tử là duy nhất - theo 1 cách so sánh nào đó)	**<set>**
stack	Một ngăn xếp.	**<stack>**
vector	Mảng động	**<vector>**

Bởi vì tên kiểu sử dụng trong container nằm trong một lớp template khai báo tùy ý, do đó các kiểu này được khai báo **typedef** thành các tên và có ý nghĩa.Các tên này làm cho định nghĩa các container khả chuyển hơn Một vài tên **typedef** phổ biến được đưa ra trong bảng sau:

Typedef Name	**Mô tả**
size_type	Một số nguyên (tương đương **size_t**)
reference	Một tham chiếu đến 1 phần tử
const_reference	Một tham trị đến một phần tử.
iterator	Một biến lặp
const_iterator	Một tham trị lặp
reverse_iterator	Một biến lặp ngược
const_reverse_iterator	Một tham trị lặp ngược
value_type	Kiểu dữ liệu được lưu trữ trong container
allocator_type	Kiểu của allocator.
key_type	Kiểu của khóa.
key_compare	Loại hàm so sánh 2 khóa
value_compare	Loại hàm so sánh 2 giá trị.

Mặc dù không thể xem xét kĩ tât cả các loại container trong chương này, nhưng ở phần sau sẽ nghiên cứu kĩ 3 đại diện: **vector, list, map** và 1 thể hiện hữu dụng của vector là **string.** Một khi bạn hiểu được cách mà những container này làm việc, thì bạn sẽ không gặp khó khăn gì trong việc sử dụng những loại khác.

2. Iterator (bộ lặp)

Là khái niệm sử dụng để chỉ một con trỏ trỏ đến các phần tử trong 1 container. Mỗi container có một loại iterator khác nhau. Trong thư viện STL thì người ta tích hợp lớp đối tượng Iterator cùng với các container. Tư tưởng đó thể hiện như sau:

- Các đối tượng Iterator là các con trỏ đến các đối tượng của lớp lưu trữ:
  ```
  typedef__gnu_cxx::__normal_iterator <pointer,vector_type> iterator;
  ```
- Khai báo lớp Iterator như là 1 lớp nằm trong lớp lưu trữ.
- Xác định trong lớp lưu trữ các phương thức thành phần như:
  - begin() – trả lại con trỏ kiểu đối tượng Iterartor đến phần tử đầu tiên của nằm trong đối tượng lớp lưu trữ.
  - end() – trả lại con trỏ kiểu Iterator trỏ đến 1 đối tượng nào đó bên ngoài tập các phần tử được lưu trữ. Đối tượng bên ngoài nào đó có thể có các định nghĩa khác nhau.Trong trường hợp cụ thể như vector ta có thể hiểu là trỏ đến phần tử sau phần tử cuối cùng.
- Xác định trong lớp đối tượng kiểu Iterator các toán tử như sau:
  - ++p hoặc p++ : chuyển iterator p đến phần tử kế tiếp.
  - --p hoặc p-- : chuyển iterator p đến phần tử đằng trước nó.
  - *p : xác định giá trị của phần tử mà iterator p trỏ đến.

Như bạn biết, mảng và con trỏ có mối quan hệ chặt chẽ với nhau trong C++. Một mảng có thể được truy xuất thông qua con trỏ. Sự tương đương này trong STL là mối quan hệ giữa iterator và container. Nó cung cấp cho chúng ta khả năng xử lý theo chu kì thông qua nội dung của container theo một cách giống như là bạn sử dụng con trỏ để tạo xử lý chu kỳ trong mảng.

Bạn có thể truy xuất đến các thành phần của một container bằng sử dụng một iterator:

```
<container>  coll;
for (<container>::iterator it = coll.begin(); it != coll.end(); ++it)
{
        …*it…
        ……..
}
```

Iterator định nghĩa thế nào là "phần tử đầu", "phần tử cuối", "phần tử tiếp theo" … của container, nó che đi cấu trúc nội tại và cho phép ta viết các đoạn mã tổng quát để duyệt hay chọn phần tử trên các container khác nhau mà không cần biết bên trong của container đó ra sao.

Có 5 loại iterator được mô tả trong bảng dưới.

Iterator	***Quyền truy cập***
Random access (RandIter)	Chứa và nhận giá trị. Các thành phần có thể truy xuất ngẫu nhiên
Bidirectional (BiIter)	Chứa và nhận giá trị. Di chuyển tới trước và sau
Forward (ForIter)	Chứa và nhận giá trị. Chỉ cho phép di chuyển tới.
Input (InIter)	Nhận nhưng không lưu trữ giá trị. Chỉ cho phép di chuyển tới.
Output (OutIter)	Chứa nhưng không nhận giá trị. Chỉ cho phép di chuyển tới.

Nếu container khai báo const, chúng ta phải dùng const_iterator thay vì iterator:

```
const list<string> list1;
list<string>::const_iterator i = list1.begin();
```

STREAM ITERATORS

Stream Iteartor cung cấp khả năng xử lý trên dòng nhập xuất, bạn có thể thêm bớt, xóa sửa trực tiếp trên stream. Một ví dụ là nhập và in ra 1 container không cần vòng for():

```
vector <int> v (istream_iterator <int>(cin), istream_iterator <int>());
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

REVERSE_ITERATOR

Trong các reversible container còn định nghĩa thêm reverse_iterator (iterator đảo ngược). Nó được định vị tuần tự theo một trình tự ngược lại với iterator. Vì vậy, reverse_iterator đầu tiên sẽ trỏ đến cuối của container, tăng giá trị của reverse_iterator sẽ làm nó trỏ đến thành phần đứng trước … Tương ứng với iterator end() và iterator begin() ta có reverse_iterator rbegin() và reverse_iterator rend();

Ví dụ : duyệt list theo 2 chiều

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
   int A[] = {3,2,3,1,2,3,5,3};
   int n = sizeof(A)/sizeof(*A);
   list<int> V;
   for (int i=0; i<n; i++)
      V.push_back(A[i]);

   list<int>::iterator vi;
   cout << endl << "Danh sach theo chieu xuoi" << endl;
   for (vi=V.begin(); vi!=V.end(); vi++)
      cout << *vi << endl;

   list<int>::reverse_iterator rvi;
   cout << endl << "Danh sach theo chieu nguoc" << endl;
   for (rvi=V.rbegin(); rvi!=V.rend(); rvi++)
      cout << *rvi << endl;

   return 0;
}
```

Chuyển đổi qua lại giữa reverse_iterator và iterator:

- Hàm thành viên base(): trả về một iterator trỏ đến phần tử hiện tại của reverse_iterator.
- Tạo reverse_iterator từ iterator: Contructor reverse_iterator(RandomAccessIterator i);

Ví dụ:

```
vector<int> v;
vector<int>::iterator it(v.begin());
vector<int>:: reverse_iterator ri(v.rbegin());
//goi contructor
assert(ri.base()==v.end()-1);
ri=v.begin();
//goi contructor
assert(ri.base()==it);
```

**Lệnh assert(); dùng để kiểm tra một biểu thức điều kiện.*

3. Sequential container

3.1. VECTOR

3.1.1. Giới thiệu :

Lớp mảng động **vector<T>** có sẵn trong thư viện chuẩn STL của C++ định nghĩa một mảng động các phần tử kiểu T, vector có các tính chất sau:

- Không cần khai báo kích thước của mảng, vector có thể tự động cấp phát bộ nhớ, bạn sẽ không phải quan tâm đến quản lý kích thước của nó.
- Vector còn có thể cho bạn biết số lượng các phần tử mà bạn đang lưu trong nó.
- Vector có các phương thức của stack, được tối ưu hóa với các phép toán ở phía đuôi (rear operations)

- Hỗ trợ tất cả các thao tác cơ bản như chèn ,xóa, sao chép ..

3.1.2. Cú pháp

Để có thể dùng vector thì bạn phải thêm 1 header #include <vector> và phải có using std::vector;
Cú pháp của vector cũng rất đơn giản, ví dụ :

```
vector<int> v ;
vector<int> v(10);
vector<int> v(10, 2);
```

Khai báo vector v có kiểu int. Chú ý kiểu của vector được để trong 2 dấu ngoặc nhọn.

Dạng 1 khởi tạo 1 vector có kích thước ban đầu là 0, vì kích thước của vector có thể nâng lên, cho nên không cần khai báo cho nó có bao nhiêu phần tử cũng được. Hoặc nếu muốn thì bạn cũng có thể khai báo như dạng 2, nhưng cũng nhấn mạnh lại rằng mặc dù size = 10, nhưng khi bạn thêm vào hoặc xoá bớt đi thì kích thước này cũng vẫn thay đổi được.
Trong dạng 3 thì 10 phần tử của vector A sẽ được khởi tạo bằng 2.

Đồng thời ta cũng có thể khởi tạo cho 1 vector sẽ là bản sao của 1 hoặc 1 phần vector khác, ví dụ :

```
vector<int> A(10,2);
vector<int> B(A);
vector<int> C(A.begin(), A.begin() + 5 );//ban sao 5 phan tu dau tien
```

Hãy theo dõi ví dụ sau:

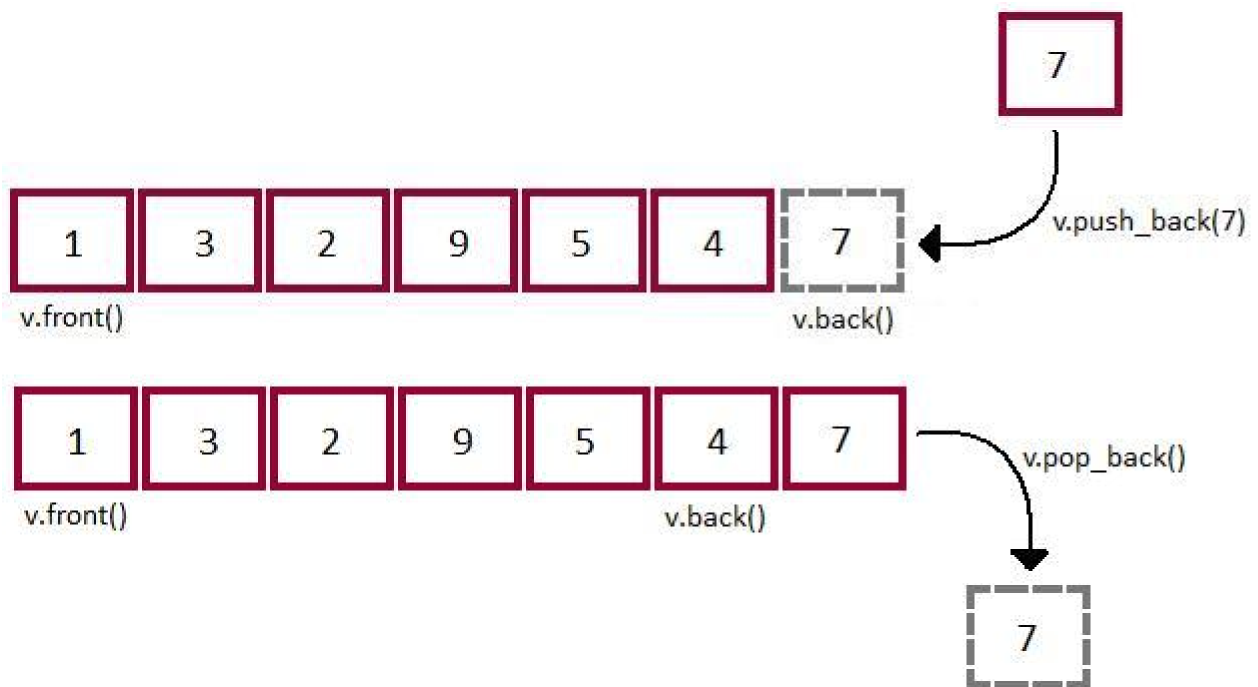
```
#include <iostream>
#include <vector>

using namespace std;
int main()
{
    vector<int> V(3);
    V[0] = 5;
    V[1] = 6;
    V[2] = 7;
    for (int i=0; i<V.size(); i++)
        cout << V[i] << endl;
    return 0;
}
```

Ví dụ trên cho bạn thấy việc sử dụng vector rất đơn giản, hoàn toàn giống với mảng nhưng bộ nhớ được quản lý tự động, bạn không phải quan tâm đến giải phóng các vùng bộ nhớ đã xin cấp phát.

Trường hợp xác định kích thước mảng khi chương trình đang chạy, chúng ta dùng hàm dựng mặc định để khai báo mảng chưa xác định kích thước, sau đó dùng phương thức **resize(size _t n)** để xác định kích thước của mảng khi cần.

3.1.3. Các phương thức

3.1.3.1. Các phương thức của stack: push_back() và pop_back()




```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   int i;
   vector<int> V;
   for (i=0; i<5; i++) // Lặp 5 lần, mỗi lần đưa thêm 1 số vào vector
      V.push_back(i); // Như vậy, vector có thể được sử dụng như stack
   cout << endl << "Mang ban dau:" << endl;
   for (i=0; i<V.size(); i++) // Ghi lại nội dung của mảng ra màn h.nh
      cout << V[i] << endl;
   V.pop_back( ); // Xóa phần tử vừa chèn vào đi
   cout << endl << "Xoa phan tu cuoi:" << endl;
   for (i=0; i<V.size(); i++) // In nội dung của vector sau khi xóa
      cout << V[i] << endl;
   return 0;
}
```

Với ví dụ trên, bạn có thể thấy ta có thể sử dụng vector như 1 stack:

- Không nên dùng toán tử [] để truy xuất các phần tử mà nó không tồn tại, nghĩa là ví dụ vector size = 10, mà bạn truy xuất 11 là sai. Để thêm vào 1 giá trị cho vector mà nó không có size trước hoặc đã full thì ta dùng hàm thành viên push_back(), hàm này sẽ thêm 1 phần tử vào cuối vector.

- Tương tự với thao tác xóa một phần tử ở cuối ra khỏi vector, bạn cũng chỉ cần sử dụng 1 lệnh: pop_back( )

3.1.3.2. Xóa tại vị trí bất kỳ, xóa trắng

Ví dụ:

```
#include <iostream>
#include <vector>
using namespace std;
template <class T>
void print(const vector<T>&v)
{
   for (int i=0; i < v.size(); i++)
      cout << v[i] << endl;
}
int main()
{
   char *chao[] = {"Xin", "chao", "tat", "ca", "cac", "ban"};
   int n = sizeof(chao)/sizeof(*chao);
   vector<char*> v(chao, chao + n);
   //đây là 1 cách khởi tạo vector

   cout << "vector truoc khi xoa" << endl;
   print(v);
   v.erase(v.begin()+ 2, v.begin()+ 5);
   //xóa từ phần tử thứ 2 đến phần tử thứ 5

   v.erase( v.begin()+1 );
   //xóa phần tử thứ 1
   cout << "vector sau khi xoa" << endl;
   print(v);
   v.clear();//Xóa toàn bộ các phần tử
   cout << "Vector sau khi clear co "
      << v.size() << " phan tu" << endl;
   return 0;
}
```

3.1.3.3. Phương thức chèn

```
iterator insert ( iterator position, const T& x );
void insert ( iterator position, size_type n, const T& x );
void insert ( iterator position, InputIterator first, InputIterator last );
```

Ví dụ:

```
// inserting into a vector
#include <iostream>
#include <vector>
using namespace std;
```

```
int main ()
{
    vector<int> v1(4,100);
        v1.insert ( v1.begin()+3 , 200 );  //chèn 200 vào trước vị trí thứ 3
        v1.insert ( v1.begin()+2 ,2,300); //chèn 2 lần 300 vào trước vị trí thứ 2
    vector<int> v2(2,400);
        int a [] = { 501, 502, 503 };
        v1.insert (v1.begin()+2, a, a+3); //chèn mảng a (3 phần tử) vào trước vị trí thứ 2
        v1.insert (v1.begin()+4,v2.begin(),v2.end());//chèn v2 vào trước vị trí thứ 4
        cout << "v1 contains:";
    for (int i=0; i < v1.size(); i++)
            cout << " " << v1[i];
    return 0;
}
```

Output:

```
v1 contains: 100 100 501 502 400 400 503 100 200 300 300 100
```

3.1.3.4. Một số hàm khác và chức năng

Những toán tử so sánh được định nghĩa cho vector: ==, <, <=, !=, >, >=

Tham chiếu back(), front()

```
template<class _TYPE, class _A>
reference vector::front( );
template<class _TYPE, class _A>
reference vector::back( );
```

Trả về tham chiếu đến phần tử đầu và cuối vector: v.front() ⇔ v[0] và v.back() ⇔ v[v.size()-1]

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

int main ()
{
```

```
    int a[] = {3,2,3,1,2,3,5,7};
    int n = sizeof(a)/sizeof(*a);
    vector<int> v(a, a+n);

    cout << "phan tu dau la " << v.front() << endl;
    cout << "phan tu cuoi la " << v.back() << endl;
    cout << "gan phan tu cuoi la 9 ..." << endl;
    v.back() = 9;
    cout << "gan phan tu dau la 100 ..." << endl;
    v.front() = 100;

    cout << "kiem tra lai vector: ";
    for (int i=0; i < v.size(); i++)
       cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

Output:

```
phan tu dau la 3

phan tu cuoi la 7

gan phan tu cuoi la 9 ...

gan phan tu dau la 100 ...

kiem tra lai vector: 100 2 3 1 2 3 5 9

Press any key to continue …
```

Hàm thành viên empty()

Để xác định vector có rỗng hay không ta dùng hàm thành viên empty(), hàm này trả về true nếu vector rỗng, và false ngược lại. Cú pháp :

```
 if(v.empty() == true) {
       cout << "No values in vector \n";
}
```

- **capacity()** : Trả về số lượng phần tử tối đa mà vector được cấp phát, đây là 1 con số có thể thay đổi do việc cấp phát bộ nhớ tự động hay bằng các hàm như reserve() và resize()

Sự khác biệt giữa 2 hàm **size()** và **capacity()** :

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

```
int main(int argc , char **argc)
{
    vector<int >so1,so2[10];
    so1.reserve(10);
    cout <<"Kich thuoc toi da:"<<so1.capacity();

    cout <<"\n Kich thuoc hien tai cua mang 2 "<<so2.size()<<endl;
    return 0 ;
}
```

- **reserve()**: cấp phát vùng nhớ cho vector, giống như **realloc()** của C và không giống **vector::resize(),** tác dụng của **reserve** để hạn chế vector tự cấp phát vùng nhớ không cần thiết.Ví dụ khi bạn thêm 1 phần tử mà vượt quá capacity thì vector sẽ cấp phát thêm, việc này lặp đi lặp lại sẽ làm giảm performance trong khi có những trường hợp ta có thể ước lượng được cần sử dụng bao nhiêu bộ nhớ.

Ví dụ nếu ko có **reserve()** thì capacity sẽ là 4 :

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

using namespace std;

int main()
{
   vector< int > my_vect;
   my_vect.reserve( 8 );
   my_vect.push_back( 1 );
   my_vect.push_back( 2 );
   my_vect.push_back( 3 );

   cout << my_vect.capacity() << "\n";
    return 0;
}
```

- **swap();** hoán đổi 2 container với nhau (giống việc hoán đổi giá trị của 2 biến). Ví dụ : v1.swap(v2);

3.1.4. Kiểm tra tràn chỉ số mảng

Có một vấn đề chưa được đề cập đến từ khi ta làm quen với vector, đó là khả năng kiểm tra tràn chỉ số mảng (range check), để biết về khả năng này, chúng ta lại tiếp tục với một ví dụ mới:

```
#include <iostream>
#include <vector>
#include <conio.h>
using namespace std;
int main()

{

        try { // sử dụng try...catch để bẫy lỗi

                vector<long> V(3, 10); // Khởi tạo vector gồm 3 thành phần
                // Tất cả gán giá trị 10
```

```
            cout << "V[0]=" << V[0] << endl; // Đưa thành phần 0 ra màn hình
            cout << "V[1]=" << V[1] << endl; // Đưa thành phần 1 ra màn hình
            cout << "V[2]=" << V[2] << endl; // Đưa thành phần 2 ra màn hình
            cout << "V[3]=" << V[3] << endl; // Thành phần 3 (lệnh này hoạt động không
            // đúng vì V chỉ có 3 thành phần 0,1,2
            cout << "V[4]=" << V[4] << endl; // Thành phần 4 (càng không đúng)
            // Nhưng 2 lệnh trên đều không gây lỗi
            cout << "V[0]=" << V.at(0) << endl; // Không sử dụng [], dùng phương thức at
            cout << "V[1]=" << V.at(1) << endl; // Thành phần 1, OK
            cout << "V[2]=" << V.at(2) << endl; // Thành phần 2, OK
            cout << "V[3]=" << V.at(3) << endl; // Thành phần 3: Lỗi, chương trình dừng
            cout << "V[4]=" << V.at(4) << endl;

            getchar();

        } catch (exception &e) {

            cout << "Tran chi so ! " << endl;

    }

    return  0;

}
```

Trong ví dụ này, chúng ta lại có thêm một số kinh nghiệm sau:

- Nếu sử dụng cú pháp biến_vector[chỉ_số], chương trình sẽ không tạo ra lỗi khi sử dụng chỉ số mảng nằm ngoài vùng hợp lệ (giống như mảng thường). Trong ví dụ, chúng ta mới chỉ lấy giá trị phần tử với chỉ số không hợp lệ, trường hợp này chỉ cho kết quả sai. Nhưng nếu chúng ta gán giá trị cho phần tử không hợp lệ này, hậu quả sẽ nghiêm trọng hơn nhiều vì thao tác đó sẽ làm hỏng các giá trị khác trên bộ nhớ.

- Phương thức at(chỉ_số) có tác dụng tương tự như dùng ký hiệu [], nhưng có một sự khác biệt là thao tác này có kiểm tra chỉ số hợp lệ. Minh chứng cho nhận xét này trong ví dụ khi chương trình chạy đến vị trí lệnh V.at(3), lệnh này không cho ra kết quả mà tạo thành thông báo lỗi.

3.1.5 Mảng 2 chiều với Vector

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

using namespace std;

int main()
{
    vector< vector<int> > matrix(3, vector<int>(2,0));

        //chu y viet > > de khong nham voi toan tu >>
    for(int x = 0; x < 3; x++)
     for(int y = 0; y < 2; y++)
            matrix[x][y] = x*y;

    for(int x = 0; x < 3; x++)
    for(int y = 0; y < 2; y++)
            cout << matrix[x][y];
```

```
    cout << '\n';

    return 0;
}
```

Ví dụ này minh họa việc sử dụng mảng 2 chiều, thực chất đây là một vector của vector. Mảng 2 chiều sử dụng biện pháp này có thể có kích thước khác nhau giữa các dòng (ví dụ mảng 2 chiều là nửa trên của ma trận)

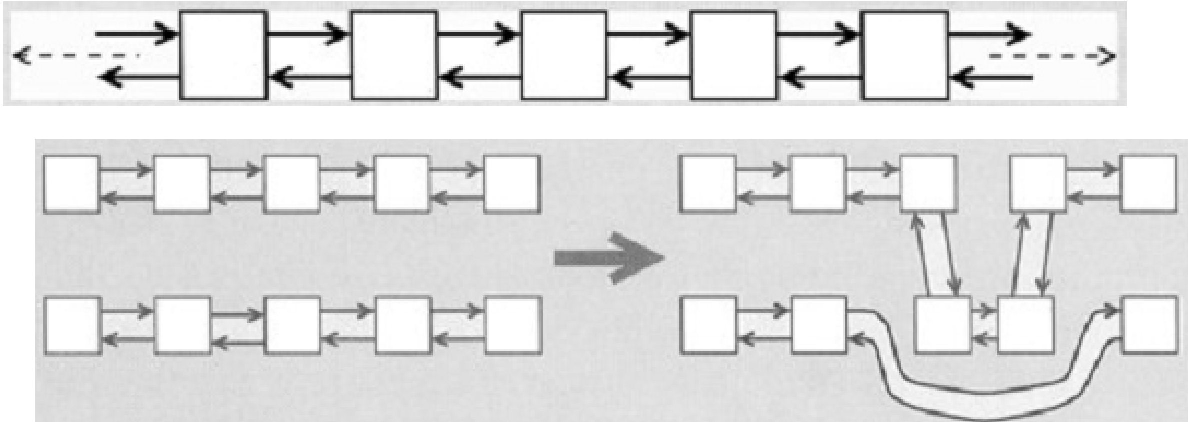
Bảng: các hàm thành viên lớp vector

Hàm thành phần	Mô tả
template<class InIter> void assign(InIter start, InIter end);	Gán giá trị cho vector theo trình tự từ start đến end.
Template<class Size, class T) Void assign(Size num, const T &val = T());	Gán giá trị của val cho num phần tử của vector.
Reference at(size_type I); Const_reference at(size_type I) const;	Trả về một tham chiếu đến một phần tử được chỉ định bởi i.
Reference back(size_type I); Const_reference at(size_type I) const;	Trả về một tham chiếu đến phần tử cuôi cùng của vector.
Iterator begin(); Const _iterator begin() const;	Trả về một biến lặp chỉ định phần tử đầu tiên của vector.
Size_type capacity() const;	Trả về dung lượng hiện thời của vector. Đây là số lượng các phần tử mà nó có thể chứa trước khi nó cần cấp phát thêm vùng nhớ.
Void clear();	Xóa tất cả các phần tử trong vector.
Bool empty() const;	Trả về true nếu vector rỗng và trả về false nếu ngược lại.
Iterator end(); Const_iterator end() const	Trả về một biến lặp để kết thúc một vector.
iterator erase(iterator i);	Xóa một phần tử được chỉ bởi i. Trả về một biến lặp chỉ đến phần tử sau phần tử được xóa.
Iterator erase(iterator start, iterator end);	Xóa những phần tử trong dãy từ start đến end. Trả về một biến lặp chỉ đến phần tử sau cùng của vector.

Reference front(); Const_reference front() const;	Trả về một tham chiếu đến phần tử đầu tiên của vector.
Allocator_type get_allocator() const;	Trả về vùng nhớ được cấp phát cho vector.
Iterator insert(iterator I, const T&val=T());	Chèn val trực tiếp vào trước thành phần được chỉ định bởi i. biến lặp chỉ đến phần tử được trả về.
Void insert(iterator I, size_type num, const T& val);	Chèn num val một cách trực tiếp trước phần tử được chỉ định bởi i.
Template<class InIter> Void insert(iterator I, InIter start, InItr end);	Chèn chuỗi xác định từ start đến end trực tiếp trước một phần tử được chỉ định bởi i.
Size_type max_size() const;	Trả về số lượng phần tử lớn nhất mà vector có thể chứa.
Reference operator[](size_type i) const; Const_reference operator[](size_type i) const;	Trả về một tham chiếu đến phần tử được chỉ định bởi i.
Void pop_back();	Xóa phần tử cuối cùng trong vector.
Void push_back(cons T&val);	Thêm vào một phần tử có giá trị val vào cuối của vector.
Reverse_iterator rbegin(); Const_reverse_iterator rbegin() const;	Trả về biến lặp nghịch chỉ điểm kết thúc của vector.
Reverse_iterator rend(); Const_reverse_iterator rend() const;	Trả về một biến lặp nghịch chỉ điểm bắt đầu của vector.
Void reverse (size_type num);	Thiết lập kích thước của vector nhiều nhất là bằng num.
Void resize (size_type num, T val =T());	Chuyển đổi kích thước của vector được xác định bởi num. Nếu như kích thước của vector tăng lên thì các phần tử có giá trị val sẽ được thêm vào cuối vector.
Size_type size() const;	Trả về số lượng các phần tử hiện thời của trong vector.
Vois swap(vector<T, Allocator>&ob)	Chuyển đổi những phần tử được lưu trong vector hiện thời với những phần tử trong ob.

3.2. LIST

List trong STL là danh sách liên kết đôi. Không giống như vector, hỗ trợ truy xuất một cách ngẫu nhiên (random access), một danh sách chỉ có thể được truy xuất một cách tuần tự. Nghĩa là nếu bạn muốn truy xuất một phần tử bất kì trong list thì bạn phải bắt đầu duyệt từ phần tử đầu tiên hoặc phần tử cuối cùng của list rồi duyệt lần lượt qua các iterator đến phần tử đó.



Để sử dụng list, bạn phải khai báo file header list: #include <list>

List có thể khởi tạo bằng constructor mặc định hoặc sao chép từ mảng, từ list khác hay container khác

```
int a[10];
list<int> list0;
list<int> list1(a+2,a+7);
list<int> list2(list1.begin()++,--list1.end());
```

Các hàm thường dùng của list

Phương thức	Mô tả
size()	Số lượng phần tử của list
empty ()	Trả về 1 nếu danh sách là trống, 0 nếu ngược lại
max_size()	Trả về số lượng phần tử tối đa của list
front()	Trả về phần tử đầu tiên của list
back()	Trả về phần tử cuối cùng của list
begin()	Trả về phần tử lặp đầu tiên của danh sách
end()	Trả về phần tử lặp cuối cùng của danh sách
sort()	Sắp xếp danh sách với toán tử <
push_front()	đưa một phần tử vào đầu list
push_end()	đưa một phần tử vào cuối list

pop_front()	gỡ phần tử đầu list ra
pop_end()	gỡ phần tử cuối list ra

Vi dụ dưới chúng ta tạo một list, đưa phần tử vào và truy xuất phần tử

```
list<string> list1;
list1.push_back("Zebra");list1.push_back("Penguin");list1.push_front("Lion");
list<string>::iterator i;
for(i=list1.begin();i!=list1.end();++i) cout<<*i<<endl;
```

Nếu chúng ta muốn một list chứa nhiều list, ta chỉ cần khai báo

```
list<list<string> > listOfList;
```

Các hàm thường dùng khác của list:

```
list1.insert(list1.begin(),"Seadog");//chèn phần tử "Seagon" vào vị trí đầu list
list1.insert(++list1.begin(),2,"Seadog");//chèn phần tử "Seagon" vào một vị trí cụ thể
list1.erase(list1.begin());//xóa một phần tử ở một vị trí cụ thể
list1.erase(++list1.begin(),3);//xóa 3 phần tử bắt đầu từ một vị trí cụ thể
list1.clear();//xóa tất cả các phần tử
list1.remove("Zebra");//tìm kiếm và xóa phần tử "Zebra"
list1.reverse();//sắp xếp giảm dần (descending)
list1.resize(int);//thiết lập số phần tử mới của list
iterator i=list1.find(++list1.begin(),--list1.end(),"Penguin");

//tìm kiếm phần tử "Penguin", bắt đầu ở một vị trí cụ thể kết thúc ở một vị trí cụ thể khác

// trả về iterator trỏ đến phần tử này. Nếu không tìm thấy, hàm này trả về vị trí kết thúc, ở đây là --
list1.end()
```

Các hàm với hai list

```
list1.splice(--list1.end(),list2,list2.begin());
//splice(cut và paste) một phần tử từ vị trí list2.begin() của list2 đến vị trí --list1.end() của list1
list1.splice(--list1.end(),list2);
//splice(cut và paste) tất cả phần tử của list2 đến vị trí --list1.end() của list1
list1.merge(list2);
// Trộn danh sách list1 với danh sách list2 với điều kiện cả 2 danh sách đều đã sắp xếp với toán
tử <
//Danh sách được trộn cũng sẽ được sắp xếp với toán tử <.
//Độ phức tạp là tuyến tính)
list2.swap(list1);//swap 2 list, nghĩa là temp = list2; list2 = list1; list1 = temp;
```

Bảng: các hàm thành viên lớp list

Template<class InIter> Void assign(InIter start, InIter end);	Gán giá trị cho danh sách theo trình tự từ bắt đầu đến kết thúc.
Template<class Size, class T>	Gán *num* phần tử của danh sách bằng giá trị *value*.

Void assign (Size num, const T&val=T());	
Reference back();	Trả về một tham chiếu đến phần tử cuối cùng trong danh sách. (Chỉ đến phần tử cuối cùng của danh sách.)
Const_reference back()const; Iterator begin();	Trả về một biến lặp chỉ đến phần tử đầu tiên của danh sách. (Chỉ đến phần tử đầu tiên danh sách.)
Void clear();	Xóa tất cả các phần tử trong danh sách.
Bool empty()const;	Trả về *true* nếu danh sách rỗng, và *false* nếu ngược lại.
Iterator end(); Const_iterator end() const;	Trỏ đến cuối danh sách.
Iterator erase(iterator i);	Xóa phần tử được chỉ định bởi *i.* Chỉ đến phần tử kế sau phần tử đã được xóa.
Iterator erase(iterator start, iterator end);	Xóa những phần tử từ vị trí *start* đến *end.* Chỉ đến phần tử kế sau phần tử cuối cùng được xóa.
Reference front(); Const_referencefront() const;	Trả về một tham chiếu chỉ đến phần tử đầu tiên của danh sách.
Allocator_type get_allocator() const;	Trả về vùng nhớ được cấp phát cho danh sách.
Iterator insert(iterator i, const T &val=T());	Chèn giá trị *val* một cách trực tiếp vào trước phần tử được chỉ định bởi *i.* Trả về một biến lặp chỉ đến phần tử này.
Void insert(iterator I, size_type num, const T&val=T());	Chèn ***num*** giá trị ***val*** trực tiếp trước phần tử được chỉ định bởi ***i.***
Template<class InIter> Void insert(iteratir i, InIter start, InIter end);	Chèn theo trình tự được xác định từ ***start*** đến ***end*** trực tiếp trước phần tử được chỉ định bởi ***i.***
Size_type max_size max_size() const;	Trả về số lượng các phần tử mà danh sách có thể giữ.
Void merge(list<T. allocator>&ob); Template<class Comp> Void merge(<list<T, Allocator>&ob, Comp cmpfn);	Trộn danh sách có thứ tự được chứa trong **ob** với một danh sách có thứ tự đang gọi thực hiện phương thức. Kết quả là một danh sách có thứ tự. Sau khi trộn, danh sách chứa trong **ob** là rỗng. Ở dạng thứ 2, một hàm so sánh được định nghĩa để xác định giá trị của một phần tử thì nhỏ hơn giá trị một phần tử khác.
Void pop_back();	Xóa phần tử cuối cùng trong danh sách.
Void pop_front();	Xóa phần tử đầu tiên trong danh sách.
Void push_back(const T&val);	Thêm một phần tử có giá trị được xác định bởi ***val*** vào

	cuối danh sách.
Void push_front(const T&val);	Thêm một phần tử có giá trị được xác định bởi ***val*** vào đầu danh sách.
Reverse_iterator rbegin(); Const_reverse_iterator rbegin() const;	Trả về một biến lặp ngược chỉ đến cuối danh sách.
Void remove(const T&val);	Xóa những phần tử có giá trị ***val*** trong danh sách.
Template <class UnPred> Void remove_if(UnPred pr);	Xóa những mà phần tử nếu như ***pr*** là true.
Reverse_iterator rend(); Const_reverse_iterator rend() const;	Trả về một biến lặp ngược chỉ đến đầu danh sách.
Void resize(isze_type num, T val=T());	Thay đổi kích thước của danh sách được xác định bởi ***num.*** Nếu như danh sách dài ra thêm, thì những phần tử có giá trị ***val*** được thêm vào cuối.
Void reverse();	Đảo ngược danh sách đang gọi thực hiện phương thức.
Size_type size() const;	Trả về số lượng các phần tử hiện tại trong mảng.
Void sort (); Template <class Comp> Void sort(conm cmpfn);	Sắp xếp danh sách. Dạng thứ 2 sử dụng hàm so sánh ***cmpfn*** để xác định giá trị của phần tử này có nhỏ hơn giá trị của phần tử kia không.
Void spilice(iterator i, list <T, allocator> &ob);	Chèn nội dung của ***ob*** vào trong danh sách đang gọi thực hiện phương thức tại vị trí được chỉ định bởi ***i.*** Sau thao tác này ***ob*** là rỗng.
Void splice(iterator i, list <allocator> &ob, iterator el);	Xóa phần tử được chỉ định bởi ***el*** trong danh sách ***ob*** và lưu trữ nó vào trong danh sách đang gọi thực hiện phương thức tại vị trí được chỉ định bởi ***i.***
Void splice(iterator I, list<T, Allocator>&ob, iterator start, iterator end).	Xóa một mảng phần tử bắt đầu từ vị trí ***start*** đến vị trí ***end*** và lưu nó vào trong danh sách đang gọi thực hiện phương thức bắt đầu từ vị trí ***i.***
Void swap(list<T, Allocator> &ob);	Hoán vị nội dung những phần tử được lưu trữ trong danh sách đang gọi thực hiện phương thức với những phần tử trong ***ob.***
Void unique(); Template<class BinPred> Void unique(BinPred pr);	Xóa những phần tử trùng trong danh sách đang gọi thực hiện phương thức. Dạng 2 sử dụng ***pr*** để định nghĩa sự duy nhất.

3.3. DEQUE #include <deque>

deque giống list ngoại trừ:
-cho phép random access, với operator[], nghĩa là d[5], d[6], etc như mảng
-được tối ưu hóa với các phép toán ở phía đầu (front operations)
-không có sẵn các hàm splice,sort, merge, reserve.

Ví dụ:

```
deque<int> d;
d.push_front(5);
cout<<d[0];
```

3.4. STRING

3.4.1. Giới thiệu :

Các chương trình C++ có thể sử dụng chuỗi theo cách thức cũ của *Ngôn ngữ C*: mảng các ký tự kết thúc bởi ký tự mã ASCII là 0 (ký tự **null**) cùng với các hàm thư viện khai báo trong <string.h> . Những bất tiện khi dùng theo cách này chủ yếu đến từ việc quản bộ nhớ động và ký tự **null**, truyền và trả lại giá trị cho hàm.

Thư viện chuẩn cung cấp kiểu string (xâu ký tự) , giúp các bạn tránh khỏi hoàn toàn các phiền phức nêu trên. String thực chất là một vector<char> có bổ sung thêm một số phương thức và thuộc tính, do đó, nó có toàn bộ các tính chất của 1 vector, vd hàm size(), push_back(), toán tử [] …

Các chỉ thị #include cần khai báo để sử dụng string :

```
#include <string>
using std::string;
//using namespace std;
```

3.4.2. Sử dụng :

3.4.2.1. Các phép toán và phương thức cơ bản:

- Các toán tử +, += dùng để ghép hai chuỗi và cũng để ghép một ký tự vào chuỗi;
- Các phép so sánh theo thứ tự từ điển: == (bằng nhau), != (khác nhau), > (lớn hơn), >= (lớn hơn hay bằng), < (nhỏ hơn), <= (nhỏ hơn hay bằng);
- Phương thức length( ) và phép lấy chỉ số [ ] để duyệt từng ký tự của chuỗi: nếu s là biến kiểu string thì s[i] là ký tự thứ i của s với 0 ≤ i <s.length( );
- Phép gán (=) dùng để gán biến kiểu string bằng một chuỗi, hoặc bằng string khác, chẳng hạn: string s=”ABCDEF”; hay s1=s2; mà không cần copy xâu.
  Những constructor thường sử dụng nhất:

```
string();
string(const char *str);
string(const string & str);
```

- Có thể dùng toán tử << với cout để xuất một chuỗi ra màn hình hoặc dùng toán tử >> với cin để nhập một chuỗi ký tự đến khi gặp một khoảng trống thì dừng.

```
char st[]="ABCDEF";
string s;
s="XYZ";
cout << s << endl;
```

```
s=st;
cout << s.length() << " : " << s << endl;
//…
```

Nhập một string: istream& getline (istream& in, string& str, char delimiter = '\n');

Đọc 1 dòng văn bản từ đối tượng nhập (istream) in (có thể là file hay đối tượng chuẩn cin) từng ký tự đến khi ký tự delimiter được nhập vào (mặc định là \n) (thường được dùng thay cho cin >> khi nhập chuỗi có ký tự space).Có thể dùng kết hợp với toán tử >>

```
// getline with strings
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  short age;
  cout << "Please enter full name and age"<< endl;
  getline( cin, str) >> age;
  cout << "Thank you " << str << "!\n";

return 0;
}
```

3.4.2.2. Các phương thức chèn, xóa, lấy chuỗi con

- Phương thức substr(int pos, int nchar) trích ra chuỗi con của một chuỗi cho trước, ví dụ str.substr(2,4) trả về chuỗi con gồm 4 ký tự của chuỗi str kể từ ký tự ở vị trí thứ 2 (ký tự đầu tiên của chuỗi ở vị trí 0).

```
//get substring
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main ()
{
        string s="ConCho chay qua rao";
        cout << s.substr(2,4) << endl;
        // cout << new string(str.begin()+2, str.begin()+2+4);
        getchar();
        return 0;
```

```
}
```

- Phương thức insert( ) chèn thêm ký tự hay chuỗi vào một vị trí nào đó của chuỗi str cho trước. Có nhiều cách dùng phương thức này:

  str.insert(int pos, char* s; chèn s (mảng ký tự kết thúc '\0') vào vị trí pos của str;

  str.insert(int pos, string s); chèn chuỗi s (kiểu string) vào vị trí pos của chuỗi str;

  str.insert(int pos, int n, int ch); chèn n lần ký tự ch vào vị trí pos của chuỗi str;

```
// inserting into a string
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main ()
{
        string str="day la .. xau thu";
        string istr = "them";
        str.insert(8, istr);
        cout << str << endl;
        getchar();
        return 0;
}
```

- Phương thức str.erase(int pos, int n) xóa n ký tự của chuỗi str kể từ vị trí pos; nếu không quy định giá trị n thì tất cả các ký tự của str từ vị trí pos trở đi sẽ bị xóa

```
// erase from a string
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main ()
```

```
{
    string str="day cung la xau thu";
    str.erase(0, 3); // " cung la xau thu"
    cout << str << endl;
    str.erase(6, 2);
    cout << str << endl; // " cung xau thu"
    getchar();
    return 0;
}
```

3.4.2.3. So sánh (compare)

Bạn có thể đơn giản là sử dụng những toán tử quan hệ (==, !=, <, <=, >=) được định nghĩa sẵn. Tuy nhiên, nếu muốn so sánh một phần của một chuỗi thì sẽ cần sử dụng phương thức compare():

```
int compare ( const string& str ) const;
int compare ( const char* s ) const;
int compare ( size_t pos1, size_t n1, const string& str ) const;
int compare ( size_t pos1, size_t n1, const char* s) const;
int compare ( size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2 ) const;
int compare ( size_t pos1, size_t n1, const char* s, size_t n2) const;
```

Hàm trả về 0 khi hai chuỗi bằng nhau và lớn hơn hoặc nhỏ hơn 0 cho trường hợp khác

Ví dụ:

```
// comparing apples with apples
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string str1 ("green apple");
    string str2 ("red apple");

        if (str1.compare(str2) != 0)

                cout << str1 << " is not " << str2 << "\n";
        if (str1.compare(6,5,"apple") == 0)

                cout << "still, " << str1 << " is an apple\n";
        if (str2.compare(str2.size()-5,5,"apple") == 0)

                cout << "and " << str2 << " is also an apple\n";
        if (str1.compare(6,5,str2,4,5) == 0)

                cout << "therefore, both are apples\n";
```

```
        return 0;
    }
```

3.4.2.4. Các phương thức tìm kiếm và thay thế

- Phương thức find( ) tìm kiếm xem một ký tự hay một chuỗi nào đó có xuất hiện trong một chuỗi str cho trước hay không. Có nhiều cách dùng phương thức này:

str.find(int ch, int pos = 0);	tìm ký tự ch kể từ vị trí pos đến cuối chuỗi str
str.find(char *s, int pos = 0);	tìm s (mảng ký tự kết thúc '\0') kể từ vị trí pos đến cuối
str.find(string& s, int pos = 0);	tìm chuỗi s kể từ vị trí pos đến cuối chuỗi.

Nếu không quy định giá trị pos thì hiểu mặc nhiên là 0; nếu tìm có thì phương thức trả về vị trí xuất hiện đầu tiên, ngược lại trả về giá trị -1.

```
//find substring
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main ()
{
        string str="ConCho chay qua rao";
        cout << str.find("chay") << endl; // 7
        cout << (int)str.find("Chay") << endl; // -1
        getchar();
        return 0;
}
```

- Hàm tìm kiếm ngược (rfind)

```
//find from back
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
```

```
int main ()
{
    string str="ConCho chay qua chay qua rao";
    cout << str.find("chay") << endl; // 7
    cout << (int)str.rfind("chay") << endl; // 16
    getchar();
    return 0;
}
```

- Phương thức replace( ) thay thế một đoạn con trong chuỗi str cho trước (đoạn con kể từ một vị trí pos và đếm tới nchar ký tự ký tự về phía cuối chuỗi) bởi một chuỗi s nào đó, hoặc bởi n ký tự ch nào đó. Có nhiều cách dùng, thứ tự tham số như sau:

```
str.replace(int pos, int nchar, char *s);
str.replace(int pos, int nchar, string s);
str.replace(int pos, int nchar, int n, int ch);
```

```
// replace from a string
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main ()
{
    string str="con cho la con cho con. Con meo ko phai la con cho";
    str.replace(4, 3, "CHO"); // "con CHO la con cho con. Con meo ko phai la con cho";
    cout << str << endl;
    getchar();
    return 0;
}
```

3.4.2.5. Chuyển vè dạng C-string

Hàm thành viên c_str() sẽ giúp bạn chuyển từ string thành dạng C-string:

```
const charT* c_str ( ) const;
```

Hàm này cũng tự động sinh ra ký tự null chèn vào cuối xâu.
Từ prototype ta cũng thấy được hàm trả về một hằng chuỗi, điều này đồng nghĩa với việc ta không thể thay đổi chuỗi trả về.

Gọi phương thức c_str();

```
string s = "some_string";
cout << s.c_str() << endl;
cout << strlen(s.c_str()) << endl;
```

Sau đây là ví dụ dùng c_str() và các hàm trong <string.h>

```
// strings vs c-strings
#include <iostream>
#include <string.h>
#include <string>
   using std::string;
int main ()
{
   char * cstr, *p;
    string str ("Xin chao tat ca cac ban");

   cstr = new char [str.size()+1];
   strcpy (cstr, str.c_str());
   // cstr là 1 bản sao c-string của str

   p=strtok (cstr," ");
   while (p!=NULL)
   {
         cout << p << endl;
         p=strtok(NULL," ");
   }

   delete[] cstr;
   return 0;
}
```

Output:

```
Xin
chao
tat
ca
cac
ban
```

3.4.2.6. Một số phương thức khác

Còn nhiều phương thức tiện ích khác như: **append(), rfind(), find_first_not_of(), find_last_not_of(), swap()**. Cách dùng các hàm này đều được trình bày trong hệ thống hướng dẫn (help) của các môi trường có hỗ trợ STL (trong VC++ là MSDN). Ngoài ra các phương thức như **find_first_of()** tương tự như **find(), find_last_of()** tương tự như **rfind().**

3.5. BITSET

bitset có cấu trúc giống như một mảng, nhưng mỗi phần tử chỉ chiếm một bit (nên nhớ kiểu dữ liệu char mỗi phần tử chiếm 8 bit)

Ví dụ sau ta khởi tạo một bitset 7 phần tử với 5 phần tử đầu là 1,1,0,1,0

```
#include<bitset>
bitset<7> b(string("01011"));
for(int i=0;i<7;i++) cout<<b[i]<<endl;
```

3.6. VALARRAY

valarray giống như là một mảng lưu trữ các phần tử. Nó đáng chú ý vì nó có thể làm việc được với các hàm toán học thường dùng trong thư viện <math>, <functional>, các toán tử cơ bản và cũng như nhiều hàm tự định nghĩa.Valaray cho phép tính toán hang loạt trên 1 bảng số.

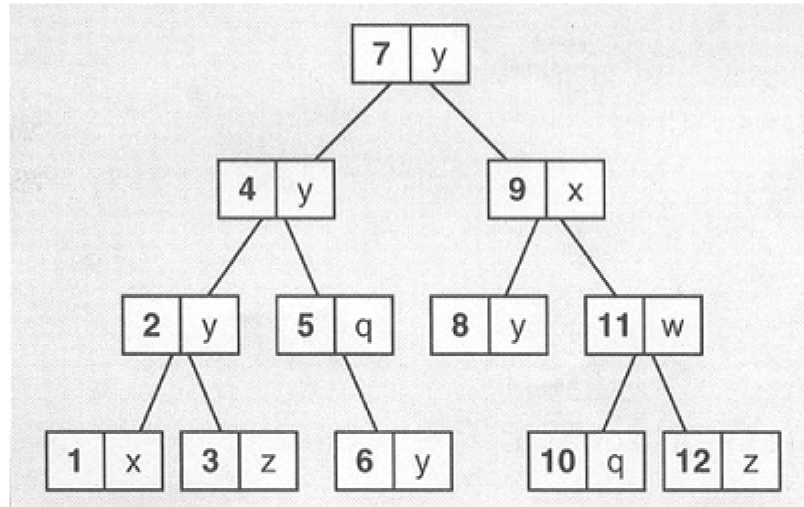
```
#include<valarray>
#include<cmath>
int i=0;
int a1[]= {3,2,6,4,5};
valarray<int> v1(a1,5);
v1 <<= 3;//phép toán << (dịch trái bit)
for(i=0;i<4;i++) cout<<v1[i]<<endl;
double a2[] = {2.4,6.8,0.2};
valarray<double> v2(a2,3);
v2 = sin(v2);//hàm sin
for(i=0;i<3;i++) cout<<v2[i]<<endl;
```

4. Associative Container

Associative bao gồm map (ánh xạ) multimap (đa ánh xạ) set (tập hợp) multiset (đa tập hợp)
Sự khác nhau giữa các associative container và sequential container :
-các sequential container lưu trữ các phần tử (gọi là các value) và các value này được truy xuất tuần tự theo vị trí của chúng trong bộ lưu trữ
-các associative container lưu trữ các phần tử (gọi là các value) và các khóa (gọi là các key) liên kết với các value và các value này được truy xuất theo các key mà chúng có liên kết

4.1. Map & multimap

Kiểu map cho phép bạn lấy tương ứng giữa một giá trị với một giá trị khác, hai giá trị này tạo thành một cặp giá trị. Trong đó giá trị đầu của cặp là khóa (key), key là duy nhất (không có 2 key cùng xuất hiện trong 1 map). Do đó, từ key bạn có thể tìm được giá trị tương ứng với key trong cặp.



Nếu thực hiện tìm kiếm bình thường trên một mảng N phần tử, bạn phải mất trung bình N/2 phép tìm kiếm. Dữ liệu của các key được tổ chức dưới dạng cây heap (lá trái nhỏ hơn gốc, lá phải lớn hơn gốc) nên việc tìm kiếm các cặp theo khóa rất nhanh, thời gian trung bình là log2N (vì độ sâu của cây là log2N).

```
#include<map>
map<char*,int> mapInt; //ánh xạ từ một char* đến một int
mapInt["one"] = 1;
cout<<mapInt["one"];
```

Như bạn thấy, trong map ta truy xuất 1 phần tử với operator[] thông qua khóa (key) thay vì chỉ số (index)

Ánh xạ trong map đi từ một key đến một value. Ví dụ sau key là lớp string, value là lớp Person

```
class Person
{
   public:string name;
   Person(string name):name(name){}
      friend ostream& operator<<(ostream& os,const Person& p)
      {
            os<<p.name;
      return os;
      }
};
typedef map<string,Person> MP;
void display(const MP& mp)
{
   for(const_iterator i=mp.begin();i!=mp.end();++i)

         cout<<(*i).first<<" "<<(*i).second<<endl;
}
int main()
{
   MP mapPerson;Person p("Viet");
   mapPerson.insert( value_type("one",p) );
   display(mapPerson);
   return 0;
}
```

Giải thích:
value_type thực chất là lớp pair của thư viện utility: pair<string,Person>, hàm dựng của nó dùng để khởi tạo một cặp (key,value) cho một ánh xạ.

Còn một cách khác là dùng hàm make_pair

```
pair<string,Person> pr = make_pair(string("two"),Person("Nam"));
mapPerson.insert(pr);
```

Comparator
Một functor dùng làm tiêu chí so sánh, sắp xếp, etc các phần tử trong một map gọi là một comparator. Comparator sẽ định nghĩa thế nào là lớn hơn, nhỏ hơn, bằng nhau theo cách của chúng ta, hoặc theo một số cách đã định nghĩa sẵn: less, greater … Khi đó map thay vì có 2 argument như map<key K,value V> thì có 3 argument là map<key K,value V,comparator C>
Dùng comparator để so sánh

```
class comparePerson
{
   public:
      bool operator()(Person p1,Person p2)
   {
            return p1.name.compare(p2.name);
      }
};
typedef map<Person,int, comparePerson> MAP;
MAP pMap;
Person p=new Person(...);
MAP::iterator i=pMap.find(d);
if(i==pMap.end()) pMap.insert(MAP::value_type(d,1));
```

Dùng comparator để sắp xếp

```
class comparePerson
{
   public:
      bool operator()(const Person& p1,const Person& p2)
      {
            return p1.name.compare(p2.name);
      }
};
typedef map<string,Person,comparePerson> MP;
MP mapPerson;Person p("Viet");
mapPerson.insert(pair<string,Person>("one",Person("Nam")));
mapPerson.insert(pair<string,Person>("two",Person("Viet")));
mapPerson.insert(pair<string,Person>("three",Person("An")));
display(mapPerson);
```

Bạn lưu ý là tất cả các asociative container đều có xây dựng sẵn comparator mặc định là less<key> (trong thư viện functional) Nghĩa là khi bạn khai báo

```
map<char*,int> mapInt;
```

thực ra là

map<char*,int,less<char*> > mapInt;

Ví dụ:

```
typedef map<char*,int> MI;
typedef map<char*,int>::iterator MII;
MI m;m["c"] = 1;m["b"] = 2;m["a"] = 3;
for(MII i=m.begin();i!=m.end();++i)
    cout<<(*i).first<<" "<<(*i).second<<endl;
```

Chạy thử bạn sẽ thấy các value trong map đã được sắp xếp lại vị trí theo các key của chúng
comparator dùng với các sequential container

```
class People
{
        public:
    int age;
        People( int age ) { (*this).age=age; }
};
class AgeSort
{
        public:
    bool operator()(const People*& a,const People*& b)
        {
                return (*a).age>(*b).age;
        }
};
typedef list<const People*> LP;
int main()
{
    const People* p1 = new People(5);const People* p2 = new People(7);
        LP p;
    p.push_back(p1);
    p.push_back(p2);
        p.sort(AgeSort());//using sort with comparator
        for(LP::const_iterator i=p.begin();i!=p.end();++i)
        cout<<(**i).age;
    return 0;
}
```

Nếu bạn biết tận dụng, map có thể ứng dụng để giải rất nhiều dạng bài toán khác nhau. Trong phần này, chúng tôi xin giới thiệu 2 ứng dụng của map:

- Sử dụng làm từ điển: Ví dụ: tra cứu thông tin của sinh viên theo mã số

- Đếm số lượng của một thành phần (cho một loạt các phần tử, tìm xem có bao nhiêu phần tử và mỗi phần tử xuất hiện bao nhiêu lần)

Ví dụ: từ điển

```
#include <iostream>
#include <map>
using namespace std;
class sv
{
    public:
    int maso;
    string ten;
    string lop;
    void print()
    {
        cout << "*****************************************" << endl
            << "Ma so: " << maso << endl
            << "Ho ten: " << ten << endl
            << "Lop: " << lop << endl;
    }
};
int main ()
{
    sv a[] =   {
        {1342,   "Mai",  "48th"},
        {43212,  "Lan",  "47th"},
        {33133,  "Cuc",  "47th"},
        {43321,  "Truc", "45th"},
        {1234,   "Dao",  "42th"}
    };
    int n = sizeof(a)/sizeof(a[0]);
    map<int, sv> m;
    for (int i=0; i<n; i++)
        m[a[i].maso] = a[i];
    m[43212].print();
    m[1234].print();
    return 0;
}
```

Ví dụ: đếm

```
#include <iostream>
#include <stdlib.h>
#include <map>
using namespace std;
int main()
{
    string a[] = {"chuoi", "na", "oi", "tao", "chuoi", "oi", "na", "tao", "oi"};
    int n = sizeof(a)/sizeof(a[0]);
    map<string, int> m;
    for (int i=0; i<n; i++)
        m[a[i]] ++;

    map<string, int>::iterator j;
    for (j=m.begin(); j!=m.end(); j++)
```

```
        cout << "Xau ky tu '" << j->first << "' xuat hien "
          << j->second << " lan " << endl;
    return 0;
    }
```

Multimap

Với map thì mỗi key chỉ ánh xạ đến một và chỉ một value. Với multimap thì mỗi key có thể ánh xạ đến nhiều hơn một value, nói cách khác là nhiều value trong multimap có chung một key

```
#include<map>
typedef multimap<string,Person> MP;
MP multimapPerson;
multimapPerson.insert(MPVT("one",Person("Nam")));
multimapPerson.insert(MPVT("one",Person("Viet")));
display(multimapPerson);
typedef multimap<Person,int,comparePerson> MAP;
```

Cũng chính vì lí do nhiều value trong multimap có thể có chung một key nên multi không có operator[] như map, tức là bạn không thể gọi multimapPerson[“one”]

4.2. Kiểu hash_map (ánh xạ dùng mảng băm)

Kiểu map cho phép ánh xạ giữa tập khóa và tập giá trị với thời gian O(logN). Map sử dụng cấu trúc dữ liệu kiểu cây nên độ phức tạp là log2N. Tuy nhiên, bạn có thể sử dụng cấu trúc dữ liệu mạnh hơn là mảng băm (hash_map). Hash_map có thể tìm kiếm theo khóa với độ phức tạp O(1).

4.3. Set & mutiset

set cũng giống map ngoại trừ một điều, key cũng chính là value

```
#include<set>
set<int> s;
for(int j=0;j<6;j++) s.insert(rand());
for(set<int>::iterator i=s.begin();i!=s.end();++i)
    cout<<(*i)<<endl;
```

set dùng với comparator (greater đóng vai trò comparator):

```
set<int,greater<int> > s;
for(int j=0;j<6;j++) s.insert(rand());
for(set<int,greater<int> >::iterator i=s.begin();i!=s.end();++i)
  cout<<(*i)<<endl;
```

set không có operator[]

Multiset

multiset cũng giống set ngoại trừ một điều, mỗi key có thể ánh xạ đến nhiều hơn một value, nói cách khác là nhiều value trong multiset có chung một key

Bạn có thể thắc mắc điều này chẳng có ý nghĩa gì, vì trong set thì key cũng chính là value, vâng, chính vì thế nên một multiset sẽ có thể chứa những phần tử giống nhau:

```
#include<set>
set<int> s;
s.insert(1);
```

```
s.insert(1);
for(set<int>::iterator i=s.begin();i!=s.end();++i)
        cout<<(*i)<<endl;
multiset<int> ms;
ms.insert(3);
ms.insert(3);
for(multiset<int>::iterator mi=ms.begin();mi!=ms.end();++mi)
        cout<<(*mi)<<endl;
```

4.4. Kiểu hash_set (tập hợp)

Pascal là một trong ít ngôn ngữ có kiểu dữ liệu nguyên thủy là tập hợp. C và C++ không có kiểu dữ liệu này. Tuy nhiên, bạn có thể sử dụng kiểu dữ liệu tập hợp trong STL. Tập hợp trong STL còn mạnh hơn tập hợp trong Pascal rất nhiều vì nó hỗ trợ tất cả các kiểu dữ liệu (tập hợp trong Pascal chỉ hỗ trợ dạng số) với số lượng phần tử không hạn chế (tập hợp của Pascal chỉ được tối đa 256 phần tử).

IV.FUNTION OBJECT (FUNCTOR)

1. KHÁI NIỆM

Một function object (đối tượng hàm) là một object (đối tượng) được sử dụng như một function (hàm). Gọi function object nghĩa là chúng ta đang gọi đến operator() của nó. Viết một function object nghĩa là viết operator() cho một lớp. Các function object là các object, bởi vậy chúng có trạng thái, còn các hàm bình thường thì không, do đó, chúng có thể ứng xử khác nhau tùy vào trạng thái – và điều đó tạo nên sự linh hoạt

Vậy function object là một instance của một lớp mà lớp đó phải có ít nhất một hàm thỏa:

- quyền truy xuất phải là public
- phải là một hàm thành viên, không phải là một hàm friend
- không phải là một hàm static
- có khai báo operator()

2. PHÂN LOẠI

- Generator: Một loại functor hoặc function không có đối số và trả về value_type ví dụ hàm rand() trong <stdlib> và một số thuật toán chẳng hạn như generate_n() - sinh một chuỗi.
- Unary: Một loại functor hoặc function dùng một đối số duy nhất của value_type và trả về một giá trị mà có thể không phải value_type (void chẳng hạn).
- Binary: Một loại functor hoặc function nhận hai đối số của hai kiểu bất kỳ và trả về giá trị nào đó.
- Unary Predicate: Một unary operation trả lại giá trị bool.
- Binary Predicate: Một binary operation trả lại giá trị bool.

Ngoài ra, ta cón phân loại dựa trên tính chất object của functor:

- LessThanComparable: Một functor có định nghĩa ít nhất một toán tử <.
- Assignable: Một functor có định nghiã toán tử gán (=)
- EqualityComparable: Một functor có định nghĩa toán tử so sánh tương đương ==

3. SỬ DỤNG FUNCTION OBJECT

Ví dụ ta viết một hàm bình thường như sau

```
void iprintf(int i) const
{
   cout<<i<<endl;
}
```

Bây giờ ta sẽ viết một lớp như sau

```
class iprintf
{
public:
   void operator()(int i) const
   {
      cout<<i<<endl;
   }
};
```

Instance của lớp này là một object được gọi là function object, là một object được sử dụng như một function. Sử dụng như thế nào ?

```
iprintf x;
x(5);
```

hoặc

```
iprintf()(5);
```

Khi ta gọi iprintf()(5) nghĩa là chúng ta đang gọi đến operator() của lớp iprintf

Cài đặt cụ thể cho operator() tùy thuộc vào ngữ cảnh sử dụng của function object. Sự phức tạp hóa này mang lại ứng dụng :

1.Làm tiêu chí sắp xếp cho các container: nếu các phần tử của set là các kiểu cơ bản như int hay string, chúng ta có thể sử dụng các tiêu chí sắp xếp sẵn có như greater hay less.

Ví dụ dòng khai báo : `std::set< std::string, greater< std::string > > strSet;`

Tuy nhiên, nếu các phần tử cần đưa vào set có kiểu do người dùng định nghĩa, ví dụ là các đối tượng của một lớp, mà thậm chí lớp đó không có operator < > thì làm sao để xác định thứ tự của chúng trong set? Cách giải quyết là chúng ta tự định nghĩa một tiêu chí sắp xếp mới, đây chính là lúc cần đến function object. (xem trong phần associative container)

2.Làm tham số cho các STL algorithm – sẽ được nói đến ngay sau đây.

Việc một function object được sử dụng ở đâu sẽ quyết định cách viết operator() của lớp đó.

Ví dụ dưới đây là một lớp có nhiều hơn một operator()

```
class iprintf
{
int i;
   public:iprintf(int i):i(i){}
public:
   void operator()() const
```

```
    {
        cout<<i<<endl;
    }
    void operator()(int i) const
    {
        cout<<"Integer:"<<i<<endl;
    }
    void operator()(float f) const
    {
        cout<<"Float:"<<f<<endl;
    }
};
int main(int argc,char** argv)
{
    iprintf x(20);
    x();
    x(5);   //giả sử không có operator()(int i), câu này sẽ gọi operator()(float f)
    x(2.3);   //giả sử không có operator()(float f), câu này sẽ gọi operator()(int i) với i = 2
    x("something");   //lỗi
    return 0;
}
```

Tương tự thay vì iprintf(5); x(7); chúng ta cũng có thể gọi iprintf(5)(7);
Có một điều chú ý ở ví dụ trên là nếu cùng tồn tại operator()(int i) và operator()(float f) thì câu lệnh x(2.3); sẽ báo lỗi ambiguous (nhập nhằng) giữa hai hàm. Có một cách đơn giản là viết lại thành x((float)2.3); Về chuyện ambiguous còn nói thêm sau.

- **Predicate**

  Predicate có một định nghĩa phức tạp hơn. Một predicate được đề cập đến ở đây là một function hoặc một functor có điều kiện giá trị trả về đúng hoặc sai hoặc một giá trị có thể chuyển kiểu thành đúng hoặc sai. Trong C/C++, đúng có nghĩa là khác 0 và sai có nghĩa là bằng 0
  Ví dụ hàm sau đây là một predicate

```
double truefalse(double n)
{
    return n;
}
```

Một số hàm thường dùng trong algorithm

- Hàm find

```
vector<int> v;
v.push_back(4);v.push_back(3);v.push_back(2);
vector<int>::iterator i = find (v.begin(),v.end(),3);
if(i!=v.end()) cout<<*i;
```

  Hàm find tìm từ phần tử v.begin() đến phần tử v.end() và trả về iterator trỏ đến phần tử có giá trị là 3, nếu không tìm thấy sẽ trả về v.end()

- Hàm find_if

```
int IsOdd(int n)
{
    return n%2;
}
int main()
{
    list<int> l;
    l.push_back(4);l.push_back(5);l.push_back(2);
    list<int>::iterator i=find_if(l.begin(),l.end(),IsOdd);
```

```
    if(i!=l.end()) cout<<*i;
}
```

Hàm find_if tìm từ phần tử v.begin() đến phần tử v.end() và trả về iterator trỏ đến phần tử có giá trị thỏa predicate, nếu không tìm thấy sẽ trả về v.end()

Lưu ý, lúc này IsOdd đóng vai trò là một predicate, xác định xem phần tử của list có là số lẻ hay không (tức là khi đưa vào làm tham số của hàm IsOdd có trả về một số khác 0 hay không)
Chúng ta viết lại predicate này bằng cách dùng functor

```
class IsOdd
{
public:
    bool operator()(int n) const
    {
        return n%2;
    }
};
int main()
{
    list<int> l;
    l.push_back(4);l.push_back(5);l.push_back(2);
    list<int>::iterator i=find_if(l.begin(),l.end(),IsOdd());
    if(i!=l.end()) cout<<*i;
}
```

- Hàm equal
  Ở trên chúng ta mới xét các ví dụ với predicate có một đối số, ta xét một hàm khác của algorithm dùng predicate nhiều hơn một đối số, hàm equal

```
class compare
{
public:
    bool operator()(int i,int j) const
    {
        return i==j;
    }
};
int main()
{
    compare c;
    int a[] = {1, 2, 3, 4, 5};
    list<int> l(a,a+3);    //list ít phần tử hơn mảng
    cout<<equal(l.begin(),l.end(),a,c)<<endl;
    a[2] = 6;
    cout<<equal(l.begin(),l.end(),a,c)<<endl;
    return 0;
}
```

Hàm equal so sánh từng phần tử của list từ phần tử l.begin() đến phần tử l.end() với từng phần tử tương ứng của mảng a sao cho mỗi cặp phần tử đều thỏa predicate là c, trả về là true nếu từng cặp phần tử so sánh với nhau đều cho giá trị true (không cần quan tâm đến số lượng phần tử có tương ứng không)
Nhưng chỉ cần một cặp trả về false thì hàm sẽ trả về false

4. THƯ VIỆN FUNCTIONAL

```
#include <functional>
```

a)Hạng của một predicate

Có nhiều sự mập mờ do từ đồng nghĩa giữa các khái niệm toán học trong cả hai ngôn ngữ tiếng Việt và tiếng Anh, do đó định nghĩa sau chỉ ở mức cố gắng chính xác nhất có thể được:
Số toán tử (operand) của một phép toán (operator), tương ứng là số đối số (argument) của một hàm (function), được gọi là hạng (arity) của phép toán hay hàm đó

Tương tự, số toán tử (operand) của một biểu thức (expression), tương ứng là số đối số (argument) của một đối tượng hàm (functor), được gọi là hạng (arity) của biểu thức hay đối tượng hàm đó

Ví dụ:

-Unary (đơn nguyên, đơn phân, một toán hạng, một ngôi):

n! (giai thừa của n) là một unary operator
n! là một unary expression, chỉ bao gồm một unary operator
int giaithua(int n) là một unary function
một object của class giaithua{int operator()(int n)…} là một unary functor

-Binary (nhị nguyên, nhị phân, hai toán hạng, hai ngôi):

a + b là một binary expression, chỉ bao gồm một binary operator
int addition(int a,int b) là một binary function
một object của class addition{int operator()(int a,int b)…} là một binary functor

-Ternary (tam nguyên, tam phân, ba toán hạng, ba ngôi):

b * b – 4 * a * c là một ternary expression, bao gồm một unary operator và ba binary operator
double delta(double a, double b,double c) là một ternary function
một object của class delta{ double operator()(double a, double b,double c)…} là một ternary functor

Ngoài ra còn có nhiều từ gốc Latin khác như quaternary (bốn toán hạng) quinary (năm toán hạng) … n-ary gọi chung là nhiều toán hạng.

Hạng của predicate tức là hạng của function hay functor mà đóng vai trò predicate. Như ví dụ ở trên, addition là một binary predicate, delta là một ternary predicate

- Cấu trúc unary_function trong thư viện functional
  Trong thư viện functional đã định nghĩa sẵn cấu trúc unary_function:

```
template<class Arg,class Result>
struct unary_function
{
   typedef Arg argument_type;
   typedef Result result_type;
};
```

  unary_function là cấu trúc định nghĩa sẵn cho tất cả unary function và unary functor với Arg là kiểu dữ liệu của đối số và Result là kiểu trả về của hàm có operator()

  Chúng ta viết lại lớp IsOdd, định nghĩa nó là một unary_function

```
class IsOdd:public unary_function<int,bool>
{
public:
   bool operator()(int n) const
   {
      return n%2;
   }
};
```

- Cấu trúc binary_function trong thư viện functional
  Tương tự, trong thư viện functional đã định nghĩa sẵn cấu trúc binary_function

```
template<class Arg1,class Arg2,class Result>
struct binary_function
```

```
{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};
```

binary_function là cấu trúc định nghĩa sẵn cho tất cả binary function và binary functor với Arg1 là kiểu dữ liệu của đối số thứ nhất và Arg2 là kiểu dữ liệu của đối số thứ hai và Result là kiểu trả về của hàm có operator()

Chúng ta viết lại lớp compare, định nghĩa nó là một binary_function

```
class compare:public binary_function<int,int,bool>
{
public:
   bool operator()(int i,int j) const
   {
      return i==j;
   }
};
```

Tương tự chúng ta có thể tự viết các cấu trúc ternary_function, quaternary_function, vân vân nếu muốn. Ví dụ dưới đây là một cấu trúc ternary_function tự viết và một lớp được định nghĩa là một ternary_function

```
template<class Arg1,class Arg2,class Arg3,class Result>
struct ternary_function
{
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Arg3 third_argument_type;
   typedef Result result_type;
};
class multiply:public ternary_function<int,float,long,double>
{
public:
   double operator()(int i,float f,long l) const
   {
      return i*f*l;
   }
};
```

- Ràng buộc (bind) toán hạng cho predicate

Có nhiều hàm chỉ chấp nhận một đối số, nhưng chúng ta lại cần chuyền vào cho nó các predicate là binary predicate như binary function hay binary functor. Trong trường hợp đó chúng ta cần ràng buộc toán hạng cho binary predicate đó để nó trở thành một unary predicate.

Ví dụ chúng ta cần dùng hàm find_if để tìm các phần tử trong một vector thỏa một binary predicate, nhưng find_if lại chỉ chấp nhận unary predicate, khi đó chúng ta cần ràng buộc toán hạng cho binary predicate đó để nó trở thành một unary predicate

binary predicate muốn được ràng buộc toán hạng phải được định nghĩa là một binary_function

+Hàm bind1st

Hàm bind1st ràng buộc toán hạng thứ nhất của một binary predicate với một giá trị cho trước để nó trở thành một unary predicate với đối số còn lại của binary predicate ban đầu trở thành đối số của unary predicate kết quả

```
class compare:public binary_function<int,int,bool>
```

```
{
public:
   bool operator()(int i,int j) const
   {
      return i+1==j;
   }
};
int main()
{
   vector<int> v;
   v.push_back(4);v.push_back(0);v.push_back(1);
   vector<int>::iterator i=find_if(v.begin(),v.end(),bind1st(compare(),0));
   if(i!=v.end()) cout<<i-v.begin();
   return 0;
}
```

Trong ví dụ trên, đối số thứ nhất của compare() đã được ràng buộc bằng 0, compare() trở thành một predicate chỉ có một đối số là đối số còn lại của compare() ban đầu, và find_if chỉ việc truyền tham số là iterator trỏ đến các phần tử của v vào đối số này, quá trình chạy vòng lặp diễn ra giống như sau

```
compare()(0,4) //phép so sánh 0 + 1 == 4 trả về false
compare()(0,0) //phép so sánh 0 + 1 == 0 trả về false
compare()(0,1) //phép so sánh 0 + 1 == 1 trả về true
```

+Hàm bind2nd

Hàm bind2nd ràng buộc toán hạng thứ hai của một binary predicate với một giá trị cho trước để nó trở thành một unary predicate với đối số còn lại của binary predicate ban đầu trở thành đối số của unary predicate kết quả

```
class compare:public binary_function<int,int,bool>
{
public:
   bool operator()(int i,int j) const
   {
      return i+1==j;
   }
};
int main()
{
   vector<int> v;
   v.push_back(4);v.push_back(0);v.push_back(1);
   vector<int>::iterator i=find_if(v.begin(),v.end(),bind2nd(compare(),1));
   if(i!=v.end()) cout<<i-v.begin();
   return 0;
}
```

Trong ví dụ trên, đối số thứ hai của compare() đã được ràng buộc bằng 1, compare() trở thành một predicate chỉ có một đối số là đối số còn lại của compare() ban đầu, và find_if chỉ việc truyền tham số là iterator trỏ đến các phần tử của v vào đối số này, quá trình chạy vòng lặp diễn ra giống như sau

```
compare()(4,1) //phép so sánh 4 + 1 == 1 trả về false
compare()(0,1) //phép so sánh 0 + 1 == 1 trả về true
compare()(1,1) //phép so sánh 1 + 1 == 1 trả về false (thực ra không có phép so sánh này, hàm
đã trả về iterator rồi)
```

b)Các hàm toán học cơ bản của thư viện functional

Bao gồm cộng (plus) trừ (minus) nhân (multiplies) chia (divides) chia lấy dư (modulus) đổi dấu (negate)
Các hàm này rất đơn giản, ví dụ:

negate:

```
int a[]={1,-2,3};
transform(a,a+3,a,negate<int>());
for_each(a,a+3,Output<int>());
```

plus:

```
int a[]={1,2,3,4,5};
int b[]={6,7};
int c[5];
transform(a,a+5,b,c,plus<int>());
```

Ở vd trên có một điều đáng chú ý, bạn tự tìm xem

modulus:

```
int a[]={1,2,3,4,5};
int b[]={2,2,2,2,2};
int c[5];
transform(a,a+5,b,c,modulus<int>());
```

Cái ví dụ hàm modulus này hơi … kì kì. Modulus là một binary function, giả sử bây giờ chúng ta muốn các phần tử của a luôn modulus cho 2 thì làm thế nào ? Phải ràng buộc toán hạng cho modulus để nó trở thành một unary function thôi:

```
int a[]={1,2,3,4,5};
int b[5];
transform(a,a+5,b,bind2nd(modulus<int>(),2));
```

Các hàm so sánh bao gồm equal_to (==) not_equal_to (!=) greater (>) less (<) greater_equal (>=) less_equal(<=) logical_and (&&) logical_or (||) logical_not (!)
Các hàm này cách dùng y như nhau, lấy một ví dụ hàm greater:

```
int a[]={3,2,5,1,4};
sort(a,a+5,greater<int>());
for_each(a,a+5,Output<int>());
```

Giả sử ta muốn dùng hàm count_if với hàm greater, trả về số phần tử nhỏ hơn 3 chẳng hạn. Ta làm thế nào ? greater là một binary function, lại phải ràng buộc toán hạng cho nó với đối số thứ nhất là 3 rồi

```
int a[]={3,2,5,1,4};
cout<<(int)count_if(a,a+5,bind1st(greater<int>(),3));
for_each(a,a+5,Output<int>());
```

V.THƯ VIỆN ALGORITHM

Như đã giới thiệu trong các phần trước, STL cung cấp các thuật toán cơ bản nhằm mục đích giúp bạn không phải code lại những giải thuật quá cơ bản như (sắp xếp, thay thế, tìm kiếm...). Các công cụ này không những giúp bạn rút ngắn thời gian lập trình mà còn cả thời gian gỡ rối khi thuật toán cơ bản được cài đặt không chính xác.

Ngoài ra, với STL Algorithm, bạn có nhiều lựa chọn cho những thuật toán cơ bản. Ví dụ, với thuật toán sắp xếp, bạn có thể lựa chọn giữa thuật toán sắp xếp nhanh (quicksort) cho kết quả rất nhanh với độ phức tạp NlogN trong đa số các trường hợp, nhưng lại có độ phức tạp N*N trong trường hợp xấu nhất và thuật toán sắp xếp vung đống (heapsort) chạy chậm hơn quicksort nhưng có độ phức tạp trong mọi trường hợp là NlogN.

Chú ý rằng các thuật toán của STL Algorithm có thể áp dụng cho mọi kiểu iterator, kể cả con trỏ thường (không phải là iterator của STL). Như vậy, các thuật toán sắp xếp, tìm kiếm, thay thế không những áp dụng được cho các kiểu vector, list... mà còn có thể áp dụng cho mảng thông thường.

Để khai báo sử dụng STL algorithm, các bạn phải include file header algorithm: #include <algorithm>

Do thư viện <algorithm> gồm rất nhiều hàm khác nhau, trong khuôn khổ tài liệu này không thể nêu hết được, xin giới thiệu sơ lược một số nhóm hàm sau:

- Nhóm các hàm không thay đổi container
- Nhóm các hàm thay đổi container
- Nhóm các hàm sắp xếp
- Nhóm các hàm trên danh sách được sắp xếp
- Nhóm các làm trên heap
- Nhóm các hàm tìm min/max

1. NHÓM CÁC HÀM KHÔNG THAY ĐỔI CONTAINER

- Các thuật toán tìm kiếm, bao gồm find(), find_if() tìm theo điều kiện, search() dùng để so khớp 1 chuỗi liên tiếp các phần tử cho trước, hàm search_n tìm kiếm với số lần lặp xác định, hàm find_end tìm kết quả cuối cùng, find_first_not_of(), find_last_not_of() …

```
//find
int A[] = {3,4,2,6,3,1,2,3,2,3,4,5,6,4,3,2,1};
int N = sizeof(A) / sizeof(*A);
int first = find(A, A+N, 1) - A;
cout << "So thu tu cua phan tu dau tien = 1: " << first << endl;
//find_if
vector<int>::iterator it;
it = find_if(v.begin(),v.end(), IsOdd );
first = it - v.begin();
cout << "Phan tu le dau tien la " << *it << " o vi tri thu " << first << endl;
//search
string A = "Xin chao tat ca moi nguoi !";
string B = "chao";
int vitri = search(A.begin(), A.end(), B.begin(), B.end()) - A.begin();
cout << "Vi tri dau tien xuat hien B trong A: " << vitri << endl;
```

- Các thuật toán đếm:

  - **Hàm count** dùng để đếm số lượng phần tử trong một chuỗi các phần tử cho trước

```
list<string> l;l.push_back("hello");l.push_back("world");
cout<<(int)count(l.begin(),l.end(),"hello")<<endl;
```

  - **Hàm count_if** dùng để đếm số lượng phần tử thỏa một điều kiện nào đó trong một chuỗi các phần tử cho trước, hàm cần một predicate một đối số

```
class IsOdd
{
public:
bool operator()(int n) const{return (n%2)==1;}
};
int main(int argc, char* argv[])
{
   list<int> l;for(int i=0;i<10;i++) l.push_back(i);
   cout<<(int)count_if(l.begin(),l.end(),IsOdd())<<endl;
}
```

2. NHÓM CÁC HÀM THAY ĐỔI CONTAINER

- Hàm **fill** để tô một vùng giá trị của 1 container (thường là 1 mảng, 1 vector)

```
vector<int> v(8);                // v: 0 0 0 0 0 0 0 0
fill(v.begin(),v.begin()+4,5);   // v: 5 5 5 5 0 0 0 0
fill(v.begin()+3,v.end()-2,8);   // v: 5 5 5 8 8 8 0 0
```

- **Hàm generate** sẽ "sinh" từng phần tử trong khoảng nào đấy của vector bằng cách gọi hàm được chỉ định (một hàm trả về cùng kiểu và không có đối số)

```
template <class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

Ví dụ với hàm rand():

```
vector<int> V;
srand( time(NULL) );
//...

generate( V.begin(), V.end(), rand );
```

- Hàm for_each dùng để duyệt từng phần tử trong một chuỗi các phần tử cho trước
Dùng for_each để in ra các phần tử, ví dụ:

```
void display(const string& s){cout<<s<<endl;}
list<string> l;l.push_back("hello");l.push_back("world");
for_each(l.begin(),l.end(),display);
```

Tương tự dùng với một functor:

```
template<typename T>class Output
{
public:
   void operator()(const T& t){cout<<t<<endl;}
};
int main(int argc, char* argv[])
{
   list<string> l;l.push_back("hello");l.push_back("world");
   for_each(l.begin(),l.end(),Output<string>());
}
```

- Hàm transform: phần tử được sửa đổi từng cái trong một phạm vi theo một chức năng mà bạn cung cấp.

Hàm này có hai phiên bản:

```
int increase(int i){return ++i;}
vector<int> v2;
v2.resize(v1.size());
transform(v1.begin(),v1.end(),v2.begin(),increase);
```

Phiên bản thứ nhất sẽ lấy tất cả phần tử từ v1.begin() đến v1.end(), transform chúng bằng hàm increase, sau đó chép giá trị đã transform vào bắt đầu từ v2.begin()

```
int addition(int i,int j){return i+j;}
vector<int> v3;
```

```
v3.resize(v1.size());
transform(v1.begin(),v1.end(),v2.begin(),v3.begin(),addition);
```

Phiên bản thứ hai sẽ lấy tất cả phần tử từ v1.begin() đến v1.end(), transform chúng bằng hàm addition với đối số thứ hai là tất cả phần tử từ v2.begin(), sau đó chép giá trị đã transform vào bắt đầu từ v3.begin()

- Thay thế các giá trị (replace)

```
int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
vector<int> a (myints, myints+8);       // 10 20 30 30 20 10 10 20
replace(a.begin(), a.end(), 20, 99);    // 10 99 30 30 99 10 10 99
```

- **Hàm replace_if** cho phép tìm giá trị theo điều kiện do một hàm trả về. Để sử dụng lệnh này bạn phải khai báo 1 hàm có giá trị trả về là bool nhận tham số là giá trị của 1 element. Khi hàm trả về true, giá trị tương ứng sẽ bị thay thế bởi giá trị mới. Hàm kiểm tra nên khai báo inline để tốc độ nhanh hơn.

```
vector<int> a;
// set some values:
for (int i=1; i<10; i++) a.push_back(i);     // 1 2 3 4 5 6 7 8 9
replace_if(a.begin(), a.end(), SoLe, 0);    // 0 2 0 4 0 6 0 8 0
```

- Đảo ngược containter (reverse)

```
vector<int> a;
// set some values:
for (int i=1; i<10; ++i) a.push_back(i);     // 1 2 3 4 5 6 7 8 9
reverse(a.begin(),a.end());                 // 9 8 7 6 5 4 3 2 1
```

- Copy iterator (tương tự memcpy() đối với pointer)

```
int a[] = {1, 2, 3, 4, 5, 6};
int n = sizeof(a)/sizeof(*a);
vector<int> v1(a, a+n);

vector<int> v2(n);
copy(v1.begin(), v1.end(), v2.begin());
//copy_n(v1.begin(), v1.size(), v2.begin());
copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
```

- Xóa với remove và remove_if

Ví dụ 1:

```
//remove

int A[] = {3,1,4,1,5,9};
int N = sizeof(A)/sizeof(*A);
vector<int> V(A, A+N);

vector<int>::iterator new_end =
   remove(V.begin(), V.end(), 1);
V.resize(new_end - V.begin());
copy(V.begin(), V.end(), ostream_iterator<int>(cout, " "));
// The output is "3 4 5 9".
```

Ví dụ 2:

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool IsOdd(int x)
{
        return x%2;
}

int main()
{
        int a[] = {3,1,4, 8, 5, 2, 9};
        int n = sizeof(a)/sizeof(*a);
        vector<int> vec(a, a+n);
        vector<int>::iterator new_end =

        remove_if( vec.begin(), vec.end(), IsOdd );
        vec.erase(new_end, vec.end());
        copy(vec.begin(), vec.end(), ostream_iterator<int>(cout, " "));
        // The output is "4 8 2".
        return 0;
}
```

- **Các hàm có hậu tố _copy như remove_copy, remove_if_copy, replace_copy, replace_if_copy, reverse_copy sử dụng tương tự nhưng tạo ra và thao tác trên bản sao container**

3. NHÓM CÁC HÀM SẮP XẾP

- Hàm sort (quicksort)

Hàm này có 2 phiên bản:

+Sắp xếp lại một chuỗi phần tử theo thứ tự tăng dần (ascending)

```
sort (v.begin(),v.end());
```

+Sắp xếp lại một chuỗi phần tử thỏa một binary predicate

```
sort(A, A+N, greater<int>() );
```

Hoặc:

```
template<typename T>class Bigger{
public:
   bool operator()(const T& t1,const T& t2){return t1>t2;}
};
template<typename T>class Output{
public:
   void operator()(const T& t){cout<<t<<endl;}
};
int main(int argc, char* argv[]){
   vector<int> v;for(int i=0;i<10;i++) v.push_back(i);
```

```
        sort(v.begin(),v.end(),Bigger<int>());
        for_each(v.begin(),v.end(),Output<int>());
        return 0;
    }
```

- **Hàm is_sorted** kiểm tra xem 1 chuỗi đã được sắp xếp hay chưa:

```
int A[] = {1, 4, 2, 8, 5, 7};
const int N = sizeof(A) / sizeof(int);

assert(!is_sorted(A, A + N));
sort(A, A + N);
assert(is_sorted(A, A + N));
```

4.NHÓM CÁC HÀM TRÊN DANH SÁCH ĐƯỢC SẮP XẾP

Một số thuật toán như tìm kiếm, thêm vào danh sách... hoạt động nhanh hơn (độ phức tạp là log2n thay vì n). Thư viện <algorithm> hỗ trợ một số hàm làm việc riêng với các danh sách đã sắp xếp theo thứ tự tăng dần.

- Tìm cận dưới và cận trên (lower_bound, upper_bound)

Hàm lower_bound(first, last, value) trả về iterator của element cuối cùng trong danh sách đã sắp xếp có giá trị không vượt quá [value].

Hàm upper_bound(first, last, value) trả về iterator của element đầu tiên có giá trị lớn hơn [value].

```
int myints[] = {10,20,30,30,20,10,10,20};
int size = sizeof(myints)/sizeof(myints[0]);
vector<int> v(myints,myints+size);            // 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 tro vao vi tri: " << int(low- v.begin()) << endl;//3
cout << "upper_bound tro vao vi tri: " << int(up - v.begin()) << endl;//6
```

- Tìm kiếm nhị phân (binary_search)

Hàm binary_search(first, last, value) trả về true nếu tìm thấy giá trị value trong danh sách đã sắp xếp từ first đến last.

```
int myints[] = {1,2,3,4,5,4,3,2,1};
int size = sizeof(myints)/sizeof(myints[0]);
vector<int> v(myints,myints+size);
sort (v.begin(), v.end(), greater<int>() );
cout << "Tim phan tu 6... ";
if (binary_search (v.begin(), v.end(), 6, LonHon))

        cout << "Tim thay!\n";
else cout << "Khong tim thay!\n";
```

- Trộn 2 danh sách đã được sắp xếp (merge)

```
int first[] = {5,10,15,20,25};
int second[] = {50,40,30,20,10};
vector<int> v(10);
vector<int>::iterator it;
sort (first,first+5);
sort (second,second+5);
merge (first,first+5,second,second+5,v.begin());
//5 10 10 15 20 20 25 30 40 50
```

- Các phép toán trên tập hợp:

  - Xác nhận tập con **includes**

```
int A1[] = { 1, 2, 3, 4, 5, 6, 7 };
int A2[] = { 1, 4, 7 };
int A3[] = { 2, 7, 9 };

const int N1 = sizeof(A1) / sizeof(int);
const int N2 = sizeof(A2) / sizeof(int);
const int N3 = sizeof(A3) / sizeof(int);

cout << "A2 contained in A1: "
   << (includes(A1, A1 + N1, A2, A2 + N2) ? "true" : "false") << endl;
cout << "A3 contained in A1: "
   << (includes(A1, A1 + N2, A3, A3 + N3) ? "true" : "false") << endl;
```

  - Hợp (set_union)

```
int first[] = {5,10,15,20,25};
int second[] = {50,40,30,20,10};
vector<int> v(10);
vector<int>::iterator it;
sort (first,first+5);
sort (second,second+5);
vector<int>::iterator end_it=set_union (first, first+5, second, second+5, v.begin());
copy(v.begin(), end_it, ostream_iterator<int>(cout, " "));
//5 10 15 20 25 30 40 50
```

  - Giao (set_intersection)

```
int first[] = {5,10,15,20,25};
int second[] = {25,40,15,20,10};
vector<int> v(10);
vector<int>::iterator it;
sort (first,first+5);
sort (second,second+5);
vector<int>::iterator end_it =set_intersection(first, first+5, second, second+5, v.begin());
//10 15 25
```

  - Phép loại (set_difference) lấy ra các phần tử sai khác

```
int first[] = {5,10,10,20,25};
int second[] = {25,40,15,20,5};
vector<int> v(10);
vector<int>::iterator it;
sort (first,first+5);
sort (second,second+5);
```

```
vector<int>::iterator end_it =set_difference(first, first+5, second, second+5, v.begin());
//10 10 15 40
```

- **Phép trừ tập hợp (set_symmetric_difference) gần giống set_difference nhưng khác ở chỗ nếu có 1 phần tử lặp n lần ở tập 1 và m lần ở tập 2 thì nó sẽ xuất hiện |m-n| lần ở output.**

5. CÁC HÀM TRÊN HEAP

Bao gồm tạo heap (make_heap), thêm phần tử vào heap (push_heap), xóa phần tử khỏi heap (pop_heap), sắp xếp heap (sort_heap)

```
// range heap example
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main ()
{
  int myints[] = {10,20,30,5,15};
  vector<int> v(myints,myints+5);
  vector<int>::iterator it;
  make_heap (v.begin(),v.end());
  cout << "initial max heap   : " << v.front() << endl;
  pop_heap (v.begin(),v.end()); v.pop_back();
  cout << "max heap after pop : " << v.front() << endl;
  v.push_back(99); push_heap (v.begin(),v.end());
  cout << "max heap after push: " << v.front() << endl;
  sort_heap (v.begin(),v.end());
  cout << "final sorted range :";
  for (unsigned i=0; i<v.size(); i++)
    cout << " " << v[i];
return 0;
}
```

6,CÁC HÀM TÌM MIN & MAX

- Tìm min & max trong 1 cặp:

```
const int x = min(3, 9),
  y = max(3, 9);
assert(x == 3);
assert(y == 9);
```

- Tìm min & max trong 1 tập

```
int A[] = {3,4,2,6,3,1,2,3,2,3,4,5,6,4,3,2,1};
int N = sizeof(A) / sizeof(*A);

cout << "So nho nhat trong mang: " << *min_element(A, A+N) << endl;
cout << "So lon nhat trong mang: " << *max_element(A, A+N) << endl;
```

VI. CÁC BỘ TƯƠNG THÍCH

1.CONTAINER ADAPTER (CÁC BỘ TƯƠNG THÍCH LƯU TRỮ)

Bao gồm stack, queue và priority_queue

Gọi là các bộ tương thích bởi vì nó làm các bộ lưu trữ khác trở nên tương thích với nó bằng cách đóng gói (encapsulate) các bộ lưu trữ khác trở thành bộ lưu trữ cơ sở của nó. Ví dụ:

```
stack<int,vector<int> > s;
```

Khi đó vector trở thành bộ lưu trữ cơ sở của bộ tương thích stack.
Nếu không khai báo bộ lưu trữ cơ sở, stack và queue mặc định sử dụng deque làm bộ lưu trữ cơ sở, trong khi priority_queue mặc định sử dụng vector làm bộ lưu trữ cơ sở, có nghĩa là khi khai báo

`stack<int> s;` thực ra là `stack<int,deque<int> > s;`

Lưu ý 2 cả stack và queue đều có các hàm sau

void push(T) thêm phần tử vào
void pop(T) gỡ phần tử ra

stack có thêm hàm T top() truy xuất phần tử ở đỉnh
queue có thêm hàm:

T front() truy xuất phần tử tiếp theo
T back() truy xuất phần tử cuối cùng của queue

priority_queue là queue trong đó phần tử đầu tiên luôn luôn là phần tử lớn nhất theo một tiêu chuẩn sắp xếp nào đó, priority_queue giống như khái niệm heap (đống) mà ta đã biết (heap và giải thuật heapsort trong môn CTDL)

Thực ra priority_queue chỉ là queue mặc định có cài sẵn thêm comparator less<T> giống như các associative container thôi. Ta có thể cài lại comparator do ta định nghĩa cho nó (ví dụ bài dưới đây cài greater<T>)

```
#include <queue>
class Plane{
   int fuel;
public:    Plane(int fuel){(*this).fuel=fuel;}
   friend ostream& operator<<(ostream& os,const Plane& p){
      os<<p.fuel<<endl;return os;}
   bool operator>(const Plane& p) const{
      return fuel>p.fuel;}
};
typedef priority_queue<Plane,vector<Plane>,greater<Plane> > PriorityQueuePlane;
int main(){
   vector<Plane> vP;
   vP.push_back(Plane(4));vP.push_back(Plane(7));
   vP.push_back(Plane(3));vP.push_back(Plane(9));
   PriorityQueuePlane v(vP.begin(),vP.end());
   while(!v.empty()){
      cout<<v.top();v.pop();
   }
    return 0;
}
```

Lưu ý là priority_queue có push, pop và top, không có front và back

2.ITERATOR ADAPTER (CÁC BỘ TƯƠNG THÍCH CON TRỎ)

Các bộ tương thích iterator thay đổi các vận hành của iterator, thường là làm container và iterator khác trở nên tương thích với nó, bằng cách đóng gói (encapsulate) các container và iterator khác trở thành container và iterator cơ sở của nó. Chúng có dạng khai báo cơ bản như sau:

```
#include<iterator>
template<class Container,class Iterator>
class IteratorAdapter
{
   //nội dung
};
IteratorAdapter<vector<int>,vector<int>::iterator> vectorIntAdapter;
```

Một số Adapter thường được sử dụng là reverse_iterator, insert_iterator, back insert iterator, front insert iterator

```
list<int> L;
L.push_front(3);
back_insert_iterator<list<int> > ii(L);
*ii++ = 0;
*ii++ = 1;
*ii++ = 2;
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 3 0 1 2
```

Không học thêm về iterator adapter

3.FUNCTION ADAPTER (CÁC BỘ TƯƠNG THÍCH HÀM)

Có 2 bộ tương thích hàm chúng ta đã học trước đó là bind1st và bind2nd. Chúng ta sắp học not1, not2, mem_fun, mem_fun_ref và ptr_fun. Tất cả đều nằm trong thư viện functional

- not1

Đổi giá trị trả về của một unary predicate từ false thành true và ngược lại, unary predicate phải được định nghĩa là unary_function
Ví dụ dùng IsOdd tìm các số chẵn (nghĩa là IsOdd trả về not(true))

```
class IsOdd:public unary_function<int,bool>{
public:bool operator()(const int& n) const{return n%2;}
};
int main(int argc, char* argv[]){
   int a[] = {1,2,3,4,5};
   cout<<count_if(a,a+5,not1(IsOdd()))<<endl;
   return 0;
}
```

- not2
  Đổi giá trị trả về của một binary predicate từ false thành true và ngược lại, binary predicate phải được định nghĩa là binary_function
  Ví dụ dùng compare để so sánh 2 mảng với các phần tử không bằng nhau (nghĩa là compare trả về not(true))

```
class compare:public binary_function<int,int,bool>{
public:bool operator()(int i,int j) const{return i==j;}
};
int main(int argc, char* argv[]){
   int a[] = {1,2,3,4,5};
   int b[] = {6,7,8,9,10};
   cout<<equal(a,a+5,b,not2(compare()))<<endl;
   return 0;
}
```

- ptr_fun
  Chuyển một con trỏ hàm (function pointer) thành một functor. Bạn đọc có thể thắc mắc functor hơn function pointer ở điểm nào mà lại cần chuyển như vậy. Câu trả lời là tốc độ:

"A suitably-defined object serves as well as - and often better than - a function. For example, it is easier to inline the application operator of a class than to inline a function passed as a pointer to function. Consequently, function objects often execute faster than do ordinary functions"- Stroustrup.

Đại ý là nếu kết hợp với từ khóa inline thì functor cho tốc độ cao hơn function

```
int addition(int a,int b){return a+b;}
int output(int a){cout<<a<<endl;return 0;}
int(*cong)(int,int) = addition;
int(*xuat)(int) = output;
int main()
{
   int a[] = {1,2,3,4,5};
   int b[] = {6,7,8,9,10};
   int c[5];
   transform(a,a+5,b,c,ptr_fun(cong));
   for_each(c,c+5,ptr_fun(xuat));
   return 0;
}
```

Ở đây chúng ta có binary function là addition và unary function là output, và binary function pointer là cong và unary function pointer là xuat, và ta dùng ptr_fun để chuyển các con trỏ hàm này thành binary functor và unary functor để đóng vai trò predicate dùng trong hai hàm transform và for_each

Xét 1 ví dụ khác:

```
#include<numeric>
double acc(double total, double elements){
   return total+elements;
}
int main(){
   multiset<double> s;
   for(int i=0;i<6;i++) s.insert(0.3);
```

```
        double sum = accumulate(s.begin(),s.end(),0.0,ptr_fun(acc));
    }
```

Ở đây dáng chú ý có hàm accumulate (của thư viện <numeric>).Hàm này được truyền vào functor acc (do ptr_fun chuyển từ function thành functor) tham số là total = 0.0 và lần lượt là các phần tử của set, sau đó acc tính tổng total và các element rồi trả về để accumulate tích lũy và cuối cùng trả giá trị ra biến sum.

ptr_fun chỉ dùng cho stand-alone và static member function, với non-static member function phải 2 hàm dưới đây:

- mem_fun và mem_fun_ref

+mem_fun
Chuyển một hàm thành viên (member function) của một lớp thành một functor và truyền vào functor này các đối số là các con trỏ mà trỏ đến các đối tượng của lớp đó

```
class Person{
    int age;
public:
    Person(int age):age(age){}
    int display(){cout<<age<<endl;return 0;}
};
int main(){
    list<Person*> l;
    l.push_back(new Person(4));
    l.push_back(new Person(5));
    for_each(l.begin(),l.end(),mem_fun(&Person::display));
    return 0;
}
```

+mem_fun_ref
Chuyển một hàm thành viên (member function) của một lớp thành một functor và truyền vào functor này các đối số là các tham chiếu mà tham chiếu đến các đối tượng của lớp đó

```
class Person{
    int age;
public:
    Person(int age):age(age){}
    int display(){cout<<age<<endl;return 0;}
};
int main(){
    list<Person> l;
    l.push_back(Person(4));
    l.push_back(Person(2));
    l.push_back(Person(5));
    for_each(l.begin(),l.end(),mem_fun_ref(&Person::display));
    return 0;
}
```

Mục đích chính là để tăng hiệu suất chương trình, thứ cực kì quan trọng trong lập trình game. Tưởng tượng bạn sẽ phải gọi 1 câu lệnh như thế này

```
for(list<Person*>::iterator i=l.begin();i!=l.end();i++) (**i).display();
```

gọi tới từng hàm thành viên của từng phần tử của list, giảm hiệu suất kinh khủng
Thay vào đó dùng mem_fun hay mem_fun_ref, chỉ cần truyền vào một con trỏ hay một tham chiếu tới hàm thành viên, tăng hiệu suất rõ rệt.

KHUYẾN CÁO: ptr_fun và mem_fun hay mem_fun_ref, cả 3 hàm này đều trả lại functor, được sử dụng rất nhiều không vì tăng tốc độ và hiệu suất chương trình. So sánh giữa các ngôn ngữ với nhau, nhờ vào những đặc điểm như con trỏ, etc, cùng với những hàm tiện ích đặc biệt trong STL nhất là 3 hàm này, để cùng đạt được một mục đích thì dùng C++ đạt được tốc độ và hiệu suất hơn bất kì ngôn ngữ bậc cao nào khác. Do đó hiểu và sử dụng nhuần nhuyễn 3 hàm này giúp tăng performance của chương trình.