TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

KHOA CÔNG NGHỆ THÔNG TIN

TRUNG TÂM TÍNH TOÁN HIỆU NĂNG CAO

NGUYỄN THANH THỦY (CHỦ BIÊN)
NGUYỄN HỮU ĐỨC
ĐẶNG CÔNG KIÊN
DOÃN TRUNG TÙNG

STL

LẬP TRÌNH KHÁI LƯỢC TRONG C++

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

KHOA CÔNG NGHỆ
THÔNG TIN

TRUNG TÂM TÍNH TOÁN
HIỆU NĂNG CAO

Nguyễn Thanh Thuỷ (Chủ biên)
Nguyễn Hữu Đức, Đặng Công Kiên, Doãn Trung Tùng

STL

Lập trình khái lược trong C++

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT
Hà Nội, 2003

MỤC LỤC

LỜI NÓI ĐẦU

Trong các cuốn sách trước đây – "Ngôn ngữ lập trình C", " Bài tập lập trình ngôn ngữ C", "Lập trình hướng đối tượng với C++" , " Bài tập lập trình ngôn ngữ C++"– chúng tôi đã có điều kiện giới thiệu tới bạn đọc những nguyên tắc cơ bản về các ngôn ngữ lập trình được xem như phổ biến nhất hiện nay. Tuy nhiên, đối với một lập trình viên, những cẩm nang tra cứu về ngôn ngữ lập trình là không đủ để có thể phát triển các ứng dụng chuyên nghiệp. Người lập trình viên giỏi là người có khả năng tổ chức xây dựng mã nguồn một cách hiệu quả, kế thừa và khai thác được tối đa thành quả của bản thân cũng như của các nhóm phát triển khác đã thực hiện mà thông thường được đóng gói dưới dạng các thư viện phần mềm. Xuất phát từ nhu cầu đó, trong cuốn sách này, chúng tôi muốn giới thiệu tới bạn đọc một thư viện khá đặc biệt của C++: thư viện khuôn hình chuẩn STL (Standard Template Library).

Sơ lược lịch sử phát triển STL

Dựa trên kỹ thuật khuôn hình (template) trong tiếp cận hướng đối tượng, Alexander Stephanov và Meng Lee của phòng thí nghiệm hãng Hewlett Parkard tại Palo Alto, California đã tiến hành xây dựng một thư viện khuôn hình cho C++ vào năm 1992, với mong muốn có được những cấu trúc dữ liệu cũng như giải thuật tổng quát nhất có được nhưng không làm mất đi tính hiệu quả. Sau đó 2 năm, vào ngày 14 tháng 7 năm 1994, các kết quả nghiên cứu và triển khai này đã được hội đồng tiêu chuẩn C++ (C++ Standards Committee) đưa vào danh sách các thư viện chuẩn của C++ theo tiêu chuẩn ANSI/ISO với tên gọi Thư viện khuôn hình chuẩn (STL). Trong giai đoạn tiếp theo, STL đã được rất nhiều tổ chức, hãng phần mềm mở rộng và phát triển trong đó đáng kể nhất phải kể đến phiên bản SGI-STL của Silicon Graphic và STLPort (hiện đã được tích hợp trong Borland C++ Builder 6). Cũng xuất phát từ ý tưởng của STL (đồng thời cũng sử dụng chính thư viện này làm nền tảng), một số thư viện khuôn hình khác ra đời phục vụ các lớp ứng dụng đặc thù hơn như thư viện khuôn hình cho ma trận MTL (trường đại học Notre Dame – Pháp) hay thư viện khuôn hình cho đồ thị GTL (trường đại học Passau – Đức). Ngày nay STL đóng góp một vị trí quan trọng

trong các ứng dụng phát triển trên C++. Điều này được thể hiện thông qua một số lượng rất lớn những ứng dụng có sử dụng STL (có thể tham khảo các ứng dụng mã nguồn mở tại địa chỉ URL SourceForge.net).

Đối tượng của cuốn sách

Cuốn sách này được chúng tôi viết trước tiên dành cho những sinh viên chuyên ngành công nghệ thông tin, cũng như các chuyên ngành khác sử dụng ngôn ngữ C++ giúp họ có được một tài liệu tham khảo tốt trợ giúp cho nhu cầu phát triển sản phẩm của mình. Cuốn sách cũng nhằm tới đối tượng là các nhà thiết kế với một góc nhìn mới của tính sử dụng lại trong phát triển phần mềm: tiếp cận về lập trình khái lược. Cuối cùng, chúng tôi cũng hy vọng cuốn sách là một tài liệu tốt dành cho các giáo viên công nghệ thông tin xây dựng những giáo trình về tiếp cận hướng đối tượng và ngôn ngữ lập trình C++.

Tổ chức cuốn sách

Cuốn sách được chia làm 6 chương và 2 phụ lục:

Chương 1. Lập trình hướng đối tượng trên C++ với STL

Nội dung chương được xem như phần nhập môn, giúp bạn đọc hiểu sơ lược về nền tảng cơ sở, cấu trúc và cách sử dụng của thư viện khuôn hình chuẩn STL.

Chương 2 : Bộ chứa

Giới thiệu những thư viện khuôn hình đóng vai trò khuôn mẫu tạo dựng các cấu trúc dữ liệu cơ bản như array, vector, list, stack, queue, map,...

Chương 3 : Bộ duyệt

Giới thiệu các khuôn hình đóng vai trò trừu tượng hóa những giải thuật truy nhập cấu trúc dữ liệu (cụ thể là các bộ chứa)

Chương 4 : Đối tượng hàm

Giới thiệu các khuôn hình trợ giúp đóng gói các toán tử thành lớp

Chương 5 : Giải thuật

Giới thiệu thư viện giải thuật được khuôn hình hóa nhằm sử dụng cho nhiều cấu trúc dữ liệu khác nhau.

Chương 6: Lập trình khái lược với STL

Bổ sung một quan điểm, một phương pháp luận cho các nhà thiết kế về tính sử dụng lại của sản phẩm phần mềm.

Phụ lục A: Các khái niệm trong STL

Thư viện STL được nhìn dưới góc độ của người tổ chức thiết kế thư viện (đối lập với cách nhìn của người sử dụng trong các chương của cuốn sách)

Phụ lục B: Một số thư viên khuôn hình khác

Thư viện MTL

Thư viện các khuôn hình cho cấu trúc dữ liệu ma trận với những biểu diễn khái lược về ma trận dầy, ma trận thưa, các giải thuật liên quan tới ma trận.

Thư viện GTL

Thư viện các khuôn hình biểu diễn cấu trúc dữ liệu đồ thị (graph) và các giải thuật cơ bản trên đồ thị.

Phụ lục C: Các trang Web hữu ích về STL

Chúng tôi hy vọng cuốn sách sẽ đem lại những lợi ích thiết thực cho bạn đọc. Những thắc mắc và đóng góp có thể gửi về nhóm tác giả theo địa chỉ:

thuytt@it-hut.edu.vn : chủ biên Nguyễn Thanh Thủy

hpcc@mail.hut.edu.vn : Nhóm tác giả

Trong quá trình biên soạn cuốn sách, nhóm tác giả nhận được sự động viên và những đóng góp chuyên môn rất quan trọng của các Giáo sư, các chuyên gia, các Thầy Cô trong khoa Công nghệ thông tin, trường Đại học Bách khoa Hà Nội. Chúng tôi xin chân thành cảm ơn sự đóng góp quý báu đó. Chúng tôi cũng xin bày tỏ lòng biết ơn tới ban Chủ nhiệm khoa Công nghệ thông tin, Chương trình Kinh tế Kỹ thuật về Công nghệ thông tin, Nhà xuất bản Khoa học và Kỹ thuật đã tạo điều kiện về tinh thần và vật chất để cuốn sách sớm ra mắt phục vụ độc giả.

Cuốn sách còn là kết quả trao đổi chuyên môn với các giáo sư thuộc Khoa Tin học, trường Đại học Winsconsin – Milwaukee, Mỹ trong khuôn khổ chương trình trao đổi học giả Fulbright mà tác giả Nguyễn Thanh Thuỷ đã tham gia. Nhân dịp này, tác giả xin trân trọng cảm ơn sự hỗ trợ quý báu của chương trình Fulbright Việt Nam.

Hà Nội ngày 20/4/2002
Thay mặt nhóm tác giả
PGS. TS. Nguyễn Thanh Thủy

Chương 1

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG TRÊN C++ VỚI STL

Mục đích chương này

- Tìm hiểu động cơ phát triển thư viện STL và vai trò của STL trong các ứng dụng C++.
- Nhắc lại những nguyên lý cơ bản trong lập trình hướng đối tượng trên C++
- Làm quen với STL thông qua một vài ví dụ đơn giản

1.1. STL – thư viện khuôn hình chuẩn

Kể từ khi ra đời tại phòng thí nghiệm Bell vào năm 1984, C++ đã dần dần chiếm lĩnh được lòng tin người sử dụng và trở nên một trong những ngôn ngữ được ưa chuộng và được áp dụng nhiều trong cả giảng dạy, nghiên cứu cũng như phát triển ứng dụng thực tiễn. Sự kết hợp những kỹ thuật mới theo tiếp cận hướng đối tượng cùng với việc bảo toàn các đặc điểm cũ của C – ngôn ngữ vốn đã rất phổ biến – đã giúp cho C++ có được những lợi thế hơn so với một số ngôn ngữ khác. Khả năng mô đun hóa cao cũng như tính sử dụng lại trong các ứng dụng C++ được thể hiện thông qua các kỹ thuật đóng gói (encapsulation) , kế thừa (inheritance) và đa hình (polymorphism). Hơn thế nữa, sự bổ sung khái niệm khuôn hình trong đầu những năm 90 giúp C++ trở thành một công cụ lý tưởng cho những nhà phát triển theo trường phái hướng đối tượng.

Tuy nhiên, với một ngôn ngữ "lai" như C++, lập trình viên luôn luôn vấp phải những trở ngại trong việc thống nhất những khái niệm như biến và đối tượng hay kiểu dữ liệu và lớp. Một ví dụ điển hình là khi xây dựng một danh sách móc nối các đối tượng, các ngôn ngữ thuần đối tượng như Java hay Smalltalk thường không quan tâm tới đối tượng thành viên của danh sách là loại gì (thuộc lớp nào), nhưng với C++ thì hoàn toàn khác. C++ yêu cầu phải chỉ rõ kiểu dữ liệu (lớp) cho các đối tượng tham gia. Trở ngại này dẫn đến việc lập trình viên phải tạo ra nhiều lớp danh sách móc nối tương ứng với các

kiểu dữ liệu khác nhau. Sự có mặt của khuôn hình trong C++ giúp người lập trình giải quyết được vấn đề này khá hiệu quả bằng cách trừu tượng hóa kiểu dữ liệu cho một lớp hoặc một hàm. Vấn đề nảy sinh ở đây là làm thế nào xây dựng được những khuôn hình tổng quát áp dụng cho mọi kiểu dữ liệu nhưng lại không mất đi tính hiệu quả so với những lớp , hàm trên kiểu dữ liệu cụ thể. Đây chính là xuất phát điểm cho những nghiên cứu về lập trình khái lược (generic programming) và phát triển thư viện khuôn hình chuẩn (STL) của Alexander Stephanov và Meng Lee. Phiên bản đầu tiên của STL ra đời trong phòng thí nghiệm của Hewlett Parkard và ngay sau đó được chính thức công nhận là một trong những thư viện chuẩn của C++ (ANSI/ISO C++) vào tháng 7 năm 1994.

Khác với các thư viện khác như OWL của Borland hay MFC của Microsoft, STL không tập trung vào phát triển những thư viện đối tượng đặc thù của môi trường phát triển (ví dụ TCanvas, CWnd, ...) mà tập trung vào phát triển các thư viện khuôn hình cho các kiểu dữ liệu tập hợp (vector, list, map,...) và các giải thuật tổng quát (sắp xếp, tìm kiếm...). Điều này cho phép STL có thể sử dụng trong nhiều môi trường phát triển khác nhau và đây cũng chính là lý do để STL trở thành một trong những thư viện chuẩn.

Để bạn đọc có thể bắt đầu làm quen với STL, chúng tôi xin đưa ra một ví dụ nhỏ:

```
// Sắp xếp trên danh sách

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

int main( )
{
   using namespace std;
   vector <int> v;
   vector <int>::iterator It;

   // Khởi tạo danh sách
   v.push_back(3);
   v.push_back(8);
   v.push_back(5);
   v.push_back(4);
   v.push_back(2);

   // In danh sách chưa sắp xếp
   cout << "Danh sach chua sap xep: ( " ;
```

```
    for ( It = v.begin( ) ; It != v.end( ) ; It++ )
       cout << *It << " ";
    cout << ")" << endl;

    // Sắp xếp danh sách
    sort( v.begin( ), v.end( ) );

    // In danh sách đã được sắp xếp
    cout << "Danh sach da sap xep: ( " ;
    for ( It = v.begin( ) ; It != v.end( ) ; It++ )
       cout << *It << " ";
    cout << ")" << endl;

 }
```

```
Danh sach chua sap xep: ( 3 8 5 4 2 )
Danh sach da sap xep: ( 2 3 4 5 8 )
```

Một số điểm cần nhấn mạnh trong chương trình trên là:

- Khi sử dụng STL cần lưu ý bổ sung các tệp tiêu đề (header) cho các cấu trúc dữ liệu và giải thuật sử dụng trong chương trình.

  #include <vector>

  #include <algorithm>

- Sử dụng tên vùng *std* cho các cấu trúc dữ liệu hoặc giải thuật STL

  using namespace std;

- Cụ thể hóa các khuôn hình bằng cách gắn chúng với kiểu dữ liệu cụ thể

  vector<int> v;

- Sử dụng khái niệm bộ duyệt thay thế cho con trỏ

  for (It = v.begin() ; It != v.end() ; it++)

  *cout << *It << " ";*

- Sử dụng giải thuật tổng quát thay cho việc tự phát triển.

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

Thực tế, STL làm được nhiều hơn những gì bạn đọc quan sát được trong ví dụ trên. Tuy nhiên, trước khi giúp bạn đọc có được một cái nhìn tương đối toàn diện về bộ thư viện này trong các chương kế tiếp, chúng tôi sẽ

trình bày sơ bộ nền tảng cơ sở, cấu trúc và cách sử dụng bộ thư viện trong các mục tiếp theo của chương.

1.2. Tiếp cận hướng đối tượng trong C++

Trong lịch sử phát triển, ngôn ngữ Simula I (1965) của nhóm tác giả Ole-Johan Dahl và Kristen Nygaard (trung tâm tính toán Nauy - Oslo - Nauy) được xem như kết quả đầu tiên đầu tiên áp dụng tiếp cận hướng đối tượng trong ngôn ngữ lập trình. Kể từ đó, tiếp cận hướng đối tượng liên tục được phát triển và hoàn thiện. Năm 1984 đánh dấu sự ra đời của C++ (Bjarne Stroustrup - Bell Labs) với sự kết hợp giữa những kỹ thuật trong tiếp cận hướng đối tượng với phong cách lập trình cấu trúc cổ điển của C. Nói về tiếp cận hướng đối tượng, người ta không thể không nhắc tới tính đóng gói (encapsulation), tính sử dụng lại (reuse) , tính dễ mở rộng (extensibility) và tính thích nghi (adaptability). Những tính chất này được phản ánh trong C++ thông qua các khái niệm về lớp (class), kỹ thuật kế thừa (inheritance), tính đa hình (polymorphism) và sau này là khuôn hình (template) .

1.2.1. Đối tượng và lớp

Đối tượng luôn là trọng tâm chính trong tiếp cận hướng đối tượng. Một đối tượng thể hiện sự đóng gói của các đặc điểm, các thuộc tính cũng như những hoạt động liên quan tới một thực thể hoặc một khái niệm nào đó trong thế giới thực. Nếu như tiếp cận cấu trúc tìm cách tách biệt giữa đặc trưng dữ liệu và các thao tác xử lý trên dữ liệu, thì ngược lại, trong tiếp cận hướng đối tượng, chúng lại được nhóm lại và tham chiếu tới cùng một tên gọi. Hãy xem xét một ví dụ đơn giản sau để có được sự so sánh về hai tiếp cận này:

```
// Tiếp cận cấu trúc
IntList l; // Định nghĩa cấu trúc dữ liệu danh sách móc nối
int total;
...
// Bổ sung 3 số nguyên vào danh sách
IntList_push_back(l,1);
IntList_push_back(2,l);
IntList_push_back(3,l);
// Tính toán tổng giá trị các phần tử trong danh sách
total = IntList_sum(l);
...
// Tiếp cận hướng đối tượng
```

```
IntList l; // Định nghĩa đối tượng danh sách móc nối
int total;
...
// Bổ sung 3 số nguyên vào danh sách
l.push_back(1);
l.push_back(2);
l.push_back(3);
// Tính toán tổng giá trị các phần tử trong danh sách
total = l.sum(l);
...
```

Rõ ràng, việc gắn các đặc trưng dữ liệu (nội dung danh sách) với các thao tác xử lý (push_back, sum) trong tiếp cận hướng đối tượng cho phép lập trình viên thể hiện được tính chất mô đun hóa của chương trình. Khi lập trình, họ chỉ cần tập trung quan tâm tới các đặc tính và hành vi của đối tượng đang được phát triển.

Trong C++, mỗi đối tượng được tạo ra nhờ một "khuôn" gọi là lớp (class). Lớp được xem như sự trừu tượng hóa của các đối tượng. Ngược lại, đối tượng lại là các thể hiện cụ thể của lớp. Khi xây dựng một chương trình, lập trình viên định nghĩa các lớp, sau đó sẽ sử dụng các lớp này để tạo ra các đối tượng tham gia. Lớp trong C++ được định nghĩa theo cú pháp cơ bản như sau:

```
class <<Tên lớp>> {
     <<phạm vi>>
            <<định nghĩa thành phần dữ liệu>>
            <<định nghĩa hàm thành phần >>
};
```

Thành phần dữ liệu

Là phần định nghĩa những đặc trưng dữ liệu của các đối tượng thuộc về lớp. Trong C++ thành phần dữ liệu có thể là các số, xâu ký tự hoặc thậm chí là các đối tượng thuộc về các lớp khác.

Hàm thành phần

Là phần định nghĩa những đặc trưng xử lý của các đối tượng thuộc về lớp. Hàm thành phần có thể là một trong các dạng sau :

- Hàm thiết lập (constructor) : Hàm được thực hiện ngay khi đối tượng được tạo ra (cấp phát).
- Hàm hủy bỏ (destructor) : Hàm được thực hiện trước khi đối tượng được hủy (giải phóng).

- Toán tử (operator) : là hàm thành phần gắn liền với các phép xử lý toán học mà đối tượng của lớp có thể tham gia
- Hàm thành phần thông thường

Phạm vi

Các thành phần dữ liệu và hàm thành phần của một đối tượng đôi khi chỉ được sử dụng cho những mục đích phát sinh nội tại bên trong đối tượng. Khi đó những thành phần này không cần thiết công bố cho các đối tượng khác. Để đảm bảo tính đóng gói, khi xây dựng lớp tương ứng, lập trình viên sẽ chỉ định thành phần nào được công bố, thành phần nào cần che giấu thông qua khái niệm phạm vi. Mỗi thành phần của đối tượng có thể có các phạm vi :

- ***public*** : Có thể sử dụng, truy nhập từ các đối tượng bên ngoài
- ***private*** : Chỉ sử dụng cho các mục đích cục bộ cho các đối tượng thuộc về lớp.
- ***protected*** : Chỉ sử dụng cho mục đích cục bộ cho các đối tượng thuộc về lớp hoặc các lớp kế thừa nó.

```
// Định nghĩa lớp point
#include <iostream.h>

class point {
protected:
     // Thành phần dữ liệu
int x,y;
public :
     // Hàm thiết lập
     point(){
            x = 0;
            y = 0;
};
point(int iX,int iY) {
            x = iX;
            y = iY;
};
// Hàm hủy
     ~point() {
            cout << "Doi tuong da duoc huy bo \n";
     };
     // Hàm thành phần
     void move(int dx,int dy);
     void display();

};
```

```
void point::move(int dx,int dy) {
     x += dx;
     y += dy;
};

void point::display() {
     cout.<< "(" << x << "," << y << ")\n";
};

void main() {
     // p1 : cấp phát động, p2 : cấp phát tĩnh
     point *p1 = new point;
     point p2(10,10);

     // di chuyển p1 với độ dời 20,20
     p1->move(20,20);
     return 0;

      }
```

```
(20,20)
(10,10)
Doi tuong da duoc huy bo
```

Ví dụ trên cho thấy cách định nghĩa và sử dụng các lớp trong C++. Các đối tượng của lớp point có thể được khởi tạo theo 2 cách : mặc định với tọa độ ban đầu (0,0) hoặc được gán ngay tọa độ ban đầu (iX,iY). Trước khi các đối tượng point được hủy, hàm hủy của đối tượng sẽ được kích hoạt.

1.2.2. Thừa kế

Thừa kế là kỹ thuật nhằm nâng cao tính sử dụng lại của chương trình. Lập trình viên có thể xây dựng một lớp mới (lớp dẫn xuất) bằng cách thừa kế một lớp đã xây dựng (lớp cơ sở). Khi đó các đối tượng của lớp dẫn xuất có thể thừa kế các thành phần dữ liệu hoặc thành hàm thành phần của các đối tượng của lớp cơ sở.

```
class <<lớp dẫn xuất>> : <<phạm vi lớp>> <<lớp cơ sở>> {
     <<định nghĩa các thành phần dữ liệu
  và hàm thành phần bổ  sung>>
};
```

Phạm vi lớp

Phạm vi lớp bổ sung thêm khái niệm phạm vi cho các thành phần dữ liệu hoặc hàm thành phần trong trường hợp một lớp kế thừa từ lớp khác.

- ***public*** : các thành phần của lớp cơ sở giữ nguyên phạm vi trong lớp dẫn xuất
- ***private***: các thành phần public và protected trong lớp cơ sở trở thành private trong lớp dẫn xuất.

Định nghĩa lại hàm thành phần

Khi thừa kế từ một lớp cơ sở, lớp dẫn xuất có thể có định nghĩa lại một vài hàm thành phần không còn hợp với lớp mới nữa. Khi đó, thay vì sử dụng hàm thành phần được thừa kế, hàm thành phần định nghĩa lại sẽ được áp dụng.

```
...
// Kế thừa từ lớp point, bổ sung thêm thuộc tính color
class colored_point : public point {
protected:
     int color;
public:
     // Hàm thiết lập
     colored_point() : point() {
            color = BLACK;
     };
     colored_point(int iX,int iY,int iC) : point(iX,iY) {
            color = iC;
     };
     // Định nghĩa lại hàm display
     void display();
};

void colored_point::display() {
     cout << "(" << x << "," << y << ":" << color << ")\n";
}
...
```

Trong ví dụ trên, hàm thành phần `display()` được định nghĩa lại, do vậy các đối tượng thuộc lớp `colored_display` thay vì in ra 2 tọa độ điểm, chúng sẽ in ra tọa độ điểm và mầu.

1.2.3. Hàm ảo và tính đa hình

Việc cho phép định nghĩa lại hàm thành phần là cơ sở của một kỹ thuật trong tiếp cận hướng đối tượng gọi là đa hình (polymorphism). Kỹ thuật này cho phép một đối tượng, tùy từng tình huống cụ thể có thể đưa ra những ứng xử thích hợp.

```
...
// Lớp cơ sở với hàm thành phần ảo area()
class polygon {
...
virtual double area() {
      return 0;
};
...
};
...
// Hàm thành phần ảo được định nghĩa lại trong lớp triangle
class triangle : public polygon {
      ...
      double area() {
            double p = (a + b + c )/2.0;
            return sqrt( p * (p-a) * (p-b) * (p-c) );
      };
      ...
};
// Hàm thành phần ảo được định nghĩa lại trong lớp rectangle
class rectangle : public polygon {
      ...
      double area() {
            return w * h;
      };
      ...
};
...
void main() {
      polygon *p1,*p2;
      p1 = new triangle(3,4,5);
      p2 = new rectangle(4,5);
      cout << p1->area() << "\n";
      cout << p2->area() << "\n";
      ...
};
```

```
...
6
20
...
```

Lưu ý, nếu không có từ khóa `virtual`, các kết quả in ra sẽ là 0,0 thay vì 6 và 20 do khi đó nó vẫn sử dụng hàm thành phần của lớp `polygon`. Từ khóa `virtual` cho phép thay đổi động các lời gọi khi đối tượng được gán cho một lớp cụ thể. Trong ví dụ, khi gán p1 cho một đối tượng `triangle`, thì hàm `area()` sẽ là hàm tính diện tích tam giác, nhưng khi gán p2 cho một đối tượng `rectangle` thì hàm này lại là hàm tính diện tích hình chữ nhật. Tính đa hình ở đây cho phép lập trình viên xử lý khá linh động các tình huống nhập nhằng như vậy mà không cần phải dùng tới phép thử if.

1.2.4. Khuôn hình

Nếu như kỹ thuật thừa kế cho phép lập trình viên nâng cao tính sử dụng lại bằng cách thừa kế những lớp đã xây dựng để tạo ra lớp mới thì khuôn hình lại làm tăng tính sử dụng lại trong trường hợp tổng quát hóa một lớp, một hàm cho nhiều kiểu dữ liệu khác nhau. Khuôn hình có thể là một hàm hay một lớp tác động trên một hoặc nhiều kiểu dữ liệu trừu tượng nào đó (gọi là các tham số khuôn hình). Khi sử dụng một khuôn hình, lập trình viên sẽ gán cho tham số khuôn hình những kiểu dữ liệu cụ thể để thu được những hàm, những lớp thể hiện của khuôn hình. Điều đáng nói ở đây là với cùng một khuôn hình được định nghĩa có thể tạo ra một họ các hàm, lớp thể hiện tương ứng với các kiểu dữ liệu đưa vào cho các tham số.

Khuôn hình được chia ra làm hai dạng: Khuôn hình hàm và khuôn hình lớp.

Khuôn hình hàm

Khuôn hình hàm là một hàm hoạt động trên các kiểu dữ liệu trừu tượng.

```
template <[tham số khuôn hình]>
[kiểu dữ liệu trả về]  [tên hàm]([tham số hàm]) {
            [định nghĩa hàm]
}
```

Khi sử dụng một khuôn hình hàm, lập trình viên chỉ cần gọi hàm với các tham số truyền vào. Tùy thuộc vào các tham số này có kiểu dữ liệu gì mà khuôn hình hàm sẽ được cụ thể hóa thành hàm thể hiện tương ứng.

```
#include <iostream.h>

// Định nghĩa khuôn hình hàm mymin
template <class T>
T& mymin(T& a,T& b) {
     if (a > b) return b;
     else return a;
};

class

void main() {
     int x1=10,y1=20;
     double x2=12.5,y2=14.3;


// Chương trình dịch sẽ tạo ra hai hàm thể hiện của

// khuôn hình tương ứng với các kiểu dữ liệu int và double
     cout << mymin(x1,y1) << "\n";
```

```
    cout << mymin(x2,y2) << "\n";
};
```

```
10
12.5
```

Trong ví dụ trên, khuôn hình hàm không những chỉ hoạt động với các kiểu dữ liệu định nghĩa sẵn mà còn có thể hoạt động với bất kỳ kiểu dữ liệu nào miễn là chúng có toán tử so sánh '>'

```
#include <iostream.h>
#include <string.h>

template <class T>
T& mymin(T& a,T& b) {
    if (a > b) return b;
    else return a;
};

class Personal {
private:
    char *name;
    char *family_name;
public:
    Personal(char *n,char *f) {
        name = n;
        family_name = f;
    };
// Định nghĩa toán tử >
    int.operator > ( Personal &p) {
        if ( strcmp(name,p.name) == 0 )
          return (strcmp(family_name,p.family_name)>=0);
        else return (strcmp(name,p.name)>=0);
    };
    void display() {
        cout << family_name << " " << name << "\n";
    };
};

void main() {
    Personal a("Anh","Nguyen Tuan");
    Personal b("Anh","Ta Tuan");

    mymin(a,b).display();
};
```

```
Nguyen Tuan Anh
```

Lưu ý là trong ví dụ nếu không có toán tử > trong lớp `Personal` chương trình dịch sẽ báo lỗi.

Cụ thể hóa hàm thể hiện

Một khuôn hình hàm cho phép định nghĩa một họ hàm tương ứng với các tham số khuôn hình. Tuy nhiên, không phải bất kỳ kiểu dữ liệu nào cũng phù hợp làm tham số. Trong ví dụ, kiểu dữ liệu làm tham số bắt buộc phải hỗ trợ toán tử >. Một số kiểu dữ liệu không hỗ trợ toán tử này. Khi đó để sử dụng hàm với cùng tên gọi, lập trình viên phải xây dựng những hàm thể hiện đặc thù cho chúng.

```
#include <iostream.h>
#include <string.h>

template <class T>
T& mymin(T& a,T& b) {
     if (a > b) return b;
     else return a;
};

// Hàm thể hiện đặc thù
char* &mymin(char * &a, char* &b) {
if (strcmp(a,b)> 0 ) return b;
     else return a;
};

void main() {
     char *a="Nguyen Tuan Anh";
     char *b="Ta Tuan Anh";

     cout << mymin(a,b);
};
```

```
Nguyen Tuan Anh
```

Khi hàm mymin(char*,char*) được yêu cầu trong chương trình, thay vì tạo ra một thể hiện của khuôn hình hàm như trong các ví dụ trước, ở đây chương trình dịch sẽ sử dụng luôn hàm thể hiện đặc thù đã định nghĩa.

Khuôn hình lớp

Giống như khuôn hình hàm, khuôn hình lớp cho phép lập trình viên tạo ra một họ lớp tùy thuộc tham số khuôn hình.

```
template <[tham số khuôn hình]>
class [tên lớp] {
     [định nghĩa các thành phần của lớp]
};
```

Nếu như trong khuôn hình hàm chương trình dịch sẽ tự động phát hiện các tham số khuôn hình thì đối với khuôn hình lớp, lập trình viên phải chỉ định ra các tham số này.

```
#include <iostream.h>

// Định nghĩa khuôn hình lớp array5 với kiểu dữ liệu trừu
//   tượng T
template <class T>
class array5 {
protected :
    T items[5];
public:
    array5() {};
    void setAt(int index,T value);
    T getAt(int index);
};

// Định nghĩa thao tác gán giá trị cho một thành phần mảng
template <class T>
void array5<T>::setAt(int index,T value) {
    items[index] = value;
};

// Định nghĩa thao tác đọc giá trị từ một thành phần mảng
template <class T>
T array5<T>::getAt(int index) {
    return items[index];
};


void main() {
// Áp dụng khuôn hình array5 cho 2 kiểu dữ liệu khác nhau
    array5<int> stt;
    array5<char*> ten;
    int i;

    stt.setAt(0,1);ten.setAt(0,"An");
    stt.setAt(1,2);ten.setAt(1,"Hoa");
    stt.setAt(2,3);ten.setAt(2,"Ha");
    stt.setAt(3,4);ten.setAt(3,"Giang");
    stt.setAt(4,5);ten.setAt(4,"Tuan");

    for (i= 0 ; i < 5 ; i++)
     cout << stt.getAt(i) << "  " << ten.getAt(i) << "\n";
    cin >> i;

};
```

```
1 An
2 Hoa
3 Ha
4 Giang
5 Tuan
```

Cũng giống như với khuôn hình hàm, không phải bất kỳ kiểu dữ liệu nào cũng có thể trở thành tham số khuôn hình. Các kiểu dữ liệu làm tham số khuôn hình phải hỗ trợ đầy đủ các toán tử hoặc các hàm thành phần đã được xác định trong phần định nghĩa khuôn hình. Trong ví dụ trên, để triển khai các hàm thành phần `setAt()` và `getAt()`, kiểu dữ liệu T phải hỗ trợ toán tử gán.

Cụ thể hóa khuôn hình lớp

Nếu kiểu dữ liệu làm tham số không phù hợp với một số hàm thành phần của khuôn hình, ta có thể định nghĩa lại các hàm thành phần này cho những kiểu dữ liệu đó.

```
#include <iostream.h>
#include <string.h>

// định nghĩa khuôn hình

template <class T>
class array5 {
protected :
    T items[5];
public:
    array5() {};
    void setAt(int index,T value);
    T getAt(int index);
    T sum();
};

template <class T>
void array5<T>::setAt(int index,T value) {
    items[index] = value;
};

template <class T>
T array5<T>::getAt(int index) {
    return items[index];
};

template <class T>
T array5<T>::sum() {
    T result;
```

```
    result = items[0];
    for (int i = 1 ; i < 5 ; i ++ )
          result += items[i];
    return result;
};

// Định nghĩa lại hàm thành phần sum()
//   cho kiểu dữ liệu char*
char* array5<char*>::sum() {
    char* result= new char[100];
    strcpy(result,"");
    for (int i = 1 ; i < 5 ; i ++ ) {
          strcat(result,items[i]);
          strcat(result," ");
    };
    return result;
};

void main() {
    array5<int> stt;
    array5<char*> ten;

    stt.setAt(0,1);ten.setAt(0,"An");
    stt.setAt(1,2);ten.setAt(1,"Hoa");
    stt.setAt(2,3);ten.setAt(2,"Ha");
    stt.setAt(3,4);ten.setAt(3,"Giang");
    stt.setAt(4,5);ten.setAt(4,"Tuan");

    cout << stt.sum() << "\n";
    cout << ten.sum() << "\n";
};
```

```
15
An Hoa Ha Giang Tuan
```

Trong ví dụ trên, hàm thành phần sum() được định nghĩa lại cho riêng trường hợp tham số khuôn hình là char*. Trong trường hợp này, thay vì sử dụng phép cộng như định nghĩa trong khuôn hình, hàm thành phần định nghĩa lại sẽ nối các xâu ký tự trong mảng lại với nhau.

1.3. Làm quen với thư viện khuôn hình chuẩn STL

STL bao gồm năm thành phần chính:

- Bộ chứa (containers) : là các cấu trúc dữ liệu lưu trữ nhóm các đối tượng đồng nhất với các tổ chức khác nhau như tập hợp (set) , mảng (array), danh sách liên kết (list), mảng kết hợp (map),...

- Bộ duyệt (iterators) : Thông thường để duyệt các phần tử trong bộ chứa lập trình viên thường sử dụng biến chỉ số (với mảng) hoặc con trỏ. STL đề xuất một kiến trúc tương đối tổng quát về phép duyệt thông qua các bộ duyệt. Với bộ duyệt, lập trình viên có thể có áp dụng những nguyên lý duyệt giống nhau trên các bộ chứa khác nhau

- Giải thuật (algorithms) : thực chất là các khuôn hình hàm cho phép thực hiện các giải thuật tính toán, tìm kiếm, sắp xếp,... trên các bộ chứa khác nhau.

- Đối tượng hàm (functors) : Là một dạng lớp đặc biệt thay mặt cho một hàm hoặc một toán tử. Với đối tượng hàm, một hàm hoặc toán tử có thể được cụ thể hóa bằng một đối tượng, do đó tạo ra sự linh động cao trong việc phát triển các giải thuật.

Ví dụ : Để sắp xếp danh sách nhân viên, người ta có thể lựa chọn nhiều tiêu chuẩn sắp xếp khác nhau : Sắp xếp theo họ tên, sắp xếp theo bậc lương, sắp xếp theo trình độ. Mỗi cách sắp xếp đòi hỏi phải có một hàm phân thứ tự giữa hai đối tượng được sắp xếp. Khi đó đối tượng hàm sẽ là đối tượng tham số cho hàm sắp xếp và đại diện cho hàm phân thứ tự này.

STL hỗ trợ những khuôn hình lớp để xây dựng các đối tượng hàm như vậy.

- Bộ thích nghi (adaptors) : Khi một hệ thống kế thừa, sử dụng một vài thành phần của một hệ thống cũ, để dễ dàng tích hợp người ta tạo ra những giao diện (interface) giữa hệ thống cũ và hệ thống mới. STL đưa ra những tiêu chuẩn cho phép xây dựng bộ thích nghi cho các bộ chứa, bộ duyệt và đối tượng hàm.

Để làm quen với các khái niệm này, chúng ta hãy phân tích một ví dụ đơn giản sau:

```
QUẢN LÝ NHÂN SỰ
Để quản lý nhân viên trong một công ty, phòng nhân sự cần lưu
trữ một danh sách hồ sơ nhân viên. Mỗi nhân viên có một hồ sơ
bao gồm các thông tin: Họ tên, lương, giới tính. Các nhu cầu
thống kê bao gồm :

- Lập danh sách nhân viên theo thứ tự họ tên, lương,

- Xác định nhân viên nhận lương cao nhất, thấp nhất,
- Tìm kiếm nhân viên theo họ tên.
```

Trước tiên, để quản lý thông tin cho mỗi nhân viên ta sử dụng một đối tượng thuộc lớp `Person`:

```
class Person {
protected :
// Thành phần dữ liệu của một nhân viên
     char name[40];
     double salary;
     char sex;
public:
     Person() {};

// Nhập dữ liệu và hiển thị thông tin nhân viên
     void input();
     void display();
};
```

Để quản lý danh sách hồ sơ nhân sự với các chức năng thống kê như yêu cầu đã đặt ra, ta xây dựng một lớp `Persons`.

```
class Persons {
protected:
     // Danh sách hồ sơ được lưu trữ nhờ khuôn hình vector
vector<Person> items;

public:
     Persons() {};

// hàm thành phần nhập liệu cho danh sách
     void input();

     // Lập danh sách với thứ tự tăng dần theo tên
     void displayByName();
// Lập danh sách với thứ tự tăng dần theo lương
     void displayBySalary();

     // Tìm kiếm nhân vân lương cao nhất
     void searchHighestSalary();
     // Tìm kiếm nhân viên lương thấp nhất
     void searchLowestSalary();
     // Tìm kiếm nhân viên theo tên
     void searchByName(char *n);
};
```

Do sử dụng khuôn hình `vector` với tham số khuôn hình là `Person` nên một điều kiện bắt buộc là lớp `Person` phải bổ sung thêm toán tử gán :

```
class Person {
...
     Person &operator =(const Person &p);
...
};
```

Để nhập liệu cho danh sách hồ sơ, lớp `Persons` có hàm thành phần `input()`:

```
void Persons::input() {
     Person p;
     char ch;
     cout << "Enter Person information.\n";
     do {
            p.input();
            items.push_back(p);
            cout << "More(Y/N)?" ;
            cin >> ch;
     } while (ch == 'Y');
};
```

trong đó `push_back()` là hàm thành phần của lớp `vector<Person>` được xây dựng bởi STL cho phép bổ sung thêm một phần tử mới vào sau danh sách.

Để hiển thị danh sách nhân viên được sắp xếp theo tên, lớp `Persons` có hàm thành phần `displayByName()`

```
void Persons::displayByName() {
     smByNameFunctor func;

     sort(items.begin(),items.end(),func);
     display();
};
```

Hàm thành phần này sử dụng giải thuật sắp xếp sẵn có của STL `sort()`, tuy nhiên để giải thuật hoạt động được, cần định nghĩa thứ tự trước sau theo tên của 2 đối tượng `Person`. Đối tượng hàm `func` chính là thành phần định nghĩa thứ tự trước sau đó.

```
class ltByNameFunctor {
public :
     ltByNameFunctor() {};

     int operator()(Person &p1,Person &p2) {
```

```
            if ( strcmp(p1.name,p2.name) < 0 ) return 1;
            else return 0;
    };
};
```

Đối tượng hàm này tương đương với một hàm nhận hai tham số là hai đối tượng thuộc lớp `Person` và trả về giá trị lôgic mô tả thứ tự trước sau của hai đối tượng đó.

Để hiển thị kết quả sau khi sắp xếp, hàm thành phần `display()`được xây dựng như sau :

```
void Persons::display() {
    displayFunctor func;
    for_each(items.begin(),items.end(),func);
};
```

Ở đây ta lại bắt gặp một giải thuật khác của STL. Thay vì sử dụng cấu trúc lặp for như thông thường, ta có thể sử dụng giải thuật duyệt `for_each` trong đó khi duyệt qua mỗi phần tử của danh sách, thao tác hiển thị sẽ được thực hiện nhờ đối tượng hàm `func`.

Với chức năng hiển thị theo thứ tự tăng dần trên lương, ta chỉ cần thay thế đối tượng hàm `func` :

```
void Persons::displayBySalary() {
    smBySalaryFunctor func;

    sort(items.begin(),items.end(),func);
    display();
};
```

Ta cũng ứng dụng chính đối tượng hàm này trong các chức năng tìm kiếm nhân viên có lương thấp nhất và cao nhất:

```
void Persons::searchHighestSalary() {
    smBySalaryFunctor func;
    vector<Person>::iterator it;

    it = max_element(items.begin(),items.end(),func);
    it->display();
};


void Persons::searchLowestSalary() {
    smBySalaryFunctor func;
    vector<Person>::iterator it;
```

```
    it = min_element(items.begin(),items.end(),func);
    it->display();
};
```

max_element() và min_element() là hai giải thuật của STL cho phép tìm kiếm phần tử lớn nhất và nhỏ nhất trên một bộ chứa theo tiêu chuẩn được đặt ra bởi func.

Để tìm kiếm chính xác một phần tử nào đó theo tên, hàm find_if() của STL được áp dụng như sau :

```
void Persons::searchByName(char *n) {
    eqByNameFunctor func(n);
    vector<Person>::iterator it;

    it = find_if(items.begin(),items.end(),func);
    it->display();
};
```

Cuối cùng, chương trình cho yêu cầu đặt ra của bài toán được viết lại :

```
#include <vector>
#include <algorithm>
#include <iostream.h>
#include <string.h>

using namespace std;

class Person {
protected :
    char name[40];
    double salary;
    char sex;
public:
    Person() {};

    void input();
    void display();

    Person &operator =(const Person &p);


    friend class displayFunctor;

    friend class smByNameFunctor;

    friend class smBySalaryFunctor;
    friend class eqByNameFunctor;
};
```

```
void Person::input() {
     cout << "Name:"; cin >> name;
     cout << "Salary:"; cin >> salary;
     cout << "Sex:"; cin >> sex;
};

void Person::display() {
     cout << name << "," << salary << "," << sex << "\n";
};

Person &Person::operator = (const Person &p) {
     strcpy(name,p.name);
     salary = p.salary;
     sex = p.sex;
     return (*this);
};

class displayFunctor {
public :
     displayFunctor() {};
     void operator()(Person &p) {
            p.display();
     };
};
class smByNameFunctor {
public :
     smByNameFunctor() {};

     int operator()(Person &p1,Person &p2) {
            if ( strcmp(p1.name,p2.name) < 0 ) return 1;
            else return 0;
     };
};

class smBySalaryFunctor {
public :
     smBySalaryFunctor() {};

     int operator()(Person &p1,Person &p2) {
            if ( p1.salary < p2.salary ) return 1;
            else return 0;
     };
};


class eqByNameFunctor {

private:
     char name[40];
public :
```

```
    eqByNameFunctor(char *n) {
        strcpy(name,n);
    };

    int operator()(Person &p) {
        if ( strcmp(name,p.name) == 0) return 1;
        else return 0;
    };
};

class Persons {
protected:
    vector<Person> items;

private:
    void display();

public:
    Persons() {};

    void input();

    void displayByName();
    void displayBySalary();

    void searchHighestSalary();
    void searchLowestSalary();
    void searchByName(char *n);
};

void Persons::input() {
    Person p;
    char ch;
    cout << "Enter Person information.\n";
    do {
        p.input();
        items.push_back(p);
        cout << "More(Y/N)?" ;
        cin >> ch;
    } while (ch == 'Y');
};

void Persons::display() {
    displayFunctor func;
    for_each(items.begin(),items.end(),func);
};

void Persons::displayByName() {
    smByNameFunctor func;
```

```
    sort(items.begin(),items.end(),func);
    display();
};

void Persons::displayBySalary() {
    smBySalaryFunctor func;

    sort(items.begin(),items.end(),func);
    display();
};

void Persons::searchHighestSalary() {
    smBySalaryFunctor func;
    vector<Person>::iterator it;

    it = max_element(items.begin(),items.end(),func);
    it->display();
};

void Persons::searchLowestSalary() {
    smBySalaryFunctor func;
    vector<Person>::iterator it;

    it = min_element(items.begin(),items.end(),func);
    it->display();
};

void Persons::searchByName(char *n) {
    eqByNameFunctor func(n);
    vector<Person>::iterator it;

    it = find_if(items.begin(),items.end(),func);
    it->display();
};

void main () {
    Persons pl;
    char s[40];
    pl.input();

    cout << "sort by name\n";
    pl.displayByName();
    cout << "sort by salary\n";
    pl.displayBySalary();
    cout << "highest salary\n";
    pl.searchHighestSalary();
```

```
    cout << "lowest salary\n";
    pl.searchLowestSalary();
    cout << "Enter a name:"; cin >> s;
    pl.searchByName(s);
    int i;
    cin >> i;
};
```

1.4. Câu hỏi ôn tập

- Tính đóng gói trong tiếp cận hướng đối tượng thể hiện như thế nào trên C++?
- Kỹ thuật thừa kế đóng vai trò gì trong phát triển ứng dụng?
- Đa hình là gì và được thể hiện thế nào trong C++?
- Khuôn hình khác với kế thừa như thế nào?
- Động cơ nào để xây dựng thư viện khuôn hình chuẩn STL?
- Thư viện khuôn hình chuẩn STL bao gồm những thành phần nào, vai trò của từng thành phần trong một ứng dụng.

Chương 2
BỘ CHỨA

Mục đích chương này

- Làm quen với thư viện STL và bộ chứa
- Giới thiệu cách sử dụng các bộ chứa cơ sở trong STL
- Tìm hiểu một số đặc trưng của các bộ chứa cơ sở trong STL
- Hướng dẫn cách lựa chọn bộ chứa cho chương trình
- Tìm hiểu và xây dựng bộ chứa từ các bộ chứa cơ sở

2.1. Giới thiệu về bộ chứa

Khi xây dựng các chương trình, việc tổ chức lưu trữ dữ liệu là một yếu tố quan trọng. Tổ chức dữ liệu tốt không những cho phép việc truy xuất dữ liệu được thuận tiện, linh hoạt mà còn có thể tối ưu chi phí tính toán của chương trình. Tuy nhiên, việc tổ chức dữ liệu phù hợp là một điều không phải đơn giản. Để đưa ra một cấu trúc dữ liệu thoả mãn các yêu cầu sử dụng cũng như các yêu cầu về không gian và thời gian tính toán đòi hỏi người lập trình có những hiểu biết nhất định cũng như kinh nghiệm lập trình.

Thư viện khuôn hình chuẩn STL (Standard Template Library) đưa ra các lớp bộ chứa với vai trò như các cấu trúc dữ liệu được sử dụng để tổ chức lưu trữ dữ liệu cho chương trình. Nó cho phép lưu trữ một tập các phần tử theo một số hình thức khác nhau như lưu trữ tuần tự, lưu trữ dựa trên giá trị khoá. Sau đây một ví dụ đơn giản để thấy sự thuận tiện khi sử dụng bộ chứa của STL. Giả sử ta cần phải lưu trữ một tập các số nguyên. Theo cách làm quen thuộc, ta khai báo như sau:

```
#define max_num = 100
int int_array[max_num];
```

hoặc

```
int* int_array;
```

Với cách lưu trữ thứ nhất, kích thước mảng bị giới hạn, ngoài ra nó còn gây lãng phí bộ nhớ. Theo cách thứ hai, người dùng phải tự lo việc cấp phát và giải phóng bộ nhớ mỗi khi một phần tử được thêm vào hay lấy ra khỏi mảng. Điều này không hề đơn giản bởi lẽ việc cấp phát hay giải phóng không tốt có thể gây lỗi chương trình hoặc "xả rác" trong bộ nhớ. Khi sử dụng các bộ chứa của STL, người dùng không cần bận tâm tới điều này. Ví dụ, để đạt được mục đích trên ta chỉ cần khai báo

```
vector<int> int_array;
```

và sử dụng các hàm thành phần mà lớp `vector` cung cấp để truy nhập, bổ sung hay xoá đi một phần tử trong mảng:

```
int_array.push_back(1);// Thêm 1 vào vector
int a = int_array[0];//Gán cho a giá trị đầu tiên trong vector
```

Các bộ chứa trong STL được thiết kế theo hướng làm mịn dần. Các bộ chứa càng đơn giản thì càng ít ràng buộc nhưng cũng ít các phương thức làm việc trên nó và ngược lại. Các ràng buộc cũng như các phương thức có hỗ trợ của bộ chứa được đề cập tới dưới dạng các KHÁI NIỆM. Ví dụ, KHÁI NIỆM Forward Container bao hàm ràng buộc chỉ cho phép duyệt bộ chứa theo chiều thuận, cũng tương tự như vậy, KHÁI NIỆM Random Access Container chỉ ra rằng có thể truy xuất giá trị của một phần tử dựa trên chỉ số. Để hiểu một cách thấu đáo về các bộ chứa, việc tìm hiểu các KHÁI NIỆM liên quan tới bộ chứa là cần thiết. Tuy nhiên trong phạm vi quyển sách này, ta sẽ đi ngay vào các lớp bộ chứa cụ thể trong thư viện STL nhằm giúp người đọc quen với bộ chứa mà không quá bận tâm tới các khái niệm trừu tượng. Cách tiếp cận này giúp những người chưa thực sự biết nhiều về lập trình cũng như về thư viện STL cũng có thể sử dụng được thư viện này.

Các lớp bộ chứa là có thể xem là sự cụ thể hoá các KHÁI NIỆM về bộ chứa. Nó là các khuôn hình lớp mà người dùng sẽ sử dụng trực tiếp khi lập trình. Các lớp bộ chứa được phân chia thành hai nhóm chính là các bộ chứa tuần tự (sequence container) và các bộ chứa liên kết (associative container). Các lớp bộ chứa tuần tự bao gồm `vector`, `deque`, `list`, còn các lớp bộ chứa liên kết tiêu biểu bao gồm `set`, `map`, `multiset` và `multimap`. Ngoài ra thư viện STL cũng cung cấp một số cấu trúc khác mà có thể xem là các bộ chứa như lớp `string`, lớp `bit_set`.

2.2. Các lớp bộ chứa tuần tự

2.2.1. Lớp vector

Lớp vector là một bộ chứa được dùng khá phổ biến. Nó được sử dụng để lưu trữ một dãy các phần tử. Lớp vector không chỉ cho phép người dùng truy xuất tới các phần tử dựa trên chỉ số như đối với cấu trúc mảng mà nó còn có khả năng tự động mở rộng khi nhu cầu người dùng vượt quá kích thước tối đa hiện tại. Điều này thuận tiện hơn nhiều khi sử dụng mảng. Khi sử dụng vector, các thao tác cấp phát cũng như giải phóng bộ nhớ trở nên trong suốt với người dùng. Ví dụ sau đây sẽ chỉ ra cách sinh một vector để lưu trữ một dãy các số nguyên từ 0 đến 9

```
#include <vector>
using namespace std;
int main (int argc, char** argv)
{
    vector<int> int_vector;
    for(int i = 0;i < 10;i++)
          int_vector.push_back(i);
}
```

Trong ví dụ này, trước tiên phải khai báo sử dụng tệp vector (lưu ý rằng các tệp chứa tiêu đề của thư viện STL đều không có phần mở rộng), tệp này chứa phần khai báo của lớp vector<>. Đối tượng int_vector là một vector chứa các số nguyên, khai báo của đối tượng này có dạng như dòng 4. Để khai báo một biến vector ta thực hiện theo định dạng sau:

```
vector <T> variable_name;
```

trong đó T là kiểu của phần tử lưu trong vector.

Kích thước của một vector khi khai báo bằng 0. Cần lưu ý rằng vector có hai tham số liên quan tới kích thước là kích thước hiện tại và dung lượng hiện tại. Kích thước hiện tại chỉ ra số phần tử hiện có của vector, dung lượng hiện tại chỉ ra kích thước tối đa của vector. Kích thước vùng nhớ mà biến vector hiện chiếm giữ được tính theo dung lượng hiện tại chứ không theo kích thước hiện tại.

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

```
using namespace std;
int main (int argc, char** argv)
{
    vector<int> int_vector;
    int_vector.reserve(100);
    cout << "Kich thuoc toi da: " << int_vector.capacity() <<
endl;
    cout << "Kich thuoc hien tai: " << int_vector.size() <<
endl;
    for(int i = 0;i < 10;i++)
        int_vector.push_back(i);
    cout << "Kich thuoc toi da: " << int_vector.capacity() <<
endl;
    cout << "Kich thuoc hien tai: " << int_vector.size() <<
endl;
}
```

Kết quả khi thực hiện chương trình như sau:

```
Kich thuoc toi da: 100
Kich thuoc hien tai: 0
Kich thuoc toi da: 100
Kich thuoc hien tai: 10
```

Lưu ý là đoạn chương trình trên sử dụng tệp tiêu đề `iostream` thay cho `iostream.h`, sử dụng tệp `iostream.h` ở đây có thể gây lỗi (xem phần lưu ý cuối chương).

Trong đoạn mã có sử dụng hàm `reserve()` để yêu cầu cấp phát vùng nhớ cho 100 phần tử kiểu nguyên. Đây là cách người dùng tự yêu cầu cấp phát thay vì để `vector` tự quản lý việc cấp phát bộ nhớ của mình. Hàm này thường được dùng khi người dùng biết được số lượng phần tử của bộ chứa. Cách này giúp cho chương trình thực hiện nhanh hơn do không sử dụng các cơ chế cấp phát tự động. Điều này sẽ được bàn kỹ hơn trong phần sau.

Hai hàm `capacity()` và `reserve()` là hai hàm định nghĩa riêng cho lớp `vector`. Hàm `reserve()` khác với cấu tử `vector<>(int)`. Cấu tử này sẽ sinh ra một `vector` có n phần tử với n là giá trị của đối số truyền vào, trong khi `reserve()` chỉ cấp phát vùng nhớ đủ để lưu trữ n phần tử với n là giá trị của đối số. Để thấy rõ sự khác biệt giữa hai hàm này, ta sử dụng ví dụ sau:

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

```
int main(int argc, char** argv)
{
    vector<int> int_vector1, int_vector2(10);
    int_vector1.reserve(10);
    cout << "Vector 1: Kich thuoc toi da " <<
int_vector1.capacity() << ", Kich thuoc hien tai" <<
int_vector1.size() << endl;
    cout << "Vector 2: Kich thuoc toi da " <<
int_vector2.capacity() << ", Kich thuoc hien tai" <<
int_vector2.size() << endl;
    return 0;
}
```

Kết quả chương trình

```
Vector 1: Kich thuoc toi da 10, Kich thuoc hien tai 0;
Vector 2: Kich thuoc toi da 10, Kich thuoc hien tai 10;
```

Để truy xuất một phần tử trong `vector`, người dùng có thể sử dụng chỉ số hoặc bộ duyệt của `vector`. Khi sử dụng chỉ số, người dùng vẫn có thể sử dụng toán tử `[]` như đối với mảng và cũng như mảng, nếu bạn truy nhập với chỉ số lớn hơn dung lượng hiện tại thì sẽ gây lỗi.

```
#include <iostream>

using namespace std;

class student
{
    static int number_student;
    int student_no;
public:
    student():student_no(number_student++){};
    void ShowId(){cout << "Dinh danh cua sinh vien:" <<
student_no endl;};
    ~student(){};
};
#include <vector>
#include "../Student.h"

int main (int argc, char** argv)
{
    vector <student> student_vector;
    for(int i = 0;i < 10;i++)
```

```
    {
            student a_student;
            student_vector.push_back(a_student);
    }
    for(i = 0;i < 10;i++)
            student_vector[i].ShowId();
    return 0;
}
```

Vẫn sẽ có được kết quả y hệt đoạn mã trên nếu vòng lặp `for` cuối cùng được viết lại như sau:

```
for(vector<student>::iterator it = student_vector.begin();it !=
student_vector.end();it++)
            it->ShowId();
```

hoặc:

```
for(vector<student>::iterator it = student_vector.begin();it !=
student_vector.end();it++)
            (*it).ShowId();
```

Mặc dù hai cách sau có vẻ phức tạp hơn, nhưng nó là quy tắc chung để duyệt qua các phần tử trong một bộ chứa. Khi sử dụng với các bộ chứa khác, gần như không phải sửa đoạn mã trên, ngoại trừ việc khai báo lại biến `it` cho hợp kiểu. Ta sẽ thấy rõ được điều này khi xét tới các bộ chứa khác. Hàm `begin()` trả về bộ duyệt trỏ tới phần tử đầu của bộ chứa. Hàm `end()` trỏ tới vị trí sau phần tử cuối cùng, do vậy `*(vector<T>::end())` là không xác định. Chúng ta sẽ nói nhiều hơn về điều này trong chương bàn về các bộ duyệt. Hai hàm này đúng cho tất cả các bộ chứa. Lưu ý rằng, toán tử `[]` của `vector` được đưa ra không nhằm mục đích tương thích ngược với mảng. Đây là toán tử của các lớp triển khai theo KHÁI NIỆM Random Access Container, `vector` là một trong những lớp thuộc số này.

Để truy nhập một phần tử thông qua bộ duyệt trỏ tới nó, ta có hai cách. Cách thứ nhất dùng toán tử giải tham chiếu (`*`) như đối với trường hợp thứ nhất hoặc dùng toán tử `->`. Lúc này, bộ duyệt được xem như một con trỏ thông thường. Các bộ duyệt không chỉ cho phép duyệt qua toàn bộ bộ chứa, nó cũng cho phép duyệt qua một phần nào đó của bộ chứa. Ví dụ sau sẽ in ra ID của 5 sinh viên từ sinh viên thứ 3 đến sinh viên thứ 7.

```
#include <vector>
#include "../Student.h"

int main (int argc, char** argv)
{
vector <student> student_vector;
     for(int i = 0;i < 10;i++)
     {
            student a_student;
            student_vector.push_back(a_student);
     }
     for(vector<student>::iterator it = student_vector.begin() +
3;it != student_vector.begin() + 8;it++)
            it->ShowId();
     return 0;
}
```

Việc liệt kê các số hiệu của sinh viên trong một đoạn nào đó ra màn hình cũng có thể làm được với hàm `copy()`. Điều này không thể được nếu dùng chỉ số, chỉ có thể dùng với bộ duyệt. Ví dụ sau sẽ liệt kê các sinh viên từ thứ ba tới hết danh sách. Lưu ý rằng cần định nghĩa toán tử << cho lớp `student`. Đây không phải là toán tử của lớp `student`, nó là một toán tử toàn cục và được lớp `student` cho phép truy nhập vào các biến thành phần của nó nhờ từ khoá `friend`. Đoạn mã cho lớp sinh viên được bổ sung như sau:

```
#include <iostream>

using namespace std;

class student
{
     static int number_student;
     int student_no;
public:
     student():student_no(number_student++){};
     void ShowId(){cout << "Dinh danh cua sinh vien:" <<
student_no << endl;};
     friend
     ostream& operator<<(ostream& out,const student& a_student)
     {
     return out << "Dinh danh cua sinh vien: " <<
a_student.student_no;
```

```
    };
    ~student(){};
};

#include <vector>
#include <iterator>
#include "../Student.h"
```

```
int main (int argc, char** argv)
{
    vector <student> student_vector;
    for(int i = 0;i < 10;i++)
    {
        student a_student;
        student_vector.push_back(a_student);
    }
    copy(student_vector.begin()+ 3, student_vector.end(),
ostream_iterator <student>(cout,"\n"));
    return 0;
}
```

Chi tiết về hàm `copy()` và `ostream_iterator` sẽ được đề cập trong phần bộ duyệt trên các luồng vào/ra và các giải thuật trong các chương sau.

Lớp `vector` cung cấp khá nhiều hàm thành phần cập nhật các phần tử cho một đối tượng `vector`. Các hàm thành phần này bao gồm thêm một hoặc nhiều phần tử vào một vị trí bất kỳ trong `vector`, xóa đi một phần tử hoặc một phần của `vector`. Tuy nhiên, do những đặc trưng trong tổ chức lưu trữ hai hàm được dùng phổ biến là `push_back()` dùng để thêm một phần tử vào cuối `vector` và `pop_back()` để lấy ra phần tử cuối của `vector`. Chi phí về thời gian cho các hàm này không phụ thuộc vào kích thước của `vector`. Hàm `pop_back()` sẽ xoá bỏ phần tử cuối cùng trong `vector`. Hàm này được thiết kế trả về một giá trị `void` thay vì một tham chiếu hay một giá trị của kiểu phần tử (như suy nghĩ thông thường) nhằm đảm bảo tính đúng đắn và hiệu quả. Nếu trả về một tham chiếu, sau đó phần tử này bị tách ra khỏi `vector` thì chương trình sẽ tạo ra một con trỏ vu vơ, còn nếu trả về một giá trị thì phải gọi cấu tử sao chép nhiều lần. Ví dụ sau sẽ thực hiện việc nhập 10 sinh viên có số hiệu từ 0 đến 9 vào danh sách (`student_vector`), sau đó xoá các sinh viên này theo thứ tự từ dưới lên. Trước khi một sinh viên bị xoá, chương trình sẽ hiện thị số hiệu của sinh viên đó.

```
#include <vector>
#include <iterator>
#include "../Student.h"

int main (int argc, char** argv)
{
    vector <student> student_vector;
    for(int i = 0;i < 10;i++)
            student_vector.push_back(student());

    while (!student_vector.empty())
    {
            cout << "Cac sinh vien bi xoa khoi danh sach: "
            (student_vector.end() - 1)->ShowId();
            student_vector.pop_back();
    }
    return 0;
}
```

Để ý rằng, khi tham chiếu tới phần tử cuối, người ta dùng bộ duyệt `end() - 1` chứ không phải là `end()`. Hàm `end()` sẽ trả về một bộ duyệt có giá trị pass-end. Nó chỉ tới vị trí sau phần tử cuối cùng, do vậy phép giải tham chiếu trên bộ duyệt `end()` là không hợp lệ.

Các hàm `insert()` và `erase()` cho phép thêm vào, xoá đi một hoặc nhiều phần tử tại vị trí bất kỳ nhưng với chi phí thời gian tuyến tính đối với số phần tử của `vector` nên ít được sử dụng. Đối với các ứng dụng đòi hỏi nhiều thao tác chèn vào giữa bộ chứa thì `vector` không được lựa chọn. Thay vào đó người ta chọn `list` hoặc `deque`. Đây không phải là yếu điểm của riêng `vector` mà của tất cả các mô hình xuất phát từ KHÁI NIỆM Back Insert Container. Yếu điểm này của `vector` sẽ được làm rõ hơn trong mục so sánh hiệu quả sử dụng của các bộ chứa tuần tự trong phần sau.

Ví dụ sau minh hoạ việc sử dụng hàm `insert()` và `erase()`. Để thực hiện ví dụ này, lớp `student` sẽ được bổ sung hai hàm `GetId()` và `SetId()`. Hàm thứ nhất trả về số hiệu của sinh viên, hàm thứ hai gán cho sinh viên một số hiệu mới. Ngoài ra, lớp sinh viên cũng được bổ sung thêm một cấu tử với tham số là số hiệu sinh viên, định nghĩa cấu tử này cũng chỉ với mục đích dễ trình bày ví dụ. Cấu tử và các hàm mới bổ sung này được định nghĩa như sau:

```
student(int id){
     student_no = id;
     number_student++;
};
int GetId() const {return student_no;}
void SetId(int id) {student_no = id;};

#include <vector>
#include <iterator>
#include <algorithm>
#include "../Student.h"


int main (int argc, char** argv)
{
     vector <student> student_vector;
     for(int i = 0;i < 10;i++)
            student_vector.push_back(student());
     int pos,id;
     cout << "Vi tri trong danh sach cua sinh vien moi:";
     cin >> pos;
     student_vector.insert(student_vector.begin() + pos - 1);
     copy(student_vector.begin(), student_vector.end(),
ostream_iterator <student>(cout,"\n"));
     cout << "Nhap vao dinh danh cua sinh vien muon xoa: ";
     cin >> id;
     vector<student>::iterator find_res =
find(student_vector.begin(),student_vector.end(),student(id));
     if(find_res == student_vector.end()){
            cout << "Khong co sinh vien co dinh danh " << id << "
trong danh sach" << endl;
            return 0;
     }
     student_vector.erase(find_res);
     cout << "Sinh vien da duoc xoa khoi danh sach" << endl;
     return 0;
}
```

Trong ví dụ trên ta sử dụng khuôn hình giải thuật `find()`, giải thuật này thực hiện việc tìm một phần tử trên một bộ chứa, kết quả được lưu trong bộ duyệt của bộ chứa. Nếu phần tử cần tìm không có trong bộ chứa, bộ duyệt sẽ mang giá trị pass-end. Đây là một giá trị tượng trưng, nó không trỏ tới bất cứ một phần tử nào. Do vậy, nếu thực hiện phép giải tham chiếu trên nó sẽ gây lỗi.

Chi tiết về hàm `find()` xem trong phần các giải thuật ở chương sau.

Tổ chức lưu trữ của vector

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

1. Cấp phát lại một vùng nhớ rộng hơn. Thông thường vùng nhớ mới có kích thước gấp hai lần vùng nhớ hiện tại.
2. Sao chép toàn bộ các phần tử từ vùng nhớ cũ sang vùng nhớ mới nhờ cấu tử sao chép.
3. Xoá bỏ các đối tượng trên vùng nhớ cũ bằng cách gọi các huỷ tử.
4. Giải phóng vùng nhớ cũ.

Do phải thực hiện các thủ tục trên nên việc cấp phát lại của vector rất mất thời gian, đặc biệt là khi phần tử được lưu trữ phức tạp và số lượng phần tử của vector lớn. Chính vì lý do này, vector không được sử dụng trong các ứng dụng đòi hỏi nhiều thao tác chèn hay xoá các phần tử trong bộ chứa. Người dùng có thể tránh được điều này bằng cách sử dụng hàm reserve() để xin cấp phát trước một vùng nhớ nào đó. Tuy nhiên, không phải lúc nào ta cũng biết trước số phần tử sẽ được lưu trong bộ chứa. Để minh hoạ cho điều này ta xét ví dụ trên vector<student>, với lớp student được bổ sung các cấu tử sao chép như sau:

```
#include <iostream>

using namespace std;

class student
{
     static int number_student;
     int student_no;
public:
     student():student_no(number_student++){};
     student(const student& a_std){
            student_no = a_std.student_no;
            cout << "Doi tuong student co ID " << student_no << "
goi cau tu sao chep" << endl;
     };
```

```
    void ShowId(){cout << "Dinh danh cua sinh vien:" <<
student_no << endl;};
    friend ostream& operator<<(ostream& out,const student&
a_student){
            return out << "Dinh danh cua sinh vien: " <<
a_student.student_no;
    };
    ~student(){
            cout << "Doi tuong Student co ID " << student_no << "
goi ham huy tu" << endl;
    };
};
```

```
#include <vector>
#include "../Student.h"

int main(int argc, char* argv[])
{
    vector<student> students;
    for (int i = 0;i < 3;i++)
    {
       cout << "Lan goi push_back thu " << i + 1 << endl;
       students.push_back(student());
    }
    cout << "Ket thuc chuong trinh" << endl;
    return 0;
}
```

Kết quả của chương trình là

```
Lan goi push_back thu 1
Doi tuong student co ID 0 goi cau tu sao chep
Doi tuong student co ID 0 goi huy tu
Lan goi push_back thu 2
Doi tuong student co ID 0 goi cau tu sao chep
Doi tuong student co ID 1 goi cau tu sao chep
Doi tuong student co ID 0 goi huy tu
Doi tuong student co ID 1 goi huy tu
Lan goi push_back thu 3
Doi tuong student co ID 0 goi cau tu sao chep
Doi tuong student co ID 1 goi cau tu sao chep
Doi tuong student co ID 2 goi cau tu sao chep
Doi tuong student co ID 0 goi huy tu
Doi tuong student co ID 1 goi huy tu
Doi tuong student co ID 2 goi huy tu
Ket thuc chuong trinh
```

```
Doi tuong student co ID 0 goi huy tu
Doi tuong student co ID 1 goi huy tu
Doi tuong student co ID 2 goi huy tu
```

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

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

using namespace std;

int main(int argc,char* argv[]) {
  vector<int> vi(10, 0);
  ostream_iterator<int> out(cout, " ");
  copy(vi.begin(), vi.end(), out);
  vector<int>::iterator i = vi.begin();
  cout << "\n i: " << long(i) << endl;
  *i = 47;
  copy(vi.begin(), vi.end(), out);
  // Bắt chương trình phải cấp phát lại bộ nhớ
  vi.resize(vi.capacity() + 1);
  cout << "\n i: " << long(i) << endl;
  cout << "vi.begin(): " << long(vi.begin()) << endl;
  *i = 48;  // Gây lỗi.
  cout << "i: " << long(i) << endl;
  return 0;
}
```

Chương trình trên gây lỗi khi thực hiện dịch và chạy chương trình trên môi trường Visual C++ 6.0 .

2.2.2. Lớp deque

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

Cũng giống như các lớp được triển khai dựa trên KHÁI NIỆM Random Access Container, deque cho phép người dùng truy cập ngẫu nhiên với toán tử [] hoặc hàm at().

```
#include <deque>
#include "../Student.h"

int main(int argc, char* argv[])
{
    deque<student> student_deque;
    for(int i = 0;i < 3;i++)
            student_deque.push_back(student());
    for(i = 0;i < 3;i++)
            student_deque.push_front(student());
    cout << "Danh sach sinh vien" << endl;
    for(i = 0;i < student_deque.size();i++)
    {
            cout << "Sinh vien thu " << i + 1 << endl;
            cout << "      " << student_deque[i] << endl;
    }
    return 0;
}
```

Kết quả chương trình :

```
Danh sach sinh vien
Sinh vien thu 1:
     Dinh danh cua sinh vien: 5
Sinh vien thu 2:
     Dinh danh cua sinh vien: 4
Sinh vien thu 3
     Dinh danh cua sinh vien: 3
Sinh vien thu 4:
     Dinh danh cua sinh vien: 0
Sinh vien thu 5:
     Dinh danh cua sinh vien: 1
```

```
Sinh vien thu 6:
     Dinh danh cua sinh vien: 2
```

(Trong phần kết quả trên, ta đã lược đi phần hiển thị các thông báo về gọi cấu tử sao chép và huỷ tử nên kết quả hiển thị thực tế trên màn hình có thể khác)

Vẫn có được kết quả trên nếu thay đoạn mã hiển thị danh sách sinh viên bằng các đoạn mã sau:

```
     cout << "Danh sach sinh vien" << endl;
     for(i = 0;i < student_deque.size();i++)
     {
            cout << "Sinh vien thu " << i+1 << ":" << endl;
            cout << "      " << student_deque.at(i) << endl;
     }
```

hoặc

```
     cout << "Danh sach sinh vien" << endl;
     deque<student>::iterator it;
     for(it = student_deque.begin();i != student_deuqe.end()
;it++)
     {
            cout << "Sinh vien thu " << i + 1 << ":" << endl;
            cout << "      " << *it << endl;
     }
```

Các hàm thành phần của `deque` đều được triển khai từ các KHÁI NIỆM được mô hình hoá. Chính vì vậy, so với `vector`, `deque` có thêm các hàm thuộc KHÁI NIỆM Front Insert Container), các hàm này bao gồm `front()`, `push_front()`, `pop_front()`. Ngoài ra, `deque` cũng không có hai hàm `capacity()` và `reverse()`, hai hàm này được thiết kế riêng cho `vector`. Rõ ràng là sử dụng lớp này để mô tả cấu trúc hàng đợi hiệu quả hơn nhiều so với khi dùng `vector`. Chính vì thế bộ chứa `queue` được xây dựng mặc định trên `deque` (xem thêm phần các bộ chứa thích nghi). Ví dụ sau mô tả hoạt động của một quầy giao dịch

```
#include <deque>
#include <iostream>
#include <ctime>

using namespace std;
```

```
class Customer
{
    int transaction_time;
public:
    Customer(){
        srand(time(0));
        transaction_time = rand() % 100;
    };
    Customer(int trans_time):transaction_time(trans_time % 100){

    };
    int GetTransactionTime(){return transaction_time;};
    ~Customer(){};
};

class Server
{
    deque<Customer> customers;
    bool isReady;
public:


Server():isReady(false){
        customers.clear();
    };
    void Serve(){
        while(isReady && !customers.empty())
        {5
            cout << "Thoi gian cua giao dich " <<
(customers.front()).GetTransactionTime() << endl;
            customers.pop_front();
        }
    };
    void BeginServe(){
        isReady = true;
        Serve();
    };
    void TerminateServe(){
        isReady = false;
    }
    void EndServe(){
        isReady = false;
        customers.clear();
    };
    void Accept(const Customer& a_customer){
        customers.push_back(a_customer);
```

```
    }

};

int main(int argc, char* argv[])
{
    Server server;
    srand(time(0));
    for(int i = 0;i < (rand() % 25);i++)
    {
        server.Accept(Customer(rand()));
    }
    server.BeginServe();
    return 0;
}
```

Ví dụ trên mô tả hoạt động hai lớp đối tượng là khách hàng (Customer) và người phục vụ (Server). Mỗi khách hàng được đặc trưng bởi thời gian yêu cầu phục vụ. Người phục vụ sẽ làm việc theo nguyên tắc: người đến trước được phục vụ trước, đồng thời tại mỗi thời điểm chỉ phục vụ không quá một người và khách hàng được phục vụ liên tục trong suốt thời gian yêu cầu. Người phục vụ có một danh sách lưu các yêu cầu phục vụ và một số tác vụ như chấp nhận yêu cầu - `Accept ()`, phục vụ - `Serve()`, bắt đầu hoặc ngừng phiên làm việc `BeginServe()`, `EndServe()`. Hai thao tác làm việc với hàng đợi các yêu cầu là `Accept()` và `Serve()`. `Accept()` xếp một yêu cầu vào cuối hàng đợi, còn `Serve()` lấy ra một yêu cầu ở đầu hàng đợi. Chương trình chính khởi tạo một người phục vụ và gửi tới nó một số yêu cầu nào đó với thời gian yêu cầu phục vụ là ngẫu nhiên.

Mặc dù `deque` cũng cho phép truy nhập ngẫu nhiên với toán tử `[]` nhưng thời gian để thực hiện thao tác này thường chậm hơn (xem ví dụ minh hoạ trong mục so sánh hiệu quả sử dụng giữa các bộ chứa tuần tự). Nguyên nhân của điều này là do tổ chức lưu trữ của `vector` khác với `deque`. Lớp `vector` lưu trữ toàn bộ các phần tử trong một vùng bộ nhớ duy nhất, còn `deque` lưu trữ các phần tử rải rác trên một hoặc nhiều vùng nhớ. Vị trí và trật tự các vùng nhớ này được lưu trữ trong bản đồ cấu trúc. Với tổ chức như vậy, `deque` không yêu cầu cấp phát lại khi số các phần tử được thêm vào vượt quá dung lượng hiện tại. Do vậy, nó cũng không phải thực hiện các thao tác sao chép và xoá đi các phần tử hiện tại như đối với `vector`. Đây cũng chính là lý do tại sao `deque` không cần tới hai hàm `capacity()` và `reverse()`. Ví dụ sau sẽ minh hoạ cho sự khác biệt này.

```
#include <deque>
#include "../Student.h"

int main(int argc, char* argv[])
{
    deque<student> student_deque;
    for(int i = 0;i < 3;i++)
    {
            cout << "Lan goi push_back thu " << i+1 << endl;
            student_deque.push_back(student());
    }
    cout << "Ket thuc chuong trinh" << endl;
    return 0;
}
```

Kết quả chương trình

```
Lan goi push_back thu 1
Doi tuong student co ID 0 goi cau tu sao chep
Doi tuong student co ID 0 goi huy tu
Lan goi push_back thu 2
Doi tuong student co ID 1 goi cau tu sao chep
Doi tuong student co ID 1 goi huy tu
Lan goi push_back thu 3
Doi tuong student co ID 2 goi cau tu sao chep
Doi tuong student co ID 2 goi huy tu
Chuong trinh ket thuc
Doi tuong student co ID 0 goi huy tu
Doi tuong student co ID 1 goi huy tu
Doi tuong student co ID 2 goi huy tu
```

Rõ ràng, so với `vector`, số lần gọi tới các cấu tử sao chép và huỷ tử của phần tử ít hơn nhiều do không phải gọi các cấu tử sao chép và huỷ tử của các phần tử đã được thêm vào deuqe.

Do `deque` không yêu cầu di chuyển vùng nhớ (chỉ có cấp phát thêm không có cấp phát lại) nên không có hiện tượng bị lỗi về truy nhập như đối với `vector`. Tuy nhiên, có hai điều nên tránh khi làm việc với `deque`. Thứ nhất là chèn thêm một phần tử vào giữa bộ chứa. Mặc dù điều này hoàn toàn hợp lệ, nhưng do `deque` không được thiết kế tối ưu cho công việc này nên không đảm bảo cho một kết quả tốt nhất. Điều thứ hai là không nên gọi hàm `insert()` lặp đi lặp lại với cùng một bộ duyệt. Nó làm cho ta không kiểm soát được các bộ duyệt này. Ví dụ đoạn chương trình sau có thể gây ra lỗi.

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

int main(int argc,char* argv[]) {
  deque<int> di(100, 0);
  copy(di.begin(), di.end(),
  ostream_iterator<int>(cout, " "));
  deque<int>::iterator i = di.begin() + di.size() / 2;
  for(int j = 0; j < 1000; j++)
  {
    cout << j << endl;
    di.insert(i++, 1);
  }
  return 0;
}
```

Chương trình trên khi chạy trên Windows 2000 với Visual C++ 6.0 thường treo khi j = 412.

2.2.3. Lớp list

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

`list` không được dùng cho các ứng dụng đòi hỏi các yêu cầu truy nhập ngẫu nhiên. Việc truy nhập các phần tử trên `list` được các ứng dụng thực hiện theo cách duyệt từ đầu đến cuối danh sách hoặc theo chiều ngược lại. Mặc dù thời gian để duyệt qua toàn bộ `list` lâu hơn nhiều so với `vector` hay `deque` có cùng số phần tử, nhưng nó cũng không gây ra hiện tượng 'thắt nút cổ chai' khi thực hiện chương trình nếu như bạn không thực hiện thủ tục này quá nhiều trong chương trình. Việc sử dụng `list` như một bộ chứa tuần tự không có gì khác so với `vector` hay `deque`. Do vậy, ta sẽ không bàn thêm gì về phần này mà chỉ đi sâu vào một số sự khác biệt của `list` với các lớp bộ chứa tuần tự khác.

Các thủ tục hoán đổi (`swap()`) hay đảo ngược vị trí (`reverse()`) trên `list` không yêu cầu việc sao chép các đối tượng như đối với `vector` hay `deque`. Thay vào đó, các liên kết được thay đổi như mong muốn. Tuy nhiên, điều này chỉ đúng với các hàm thành phần của lớp `list`, đối với các hàm khái lược như `swap()`, `reverse()` thì nó vẫn thực hiện việc sao chép.

Điều này hoàn toàn dễ hiểu bởi lẽ các hàm thành phần của lớp `list` được thiết kế dựa trên những đặc trưng của `list` nên nó chỉ thực hiện việc thay đổi liên kết giữa các phần tử mà không thực hiện việc sao chép các đối tượng. Trong khi đó, các hàm khái lược không quan tâm tới đặc trưng này. Trên thực tế, không riêng gì `list` mà tất cả các bộ chứa đều cụ thể hoá hàm `swap()` để tối ưu thời gian thực hiện. Ta sẽ nói rõ về các hàm hoán đổi trên các bộ chứa trong phần cuối chương như một phần tìm hiểu sâu hơn về bộ chứa. Ví dụ sau đây minh hoạ sự khác biệt giữa hàm thành phần `reverse()` của `list` và hàm `reverse()` khái lược. Trong ví dụ, mã của các cấu tử và huỷ tử của lớp `student` được sửa lại để có thể lưu lại số lần gọi trong các biến `static`. Phần khai báo lớp `student` được sửa lại như sau:

```
#include <iostream>

using namespace std;

class student
{
    int student_no;
public:
    static int number_student;
    static int number_empty_constructor_call;
    static int number_copy_constructor_call;
    static int number_destructor_call;

public:
    student():student_no(number_student++){
        number_empty_constructor_call++;
    };
    student(const student& a_std){
        student_no = a_std.student_no;
        number_copy_constructor_call++;
    };
    void ShowId(){cout << "Dinh danh cua sinh vien:" <<
student_no << endl;};
    friend ostream& operator<<(ostream& out,const student&
a_student){
        return out << "Dinh danh cua sinh vien: " <<
a_student.student_no;
    };
    ~student(){
        number_destructor_call++;
    };
};
```

```
#include <list>
#include <algorithm>
#include "../Student.h"

int student::number_student = 0;
int student::number_copy_constructor_call = 0;
int student::number_destructor_call = 0;
int student::number_empty_constructor_call = 0;

int main(int argc, char* argv[])
{
    student student_array[5] =
{student(),student(),student(),student(),student()};
    int student_array_size =
sizeof(student_array)/sizeof(*student_array);
    list<student> student_list(student_array,student_array +
student_array_size);
    cout << "Truoc khi goi ham reverse" << endl;
    cout << "So lan goi cau tu sao chep: " <<
student::number_copy_constructor_call << endl;
    cout << "So lan goi huy tu: " <<
student::number_destructor_call << endl;
    student_list.reverse();
    cout << "Sau khi goi ham reverse thanh phan" << endl;
    cout << "So lan goi cau tu sao chep: " <<
student::number_copy_constructor_call << endl;
    cout << "So lan goi huy tu: " <<
student::number_destructor_call << endl;
    reverse(student_list.begin(),student_list.end());
    cout << "Sau khi goi ham reverse khai luoc" << endl;
    cout << "So lan goi cau tu sao chep: " <<
student::number_copy_constructor_call << endl;
    cout << "So lan goi huy tu: " <<
student::number_destructor_call << endl;

    return 0;
}
```

Kết quả của chương trình :

```
Truoc khi goi ham reverse
So lan goi cau tu sao chep: 5
So lan goi huy tu: 0
Sau khi goi ham reverse thanh phan
So lan goi cau tu sao chep: 5
So lan goi huy tu: 0
```

```
Sau khi goi ham reverse khai luoc
So lan goi cau tu sao chep: 7
So lan goi huy tu: 2
```

Lưu ý rằng, hàm `sort()` khái lược không làm việc trên `list` do nó chỉ hỗ trợ các bộ chứa cho phép truy nhập ngẫu nhiên, nhưng `list` cũng có hàm `sort()` của riêng mình. Và cũng giống như các hàm `reverse()` hay `swap()`, hàm `sort()` chỉ thực hiện việc thay đổi liên kết nên không gọi tới các cấu tử sao chép hay huỷ tử. Để thực hiện việc sắp xếp trên danh sách các `student`, lớp `student` phải định nghĩa toán tử >. Định nghĩa của toán tử này được mô tả trong đoạn mã dưới và được thêm vào phần khai báo của lớp `student` trong tệp `Student.h`

```
bool operator>(const student& other_student)const{
            return student_no < other_student.student_no;
```

hoặc

```
friend bool operator>(const student& first_student,const student&
second_student)
{
return first_student.student_no < second_student.student_no;
}
```

Đoạn chương trình sau sẽ sắp xếp sinh viên theo số hiệu và in ra màn hình

```
#include <list>
#include <algorithm>
#include "../Student.h"

int main(int argc, char* argv[])
{
     student student_array[5] =
{student(),student(),student(),student(),student()};
     int student_array_size =
sizeof(student_array)/sizeof(*student_array);
      list<student> student_list(student_array,student_array +
student_array_size);
     greater<student> student_comp;
     student_list.sort(student_comp);
     copy(student_list.begin(),student_list.end(),ostream_iterato
r<student>(cout,"\n"));
     return 0;
}
```

Ngoài hai hàm `reverse()` và `sort()` được cụ thể hoá cho phù hợp với các đặc trưng riêng, `list` còn có một số hàm đặc biệt hay được dùng đó là `merge()`, `unique()`, `reverse()`, `splice()`, `remove()` và `remove_if()`. Các hàm `merge()` và `splice()` đều thực hiện việc trộn hai danh sách. Nói đúng hơn là chuyển các phần tử từ danh sách này sang danh sách khác, vì sau khi thực hiện phép trộn, một trong danh sách ban đầu trở thành rỗng. Hàm `merge()` thực hiện trộn hai danh sách theo quy tắc sau: tại mỗi phần tử thuộc danh sách thứ nhất chèn vào phía trước nó các phần của danh sách thứ hai sao cho các phần tử được chèn vào "lớn hơn" nó. Tính "lớn hơn" được xác định dựa trên định nghĩa toán tử < trong lớp phần tử, ta để dấu lớn hơn trong dấu nháy vì có thể tự quy định thế nào là lớn hơn. Hàm `splice()` thực hiện việc trộn danh sách theo cách đơn giản hơn. Người dùng chỉ ra vị trí trên danh sách thứ nhất mà tại đó danh sách thứ hai được nối vào, các phần tử từ sau vị trị này trên danh sách thứ nhất sẽ được nối vào cuối danh sách thứ hai.

```
#include <list>
#include "../Student.h"

int main(int argc, char* argv[])
{
    student student_array[5] =
{student(1),student(5),student(3),student(9),student(7)};
    int student_array_size =
sizeof(student_array)/sizeof(*student_array);
    list<student> student_list1(student_array,student_array +
student_array_size);
    list<student> student_list2(student_array,student_array +
student_array_size);
    list<student> student_list3,student_list4;
    for(int i = 0;i < 5;i++)
    {
        student_list3.push_back(student(2*i));
        student_list4.push_back(student(2*i));
    };
    greater<student> student_comp;

    student_list1.splice(student_list1.begin(),student_list3);
    student_list2.merge(student_list4,student_comp);
    cout << "Ket qua cua ham Splice" << endl;
    copy(student_list1.begin(),student_list1.end(),ostream_itera
tor<student>(cout,"\n"));
```

```
    cout << "Ket qua cua ham Merge" << endl;
    copy(student_list2.begin(),student_list2.end(),ostream_itera
tor<student>(cout,"\n"));
    return 0;
}
```

Kết quả của đoạn chương trình trên:

```
Ket qua cua ham Splice
Dinh danh cua sinh vien: 0
Dinh danh cua sinh vien: 2
Dinh danh cua sinh vien: 4
Dinh danh cua sinh vien: 6
Dinh danh cua sinh vien: 8
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 5
Dinh danh cua sinh vien: 3
Dinh danh cua sinh vien: 9
Dinh danh cua sinh vien: 7
Ket qua cua ham Merge
Dinh danh cua sinh vien: 0
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 2
Dinh danh cua sinh vien: 4
Dinh danh cua sinh vien: 5
Dinh danh cua sinh vien: 3
Dinh danh cua sinh vien: 6
Dinh danh cua sinh vien: 8
Dinh danh cua sinh vien: 9
Dinh danh cua sinh vien: 7
```

Hàm `remove()` và `remove_if()` loại bỏ một hoặc nhiều phần tử trong `list` theo một tiêu chuẩn nào đó. Hàm `remove()` sẽ xoá tất cả các phần tử nào bằng với đối số của hàm. Hàm `remove_if()` nhận đối số là một biểu thức logic vị từ, các phần tử mà với nó biểu thức logic nhận giá trị đúng sẽ bị loại khỏi danh sách. Để thực hiện được hàm `remove()`, lớp phần tử của `list` phải định nghĩa toán tử ==. Trong ví dụ này ta định nghĩa hai đối tượng `student` là bằng nhau nếu `student_no` của chúng bằng nhau. Phần định nghĩa toán tử == sau sẽ được thêm vào lớp `student`

```
bool operator==(const student& a_student){
    return student_no == a_student.student_no;
};
```

```
int main(int argc, char* argv[])
{
    student student_array[5] =
{student(1),student(5),student(3),student(9),student(7)};
    int student_array_size =
sizeof(student_array)/sizeof(*student_array);
    list<student> student_list(student_array,student_array +
student_array_size);
    cout << "Danh sach ban dau:" << endl;
    copy(student_list.begin(),student_list.end(),ostream_iterato
r<student>(cout,"\n"));
    student_list.remove(student(5));
    cout << "Sau khi xoa bo:" << endl;
    copy(student_list.begin(),student_list.end(),ostream_iterato
r<student>(cout,"\n"));
    return 0;
}
```

Kết quả của chương trình

```
Danh sach ban dau:
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 5
Dinh danh cua sinh vien: 3
Dinh danh cua sinh vien: 9
Dinh danh cua sinh vien: 7
Sau khi xoa bo:
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 3
Dinh danh cua sinh vien: 9
Dinh danh cua sinh vien: 7
```

Hàm `remove_if()` là một khuôn hình hàm thành phần nên không phải trình dịch nào cũng hỗ trợ. Đối với các trình dịch hỗ trợ khuôn hình hàm thành phần thì đối số của hàm này là một hàm vị từ bất kỳ. Trong trường hợp trình dịch của Visual C++, đối số của hàm này chỉ có thể là `binder2nd<not_equal_to<_Ty> >`, nghĩa là nó chỉ cho phép xoá tất cả các phần tử không bằng một phần tử nào đó. Ví dụ sau sẽ minh hoạ sử dụng hàm này trên môi trường Visual C++ Hàm `remove_if()` sẽ xoá bỏ tất cả các sinh viên trong danh sách có số hiệu khác 5. Để thực hiện hàm này, lớp `student` cần bổ sung toán tử so sánh `!=`.

```
bool operator!=(const student& other_student)const{
    return student_no != other_student.student_no;
```

```
}
#include <list>
#include <algorithm>
#include "../Student.h"

int main(int argc, char* argv[])
{
    student student_array[5] =
{student(1),student(5),student(3),student(9),student(7)};
    int student_array_size =
sizeof(student_array)/sizeof(*student_array);
    list<student> student_list(student_array,student_array +
student_array_size);
    binder2nd<not_equal_to<student> >
student_diff(not_equal_to<student>(),student(5));
    cout << "Danh sach ban dau:" << endl;
    copy(student_list.begin(),student_list.end(),ostream_iterato
r<student>(cout,"\n"));
    student_list.remove_if(student_diff);
    cout << "Sau khi xoa bo:" << endl;
    copy(student_list.begin(),student_list.end(),ostream_iterato
r<student>(cout,"\n"));
    return 0;
}
```

Chạy đoạn chương trình trên ta thu được kết quả sau:

```
Danh sach ban dau:
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 5
Dinh danh cua sinh vien: 3
Dinh danh cua sinh vien: 9
Dinh danh cua sinh vien: 7
Sau khi xoa bo:
Dinh danh cua sinh vien: 5
```

Trong ví dụ trên, ta có sử dụng bộ thích nghi trên đối tượng hàm `binder2nd()`. Xét một cách đơn giản, đối tượng hàm này thực hiện chuyển một đối tượng hàm hai đối số về đối tượng hàm một đối số. Cấu tử của đối tượng hàm này gồm một đối tượng hàm hai đối số (trong ví dụ là `not_equal_to<student>`) và giá trị của đối số thứ hai (trong ví dụ là `student(5)` – sinh viên có số hiệu là 5). Chi tiết về đối tượng hàm và các bộ thích nghi trên đối tượng hàm sẽ được trình bày trong các chương sau.

Hàm `unique()` thực hiện việc loại bỏ các phần tử tương đương trong `list`. Mỗi nhóm các phần tử tương đương chỉ giữ lại một phần tử. Quan hệ

tương đương giữa hai phần tử trong danh sách được xác định qua vị từ hai biến là đối số của hàm. Trong trường hợp không chỉ ra đối số này, (hàm `unique()` không được truyền tham số) quan hệ tương đương được hiểu là quan hệ ngang bằng, quan hệ này được xác định qua toán tử ==. Lưu ý rằng, đối tượng hàm truyền vào cho `unique()` là bất kỳ nếu trình dịch hỗ trợ khái niệm khuôn hình hàm thành phần. Đối với Visual C++, vị từ này được chỉ định là `not_equal_to<_Ty>`. Do vậy, hàm `unique()` có đối số khi chạy với Visual C ++ sẽ trả về một danh sách gồm toàn các phần tử trùng với phần tử đầu tiên của danh sách. Trong trường hợp các phần tử của danh sách ban đầu đều khác nhau (xét theo toán tử ==), hàm trả về một danh sách có duy nhất một phần tử là phần tử đầu. Ví dụ:

```
nt main(int argc, char* argv[])
{
    student student_array[5] =
{student(1),student(1),student(5),student(5),student(9)};
    int student_array_size =
sizeof(student_array)/sizeof(*student_array);
    list<student> student_list(student_array,student_array +
student_array_size);
    cout << "Danh sach ban dau:" << endl;
    copy(student_list.begin(),student_list.end(),ostream_iterato
r<student>(cout,"\n"));
    student_list.unique(not_equal_to<student>());
    cout << "Sau khi xoa bo:" << endl;
    copy(student_list.begin(),student_list.end(),ostream_iterato
r<student>(cout,"\n"));
    return 0;
}
```

Kết quả thu được :

```
Danh sach ban dau:
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 5
Dinh danh cua sinh vien: 5
Dinh danh cua sinh vien: 9
Sau khi thuc hien ham unique:
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 1
```

Nếu

```
student_list.unique(not_equal_to<student>());
```

được thay bởi

```
student_list.unique();
```

thì ta thu được kết quả sau:

```
Danh sach ban dau:
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 5
Dinh danh cua sinh vien: 5
Dinh danh cua sinh vien: 9
Sau khi thuc hien ham unique:
Dinh danh cua sinh vien: 1
Dinh danh cua sinh vien: 5
Dinh danh cua sinh vien: 9
```

2.2.4. Tìm hiểu sâu hơn về các bộ chứa tuần tự

Trong phần trên, chúng tôi đã giới thiệu cách sử dụng các bộ chứa tuần tự và một số khía cạnh đặc thù của từng lớp bộ chứa. Với những kiến thức cơ bản đã nêu, chúng ta đã có thể sử dụng các bộ chứa trong chương trình của mình. Tuy nhiên, để thực sự làm chủ được các bộ chứa và sử dụng chúng một cách linh hoạt, cần hiểu thêm một số khía cạnh bên trong của các bộ chứa tuần tự do STL cung cấp. Trong phần này, ta sẽ đề cập tới hai khía cạnh là các yếu tố cần quan tâm khi lựa chọn bộ chứa và sao chép nội dung giữa hai bộ chứa khác kiểu. Đây là hai vấn đề mà hầu hết người sử dụng đều quan tâm. Qua phần trình bày về hai vấn đề này, mong bạn đọc có thể rút ra những nguyên lý sử dụng phù hợp cho riêng mình.

Lựa chọn bộ chứa cho chương trình

Xét từ góc độ người dùng, các bộ chứa tuần tự đều lưu trữ các phần tử theo trật tự tuyến tính. Tuy nhiên, tổ chức lưu trữ thật sự của từng loại bộ chứa rất khác nhau. Sự khác nhau này nhằm tối ưu hoá hiệu quả sử dụng của các bộ chứa cho từng mục đích cụ thể. Để lựa chọn một bộ chứa cho chương trình của mình, ta cần xem xét trên các khía cạnh là mục đích sử dụng và hiệu quả sử dụng. Dựa trên mục đích sử dụng, chọn ra bộ chứa hỗ trợ tối đa các hàm thành phần chương trình yêu cầu. Tiếp sau, sẽ xét tới chi phí về thời gian tính toán trên bộ chứa mình sử dụng. Sau đây là một ví dụ so sánh chi phí tính toán về mặt thời gian giữa các bộ chứa tuần tự, để qua đó giúp bạn đọc có thể có một lựa chọn tốt cho chương trình của mình.

Ví dụ thực hiện so sánh hiệu năng của các bộ chứa ứng với các thao tác trên nó. Các thao tác này bao gồm: bổ sung một phần tử vào đầu, vào cuối và vào giữa bộ chứa; truy nhập ngẫu nhiên; duyệt qua bộ chứa; hoán đổi hai bộ chứa (cùng kiểu); xoá một phần tử ở cuối hoặc ở giữa bộ chứa. Mỗi thao tác được viết dưới dạng một đối tượng hàm. Khuôn hình của các đối tượng hàm có một tham số là loại bộ chứa cho thực hiện thao tác này. Trong phần định nghĩa toán tử `()` của các đối tượng hàm, ta không có phần kiểm tra loại bộ chứa. Do vậy, nếu thực hiện các thao tác mà bộ chứa không hỗ trợ có thể gây lỗi khi dịch chương trình. Ví dụ, gọi `InsertFront()` với đối số là một đối tượng `vector` sẽ gây lỗi vì lớp `vector` không hỗ trợ `push_front()`. Ngoài các đối tượng hàm, chương trình sử dụng hàm `MeasureTime()` để tính thời gian thực hiện một thao tác trên một bộ chứa. Khuôn hình của hàm `MeasureTime()` cần hai tham số là thao tác cần thực hiện và lớp bộ chứa.

Số phần tử mặc định cho các bộ chứa là 1000. Người dùng có thể thay đổi giá trị này bằng cách ấn định giá trị cho đối số thứ nhất của chương trình.

```
#include <vector>
#include <queue>
#include <list>
#include <iostream>
#include <string>
#include <typeinfo>
#include <ctime>
#include <cstdlib>

using namespace std;

class FixedSize {
  int x[20];
} fs;

template<class Cont>
struct InsertBack {
  void operator()(Cont& c, long count) {
    for(long i = 0; i < count; i++)
      c.push_back(fs);
  }
  char* GetName() { return "Thao tac bo sung vao cuoi bo chua"; }
};

template<class Cont>
```

```
struct InsertFront {
  void operator()(Cont& c, long count) {
    long cnt = count * 10;
    for(long i = 0; i < cnt; i++)
      c.push_front(fs);
  }
  char* GetName() { return "Thao tac bo sung vao dau bo chua"; }
};

template<class Cont>
struct InsertMiddle {
  void operator()(Cont& c, long count) {
    typename Cont::iterator it;
    long cnt = count / 10;
    for(long i = 0; i < cnt; i++) {
      it = c.begin();
      it++;
      c.insert(it, fs);
    }
  }
  char* GetName() { return "Thao tac bo sung vao giua bo chua ";
}
};

template<class Cont>
struct RandomAccess { // không dùng cho list
  void operator()(Cont& c, long count) {
    int sz = c.size();
    long cnt = count * 100;
    for(long i = 0; i < cnt; i++)
      c[rand() % sz];
  }
  char* GetName() { return "Thao tac truy nhap ngau  nhien tren
bo chua"; }
};

template<class Cont>
struct Traversal {
  void operator()(Cont& c, long count) {
    long cnt = count / 100;
    for(long i = 0; i < cnt; i++) {
      typename Cont::iterator it = c.begin(),
        end = c.end();
      while(it != end) it++;
    }
```

```
  }
  char* GetName() { return "Thao tac duyet bo chua"; }
};

template<class Cont>
struct Swap {
  void operator()(Cont& c, long count) {
    int middle = c.size() / 2;
    typename Cont::iterator it = c.begin(),
      mid = c.begin();
    it++;
    // Để bộ duyệt trỏ tới giữa bộ chứa
    for(int x = 0; x < middle + 1; x++)
      mid++;
    long cnt = count * 10;
    for(long i = 0; i < cnt; i++)
      swap(*it, *mid);
  }
  char* GetName() { return "Swap"; }
};

template<class Cont>
struct RemoveMiddle {
  void operator()(Cont& c, long count) {
    long cnt = count / 10;
    if(cnt > c.size()) {
      cout << "Thao tac xoa o giua: not enough elements"
        << endl;
      return;
    }
    for(long i = 0; i < cnt; i++) {
      typename Cont::iterator it = c.begin();
      it++;
      c.erase(it);
    }
  }
  char* GetName() { return "Thao tac xoa o giua"; }
};

template<class Cont>
struct RemoveBack {
  void operator()(Cont& c, long count) {
    long cnt = count * 10;
    if(cnt > c.size()) {
      cout << "Thao tac xoa o cuoi: not enough elements"
```

```
        << endl;
      return;
    }
    for(long i = 0; i < cnt; i++)
      c.pop_back();
    return;
  }
  char* GetName() { return "Thao tac xoa o cuoi"; }
};

template<class Op, class Container>
void measureTime(Op f, Container& c, long count){
  string id(typeid(f).name());
  bool Deque = id.find("deque") != string::npos;
  bool List = id.find("list") != string::npos;
  bool Vector = id.find("vector") !=string::npos;
  string cont_type = Deque ? "deque" : List ? "list"
    : Vector? "vector" : "unknown";
  cout << f.GetName() << " tren " << cont_type << ": ";
clock_t ticks = clock();
        f(c, count);
  ticks = clock() - ticks;
  cout << ticks << endl;
}

typedef deque<FixedSize> DF;
typedef list<FixedSize> LF;
typedef vector<FixedSize> VF;

srand(time(0));
  long count = 1000;
  if(argc >= 2) count = atoi(argv[1]);
  DF deq;
  LF lst;
  VF vec, vecres;
  vecres.reserve(count); // Preallocate storage
  cout << "So sanh doi voi thao tac bo sung vao cuoi bo chua:" <<
endl;
  measureTime(InsertBack<VF>(), vec, count);
  measureTime(InsertBack<VF>(), vecres, count);
  measureTime(InsertBack<DF>(), deq, count);
  measureTime(InsertBack<LF>(), lst, count);
  cout << "\nSo sanh doi voi thao tac bo sung vao dau  bo chua
(khong thuc hien tren vector):" << endl;
```

```
  measureTime(InsertFront<DF>(), deq, count);
  measureTime(InsertFront<LF>(), lst, count);
  cout << "\nSo sanh doi voi thao tac bo sung o giua bo chua :"
<< endl;
  measureTime(InsertMiddle<VF>(), vec, count);
  measureTime(InsertMiddle<DF>(), deq, count);
  measureTime(InsertMiddle<LF>(), lst, count);
  cout << "\nSo sanh doi voi thao tac nhap ngau nhien tren bo
chua (khong thuc hien tren list):" << endl;
  measureTime(RandomAccess<VF>(), vec, count);
  measureTime(RandomAccess<DF>(), deq, count);
  cout << "\nSo sanh doi voi thao tac duyet:" << endl;
  measureTime(Traversal<VF>(), vec, count);
  measureTime(Traversal<DF>(), deq, count);
  measureTime(Traversal<LF>(), lst, count);
  cout << "\nSo sanh doi voi thao tac hoan doi noi dung giua tren
bo  chua (khong thuc hien tren list):" << endl;
  measureTime(Swap<VF>(), vec, count);
  measureTime(Swap<DF>(), deq, count);
  measureTime(Swap<LF>(), lst, count);
  cout << "\nSo sanh doi voi thao tac xoa o giua bo chua:" <<
endl;
  measureTime(RemoveMiddle<VF>(), vec, count);
  measureTime(RemoveMiddle<DF>(), deq, count);
  measureTime(RemoveMiddle<LF>(), lst, count);
  vec.resize(vec.size() * 10); // Tăng kích thước
  cout << "\nSo sanh doi voi thao tac xoa bo" <<endl;
  measureTime(RemoveBack<VF>(), vec, count);
  measureTime(RemoveBack<DF>(), deq, count);
  measureTime(RemoveBack<LF>(), lst, count);
  return 0;
```

Chương trình trên được chạy trên máy có cấu hình Pentium III 800MHz, 256 MBRAM, hệ điều hành Windows 2000 Pro. Với số lượng các phần tử cho từng bộ chứa là 1000, kết quả như sau:

```
So sanh doi voi thao tac bo sung vao cuoi bo chua:
Thao tac bo sung vao cuoi bo chua tren vector: 0
Thao tac bo sung vao cuoi bo chua tren vector: 0
Thao tac bo sung vao cuoi bo chua tren deque: 0
Thao tac bo sung vao cuoi bo chua tren list: 0

So sanh doi voi thao tac bo sung vao dau  bo chua (khong thuc
hien tren vector):
Thao tac bo sung vao dau bo chua tren deque: 10
Thao tac bo sung vao dau bo chua tren list: 30
```

```
So sanh doi voi thao tac bo sung o giua bo chua :
Thao tac bo sung vao giua bo chua tren vector: 10
Thao tac bo sung vao giua bo chua tren deque: 0
Thao tac bo sung vao giua bo chua tren list: 0

So sanh doi voi thao tac nhap ngau nhien tren bo chua (khong
thuc hien tren list):
Thao tac truy nhap ngau nhien tren bo chua tren vector: 30
Thao tac truy nhap ngau nhien tren bo chua tren deque: 80

So sanh doi voi thao tac duyet:
Thao tac duyet bo chua tren vector: 0
Thao tac duyet bo chua tren deque: 50
Thao tac duyet bo chua tren list: 70

So sanh doi voi thao tac hoan doi noi dung giua tren bo  chua
(khong thuc hien tren list):
Thao tac hoan doi noi dung bo chua tren vector: 0
Thao tac hoan doi noi dung bo chua tren deque: 10
Thao tac hoan doi noi dung bo chua tren list: 10

So sanh doi voi thao tac xoa o giua bo chua:
Thao tac xoa o giua tren vector: 10
Thao tac xoa o giua tren deque: 0
Thao tac xoa o giua tren list: 0

So sanh doi voi thao tac xoa bo
Thao tac xoa o cuoi tren vector: 0
Thao tac xoa o cuoi tren deque: 10
Thao tac xoa o cuoi tren list: 30
```

Đối với máy cấu hình như trên và số lượng phần tử là 1000, sự khác biệt chưa thực sự rõ ràng. Kết quả chạy với số lượng phần tử 10000 cho minh hoạ rõ hơn:

```
So sanh doi voi thao tac bo sung vao cuoi bo chua:
Thao tac bo sung vao cuoi bo chua tren vector: 50
Thao tac bo sung vao cuoi bo chua tren vector: 20
Thao tac bo sung vao cuoi bo chua tren deque: 10
Thao tac bo sung vao cuoi bo chua tren list: 30

So sanh doi voi thao tac bo sung vao dau  bo chua (khong thuc
hien tren vector):
Thao tac bo sung vao dau bo chua tren deque: 110
Thao tac bo sung vao dau bo chua tren list: 310

So sanh doi voi thao tac bo sung o giua bo chua :
Thao tac bo sung vao giua bo chua tren vector: 5018
```

```
Thao tac bo sung vao giua bo chua tren deque: 10
Thao tac bo sung vao giua bo chua tren list: 10

So sanh doi voi thao tac nhap ngau nhien tren bo chua (khong
thuc hien tren list):
Thao tac truy nhap ngau nhien tren bo chua tren vector: 270
Thao tac truy nhap ngau nhien tren bo chua tren deque: 851

So sanh doi voi thao tac duyet:
Thao tac duyet bo chua tren vector: 10
Thao tac duyet bo chua tren deque: 4808
Thao tac duyet bo chua tren list: 6529

So sanh doi voi thao tac hoan doi noi dung bo chua:
Thao tac hoan doi noi dung bo chua tren vector: 40
Thao tac hoan doi noi dung bo chua tren deque: 70
Thao tac hoan doi noi dung bo chua tren list: 91

So sanh doi voi thao tac xoa o giua bo chua:
Thao tac xoa o giua tren vector: 5197
Thao tac xoa o giua tren deque: 10
Thao tac xoa o giua tren list: 0

So sanh doi voi thao tac xoa bo
Thao tac xoa o cuoi tren vector: 71
Thao tac xoa o cuoi tren deque: 110
Thao tac xoa o cuoi tren list: 320
```

Kết quả cho thấy lớp `vector` thích hợp với các thao tác: truy nhập ngẫu nhiên, duyệt qua bộ chứa, hoán đổi nội dung hai bộ chứa và xoá đi phần tử cuối. Nhưng `vector` có mất khá nhiều thời gian cho các thao tác bổ sung phần tử. Tuy nhiên, trong trường hợp bổ sung phần tử ở vị trí cuối và `vector` đã được cấp phát trước, chi phí ở mức chấp nhận được. Lớp `deque` hiệu quả với các thao tác chèn ở hai đầu bộ chứa. Thời gian thực hiện thao tác này trên deque có thể xem là hằng số, không phụ thuộc vào kích thước của `deque`. Lớp `list` cũng hiệu quả với các thao tác cập nhật phần tử, đặc biệt là thao tác xoá một phần tử nằm giữa danh sách.

Dựa trên kết quả này, ta có thể đưa ra các lựa chọn bộ chứa cho từng ứng dụng một cách phù hợp nhất. Đối với các ứng dụng đòi hỏi nhiều thao tác truy nhập ngẫu nhiên cũng như duyệt qua bộ chứa, `vector` là lựa chọn số một. Đối với các thao tác cập nhật phần tử ở các vị trí bất kỳ, ta có thể chọn `deque` hoặc `list`. Trong trường hợp ứng dụng có yêu cầu cả các thao tác truy nhập ngẫu nhiên, `deque` được ưu tiên hơn `list`. Tuy nhiên, trên thực tế,

nhu cầu của các ứng dụng rất đa dạng. Do vậy, nếu chỉ sử dụng một loại bộ chứa sẽ không đảm bảo tính tối ưu cho chương trình. Để đảm bảo tính tối ưu cho chương trình, bạn nên thiết kế chương trình thành từng đoạn, mỗi đoạn có thao tác chủ đạo. Sau đó, ta sử dụng các cơ chế chuyển đổi nội dung giữa các bộ chứa tuần tự để có thể tối ưu hiệu quả cho từng đoạn mã. Ví dụ, khi nhập liệu, ta sử dụng `deque`. Còn khi tra cứu, ta lại sử dụng `vector`.

Chuyển đổi dữ liệu giữa các bộ chứa tuần tự

Rõ ràng, mỗi một loại bộ chứa tuần tự được thiết kế cho những mục đích sử dụng riêng và nó được tổ chức tối ưu cho mục đích này. Tuy nhiên, có khá nhiều bài toán thực tế nếu chỉ sử dụng một loại bộ chứa chỉ cho phép chương trình được tối ưu trong một giai đoạn nào đó. Ví dụ, chương trình ứng dụng cần sự hiệu quả của `deque` khi bổ sung dữ liệu trong gian đoạn nhập liệu. Nhưng nó cũng cần hiệu quả của `vector` khi đánh chỉ số dữ liệu. Điều này đặt ra yêu cầu chuyển đổi dữ liệu giữa các dạng bộ chứa.

Nói chung, người ta thường dùng `deque` để nhập dữ liệu do các thao tác cập nhật có chi phí thấp hơn nhiều so với `vector`. Sau đó, dữ liệu được chuyển sang một `vector` bằng hàm `copy` hoặc sử dụng cấu tử-sao chép của `vector`. Trong trường hợp này, cấu tử sao chép của `vector` có dạng một khuôn hình hàm của hàm thành phần với tham số hoàn toàn độc lập với tham số khuôn hình lớp.

```
Template<class InputIterator>
vector(InputIterator first,InputIterator last)
```

Cần nhắc lại là khái niệm khuôn hình của hàm thành phần (template member function) mới chỉ được đưa vào chuẩn năm 1998. Do vậy, không phải trình dịch nào cũng hỗ trợ. Trong trường hợp trình dịch của bạn hỗ trợ khuôn hình của hàm thành phần, tham số Input Iterator có thể là bất cứ lớp bộ duyệt nào. Trong trường hợp ngược lại, tham số cho cấu tử này chỉ có thể là `const value_type*`.

`deque` cũng có một cấu tử sao chép tương tự, khuôn dạng của cấu tử như sau:

```
Template<class InputIterator>
deque(InputIterator first,InputIterator last)
```

Trong trường hợp trình dịch không hỗ trợ khuôn hình của hàm thành phần, đối số của cấu tử này là `const value_type*` hoặc `const deque<T>::iterator`.

Ngoài việc sử dụng các cấu tử dạng trên, các lớp bộ chứa tuần tự cũng cung cấp một số hàm để sao chép nội dung giữa hai bộ chứa khác loại. Ví dụ đơn giản sau sẽ minh hoạ cho cách làm trên.

```
#include <vector>
#include <deque>
#include <algorithm>
#include "../Student.h"

int main(int argc, char* argv[])
{
    student student_array[5] =
{student(),student(),student(),student(),student()};
    vector<student> student_vector_1(student_array,student_array
+ 5);
    cout << "Toan bo danh sach duoc ghi trong vector thu
nhat\n";
    copy(student_vector_1.begin(),student_vector_1.end(),ostream
_iterator<student>(cout,"\n"));
    deque<student>
student_deque(student_vector_1.begin(),student_vector_1.end());
    cout << "Toan bo danh sach duoc chuyen sang deque:\n";
    copy(student_deque.begin(),student_deque.end(),ostream_itera
tor<student>(cout,"\n"));
    vector<student> student_vector_2(student_deque.begin()+
2,student_deque.end());
    cout << "Mot phan cua danh sach duoc chuyen tu deque sang
vector thu hai:\n";
    copy(student_vector_1.begin(),student_vector_1.end(),ostream
_iterator<student>(cout,"\n"));
    return 0;
}
```

Sau khi biến `student_vector_1` được khởi tạo với nội dung lấy từ `student_array`, biến `student_deque` lại được khởi tạo với các phần tử được chép sang từ `student_vector_1`. Công việc này được thực hiện lại một lần nữa nhưng theo chiều ngược lại: một phần nội dung của `student_deque` được chép sang `student_vector_2`.

Ngoài cách trên, bạn vẫn có thể thực hiện việc chuyển đổi bằng hàm `assign()`. Hàm `assign()` là sự cụ thể hoá một phương thức của bộ chứa tuần tự, nghĩa là các lớp `vector`, `deque`, `list` đều có hàm này.

```
#include <vector>
#include <deque>
```

```
#include <algorithm>
#include "../Student.h"

int main(int argc, char* argv[])
{
    student student_array[5] =
{student(),student(),student(),student(),student()};
    vector<student> student_vector;
    student_vector.assign(student_array,student_array+5);
    cout << "Toan bo danh sach duoc luu tru trong vector thu
nhat:\n";
    copy(student_vector.begin(),student_vector.end(),ostream_ite
rator<student>(cout,"\n"));
    deque<student> student_deque;
    student_deque.assign(student_vector.begin(),student_vector.e
nd());
    cout << "Toan bo danh sach duoc chuyen sang deque:\n";
    copy(student_deque.begin(),student_deque.end(),ostream_itera
tor<student>(cout,"\n"));
    return 0;
}
```

Lưu ý rằng, `assign()` cũng là một khuôn hình hàm thành phần. Do vậy, ta cần lựa chọn môi trường thích hợp để thực hiện ví dụ.

Nếu trình dịch bạn sử dụng không hỗ trợ khuôn hình hàm thành phần, bạn vẫn có thể thực hiện việc chuyển đổi bằng hàm `copy()`. Cách sử dụng hàm `copy()` cho việc chuyển đổi cũng giống như dùng hàm `copy()` để hiển thị nội dung bộ chứa ra màn hình như đã làm trong các ví dụ trước. Tuy nhiên, tham số thứ ba không còn là một `ostream_iterator`. Nó là một trong ba lớp bộ duyệt bổ sung phần tử bao gồm `back_insert_iterator`, `front_insert_iterator` và `insert_iterator`.

```
#include <vector>
#include <deque>
#include <list>
#include <algorithm>
#include <iterator>
#include "../Student.h"


int main(int argc, char* argv[])
{
```

```
    student student_array[5] =
{student(),student(),student(),student(),student()};
    vector<student> student_vector(student_array,student_array +
5);
    cout << "Toan bo danh sach duoc luu tru trong vector thu
nhat:\n";
    copy(student_vector.begin(),student_vector_1.end(),ostream_i
terator<student>(cout,"\n"));
    deque<student> student_deque;
    copy(student_vector.begin(),student_vector.end(),back_insert
_iterator<deque<student> >(student_deque));
    cout << "Toan bo danh sach duoc chuyen sang deque:\n";
    copy(student_deque.begin(),student_deque.end(),ostream_itera
tor<student>(cout,"\n"));
    list<student> student_list;
    copy(student_deque.begin(),student_deque.end(),front_insert_
iterator<list<student> >(student_list));
    cout << "Mot phan danh sach duoc chuyen tu deque sang vector
thu hai:\n";
    copy(student_list.begin()+2,student_list.end(),ostream_itera
tor<student>(cout,"\n"));
    return 0;
}
```

Lưu ý rằng, không phải bất kỳ bộ chứa nào đều cũng có thể sử dụng ba lớp kể trên. Ví dụ, `vector` không thể sử dụng cùng với lớp `front_insert_iterator`. Nếu thay dòng lệnh

```
copy(student_vector.begin(),student_vector.end(),back_insert_iter
ator<deque<student> >(student_deque));
```

bằng dòng lệnh

```
copy(student_vector.begin(),student_vector.end(),back_insert_iter
ator<vector<student> >(student_vector_1));
```

với `student_vector_1` là một biến kiểu `vector<student>` thì sẽ nhận được thông báo lỗi: lớp `vector` không có hàm `push_front()`.

Một cách tổng quát, bất kỳ lớp bộ chứa nào có định nghĩa hàm `insert()` đều có thể sử dụng với lớp `insert_iterator`. Lớp nào định nghĩa hàm `push_back()` (các lớp xuất phát từ KHÁI NIỆM Back Insert Container) thì sử dụng được `back_insert_iterator`. Và lớp nào định nghĩa hàm `push_front()` (các lớp xuất phát từ KHÁI NIỆM Front Insert Container) thì làm việc được với lớp `front_insert_iterator`. Theo

nguyên tắc này, vector chỉ có thể sử dụng cùng insert_iterator và back_insert_iterator, còn deque và list làm việc cùng với cả ba lớp bộ duyệt bổ sung phần tử. Chi tiết về các bộ duyệt này, chúng ta sẽ cùng bàn tới trong chương tiếp sau.

2.3 Các lớp bộ chứa liên kết

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

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

2.3.1 Lớp set

Lớp set<> được sử dụng để lưu trữ các đối tượng có kiểu được chỉ định qua tham số Key của khuôn hình lớp. Các đối tượng trong set được tổ chức lưu trữ theo một trật tự nhất định, trật tự này được chỉ định bằng tham số thứ hai. Cách thức lưu trữ được xác định bằng tham số thứ ba. Khi hai tham số thứ hai và thứ ba không được chỉ định tường minh, các giá trị mặc định cho hai tham số này được sử dụng.

Nói tóm lại, set được dùng lưu trữ một tập các đối tượng theo một trật tự nhất định sao cho mỗi đối tượng chỉ xuất hiện nhiều nhất một lần. Sử dụng lớp set cho một ứng dụng đơn giản được minh hoạ trong ví dụ sau:

```
#include <set>
#include <iostream>
#include <algorithm>
#include <cstdlib>

using namespace std;
```

```
int main(int argc, char* argv[])
{
    float float_array[10] = {0,1,1,2,2,2,3,3,3,3};
    set<float> float_set;
    float_set.insert(float_array,float_array+10);
    set<int> int_set;
    srand(0);
    for(int i = 0;i<100;i++)
        int_set.insert(rand() % 25);
    cout << "Tap so thuc:" << endl;
    copy(float_set.begin(),float_set.end(),ostream_iterator<floa
t>(cout," "));
    cout << endl;
    cout << "Tap so nguyen:" << endl;
    copy(int_set.begin(),int_set.end(),ostream_iterator<int>(cou
t," "));
    cout << endl;
    return 0;
}
```

```
Tap so thuc:
0 1 2 3
Tap so nguyen:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24
```

Ví dụ trên minh hoạ cách khai báo và sử dụng lớp `set` một cách đơn giản. Hai tập số nguyên và số thực được sinh ra theo hai cách khác nhau. Theo cách thứ nhất, các đối tượng được lấy từ một mảng. Theo cách thứ hai, các đối tượng được lần lượt thêm vào trong mỗi bước lặp bằng hàm `insert()`. Hàm `insert()` còn có một phiên bản khác nữa nhưng nó ít được sử dụng. Sau khi hai tập được sinh ra, các đối tượng của nó được liệt kê ra màn hình với hàm `copy()` như đã làm cho các bộ chứa khác. Kết quả cho thấy, các đối tượng được sắp theo thứ tự tăng dần và chỉ xuất hiện nhiều nhất là một lần trong `set`. Kết quả sẽ khác rất nhiều nếu như ta thay đổi khai báo của hai tập số nguyên và số thực trên. Thay vì dùng đối tượng hàm so sánh mặc định là `less<_Ty>`, đối tượng hàm `greater<_Ty>` sẽ được dùng cho `float`.

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

```
using namespace std;

int main(int argc, char* argv[])
{
     float float_array[10] = {0,1,1,2,2,2,3,3,3,3};
     set<float,greater<float> > float_set;
     float_set.insert(float_array,float_array+10);
     set<int,not_equal_to<int> > int_set;
     srand(0);
     for(int i = 0;i<100;i++)
          int_set.insert(rand() % 25);
     cout << "Tap so thuc:" << endl;
     copy(float_set.begin(),float_set.end(),ostream_iterator<floa
t>(cout," "));
     cout << endl;
     cout << "Tap so nguyen:" << endl;
     copy(int_set.begin(),int_set.end(),ostream_iterator<int>(cou
t," "));
     cout << "\nKich thuoc cua tap: " << int_set.size() << endl;
     return 0;
}
```

Kết quả thu được như sau:

```
Tap so thuc:
3 2 1 0
Tap so nguyen:
5 19 15 11 24 20 18 12 11 14 19 4 21 0 22 17 3 13 22 23 6 12 4
2 11 14 6 1 4 24 13 5 4 16 11 9 22 12 23 8 6 17 11 7 12 15 21 3
15 23 7 5 16 1 22 10 20 21 12 6 17 3 12 0 10 15 22 5 12 19 13
Kich thuoc cua tap: 71
```

Lưu ý rằng, kết quả trên không có nghĩa là tính duy nhất của các phần tử trong `set` không được bảo đảm. Lúc này, quan hệ tương đương giữa hai số nguyên được xét theo một quan hệ khác chứ không còn là quan hệ ngang bằng thông thường. Quan hệ tương đương giữa hai phần tử trong `set` được xác định dựa trên `key_compare<>` theo quy tắc sau: hai phần tử `x`, `y` trong bộ chứa được gọi là trùng nhau nếu `key_compare(x,y)` và `key_compare(y,x)` đồng thời nhận giá trị `false`. Với quy tắc này, lớp phần tử của `set<>` không nhất thiết phải hỗ trợ toán tử ==, nhưng cũng không đảm bảo rằng các phần tử trong lớp `set` không trùng nhau theo nghĩa thông thường. Ví dụ, ta có lớp `CInt` được định nghĩa như sau:

```
#include <iostream>

using namespace std;

class CInt{
    int core;
public:
    CInt(){};
    CInt(int int_num){
        core = int_num;
    };
    CInt(const CInt& int_num){
        core = int_num.core;
    };
    bool operator!=(const CInt& int_num)const{
        return core != int_num.core;
    };
    friend ostream& operator<<(ostream& out,const CInt&
int_number){
        out << int_number.core;
        return out;
    }

};
```

Chương trình sau sẽ minh hoạ cho nhận xét trên:

```
#include <set>
#include "IntNumber.h"

int main(int argc, char* argv[])
{
    CInt int_array[10] = {CInt(2), CInt(0), CInt(1),
CInt(1),CInt(2),CInt(1),CInt(1),CInt(2),CInt(1),CInt(0)};
    typedef set<CInt,not_equal_to<CInt> > CIntSet;
    srand(0);
    CIntSet int_set;
    pair<CIntSet::iterator,bool> inserted_element;
    for(int i = 0;i<10;i++)
    {
        inserted_element = int_set.insert(int_array[i]);
        if(!inserted_element.second)
            cout << int_array[i] << " da co trong tap dang
xet, no tuong duong voi " << *(inserted_element.first) << endl;
        else
```

```
                    cout << *(inserted_element.first) << " da duoc
bo sung them vao tap dang xet\n";

     copy(int_set.begin(),int_set.end(),ostream_iterator<CInt>(co
ut," "));
            cout << endl;

     }
     return 0;
}
```

Kết quả của chương trình:

```
2 da duoc bo sung them vao tap dang xet
2
0 da duoc bo sung them vao tap dang xet
0 2
1 da duoc bo sung them vao tap dang xet
1 0 2
1 da co trong tap dang xet, no tuong duong voi 1
1 0 2
2 da duoc bo sung them vao tap dang xet
2 1 0 2
1 da co trong tap dang xet, no tuong duong voi 1
2 1 0 2
1 da co trong tap dang xet, no tuong duong voi 1
2 1 0 2
2 da co trong tap dang xet, no tuong duong voi 2
2 1 0 2
1 da co trong tap dang xet, no tuong duong voi 1
2 1 0 2
0 da co trong tap dang xet, no tuong duong voi 0
2 1 0 2
```

Để đảm bảo ràng buộc về trật tự cũng như tối ưu thời gian thực hiện các thao tác bổ sung hay xoá các phần tử bộ chứa, các phần tử trong `set` nói riêng và trong một bộ chứa có thứ tự nói chung được tổ chức lưu trữ theo một cây nhị phân tìm kiếm. Trong cây nhị phân này, mỗi nút của cây ứng với một phần tử. Ngoài ra, cây được tổ chức sao cho: tại một nút `x` bất kỳ thì `key_compare(x, left(x))` luôn là giá trị `True`, còn `key_compare(x, right(x))` thì luôn nhận giá trị `False` với `left(x)` và `right(x)` là một nút thuộc nhánh trái hoặc phải tương ứng của cây con có `x` là nút gốc. Các thủ tục bổ sung, xóa được thực hiện theo các thuật toán của cây nhị phân tìm kiếm. Đối với các lớp bộ chứa liên kết đơn nhất, thủ tục bổ sung phần tử có thêm một phần kiểm tra tính duy nhất của phần tử sau khi

đã xác định được vị trí có thể của phần tử mới. Thứ tự tuyến tính của bộ chứa dựa trên phép duyệt cây tiền tố.

Yêu cầu sắp xếp các phần tử trong `set` theo một trật tự nhất định là yêu cầu của KHÁI NIỆM Sorted Associative Container. Sự sắp xếp này nhằm tối ưu hoá các tác vụ liên quan tới tìm kiếm như hai hàm `lower_bound()` và `upper_bound()`. Cả hai hàm này đều có đối số là một giá trị kiểu lớp khoá (tham số Key trong khuôn hình `set<>`). Hàm `lower_bound(key_value)` sẽ tìm ra vị trí của của phần tử đầu tiên (tính từ đầu tới cuối bộ chứa) trong bộ chứa mà `key_comp(key_value,x)` nhận giá trị False. Hàm `upper_bound(key_value)` trả về vị trí của phần tử đầu tiên trong bộ chứa tại đó `key_comp(x,key_value)` nhận giá trị True. Ví dụ, đối với `set<_Ty,less<_Ty>>`, `lower_bound(key_value)` trả về ví trị phần tử đầu tiên trong `set` lớn hơn hoặc bằng `key_value`. Trong trường hợp bộ chứa có chứa các phần tử với khoá trùng với `key_value`, vị trí của phần tử đầu tiên trong các phần tử này sẽ được trả về. Hàm `upper_bound(key_value)` sẽ tìm vị trí của phần tử đầu tiên trong bộ chứa lớn hơn `key_value`. Trong trường hợp bộ chứa có chứa các phần tử có khoá trùng với `key_value`, hàm này sẽ trả về vị trí sau phần tử cuối cùng trong các phần tử này.

```
#include <set>
#include "../set03/IntNumber.h"

int main(int argc, char* argv[])
{
    typedef set<CInt,less<CInt> > CIntSet;
    srand(0);
    CIntSet::iterator it;
    CInt int_array[10] =
{CInt(1),CInt(5),CInt(4),CInt(2),CInt(1),CInt(1),CInt(4),CInt(6),
CInt(7)};
    CIntSet int_set(int_array,int_array+9);
    copy(int_set.begin(),int_set.end(),ostream_iterator<CInt>(co
ut," "));
    it = int_set.lower_bound(CInt(3));
    if(it != int_set.end())
        cout << "\nCan duoi cua 3 trong tap dang xet la " <<
*it << endl;
    else
        cout << "\n3 khong co trong tap dang xet" << endl;
    it = int_set.upper_bound(CInt(3));
```

```
    if(it != int_set.end())
            cout << "Can tren cua 3 trong tap dang xet la " <<
*it << endl;
    else
            cout << "3 khong co trong tap dang xet" << endl;
    cout << endl;
    return 0;
}
```

Kết quả đoạn mã trên như sau:

```
1 2 4 5 6 7
Can duoi cua 3 trong tap dang xet la 4
Can tren cua 3 trong tap dang xet la 4
```

Đi kèm với hai hàm `upper_bound()` và `lower_bound()` là hàm `equal_range()`. Hàm này xác định hai vị trí biên của một khoảng sao cho khi xét theo quan hệ `key_compare`, các phần tử trong khoảng này tương đương với đối số của hàm. Đối với `set`, hàm này luôn trả về một khoảng rỗng nếu đối số không có mặt trong `set` hoặc trả về khoảng chỉ có một phần tử chính là phần tử có giá trị trùng với đối số của hàm.

```
#include <set>
#include "../set03/IntNumber.h"

int main(int argc, char* argv[])
{
    typedef set<CInt> CIntSet;
    srand(0);
    CInt int_array[10] =
{CInt(1),CInt(5),CInt(4),CInt(2),CInt(1),CInt(1),CInt(4),CInt(6),
CInt(7)};
    CIntSet int_set(int_array,int_array+9);
    copy(int_set.begin(),int_set.end(),ostream_iterator<CInt>(co
ut," "));
    pair<CIntSet::iterator,CIntSet::iterator> range_bound;
    range_bound = int_set.equal_range(CInt(3));
    cout << "\n3 bang voi moi phan tu trong khoang [" <<
*(range_bound.first) << "," << *(range_bound.second) << "]" <<
endl;
    for(CIntSet::iterator it = range_bound.first;it !=
range_bound.second;it++)
            number_equal_element++;
    cout << "Khoang nay chua " << number_equal_element << "
phan tu\n";
```

```
    return 0;
}
```

```
1 2 3 4 5 6 7
3 bang voi moi phan tu trong khoang (3,4)
Khoang nay chua 1 phan tu
```

Nói chung, các hàm `upper_bound()`, `lower_bound()` hay `equal_range()` chỉ thực sự có ý nghĩa với các bộ chứa được cụ thể hoá từ KHÁI NIỆM Multiple Associative Container. Đó là các bộ chứa liên kết cho phép có sự trùng lặp về khoá. Chúng ta sẽ trở lại với các hàm này một cách chi tiết hơn khi nói về các bộ chứa liên kết phức.

So với `map`, `set` có một khác biệt cơ bản. Được xếp vào lớp các bộ chứa liên kết đơn giản, `set::key_type` và `set::value_type` là một. Do vậy, ta có thể truy nhập các phần tử với bộ duyệt giống như đối với các bộ chứa tuần tự

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

using namespace std;

int main(int argc, char* argv[])
{
    set<int> int_set;
    for(int i = 0;i<10;i++)
        int_set.insert(rand() % 5);
    set<int>::iterator it;
    for(it = int_set.begin();it != int_set.end();it++)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

Kết quả chương trình

```
0 1 2 3 4
```

Các bộ chứa là mô hình của KHÁI NIỆM Simple Associative Container thường được dùng với các khuôn hình giải thuật trên tập hợp như hợp của hai tập hợp (`set_union`), giao của hai tập hợp (`set_intersection`) và

phép trừ (set_difference). Trên thực tế, các giải thuật này không chỉ làm việc trên set, nó vẫn làm việc được trên các bộ chứa khác như vector, deque hay list. Chi tiết về các khuôn hình giải thuật này sẽ được nói kỹ trong các chương sau. Ví dụ sau đây minh hoạ cách sử dụng các khuôn hình giải thuật này với set.

```
#include <set>
#include <iterator>
#include <algorithm>
#include "../set03/IntNumber.h"
int main(int argc, char* argv[])
{
    typedef set<CInt> CIntSet;
    CInt int_array[10] =
{CInt(1),CInt(5),CInt(4),CInt(2),CInt(3),CInt(1),CInt(4),CInt(6),
CInt(7)};
    CIntSet A(int_array,int_array+5);
    CIntSet B(int_array+4,int_array+9);
    cout << "Tap A = {";
    copy(A.begin(),A.end(),ostream_iterator<CInt>(cout," "));
    cout << "}\nTap B = {";
    copy(B.begin(),B.end(),ostream_iterator<CInt>(cout," "));
    cout << "}\nGiao cua A va B = {";
    set_intersection(A.begin(),A.end(),B.begin(),B.end(),ostream
_iterator<CInt>(cout," "));
    cout << "}\nHop cua A va B = {";
    set_union(A.begin(),A.end(),B.begin(),B.end(),ostream_iterat
or<CInt>(cout," "));
    cout << "}\nHieu cua A va B = {";
    set_difference(A.begin(),A.end(),B.begin(),B.end(),ostream_i
terator<CInt>(cout," "));
    cout <<"}\n";
    return 0;
}
```

Kết quả chương trình

```
Tap A = {1 2 3 4 5 }
Tap B = {1 3 4 6 7 }
Giao cua A va B = {1 3 4 }
Hop cua A va B = {1 2 3 4 5 6 7 }
hieu cua A va B = {2 5 }
```

Trong ví dụ trên, kết quả được gửi trực tiếp ra màn hình mà không được lưu lại trong bất kỳ một biến nào. Để lưu kết quả vào một biến ta sửa lại các

dòng thực hiện phép toán như sau:

```
set_intersection(A.begin(),A.end(),B.begin(),B.end(),insert_itera
tor<CIntSet>(C,C.begin()));

set_union(A.begin(),A.end(),B.begin(),B.end(),insert_iterator<CIn
tSet>(D,D.begin()));

set_difference(A.begin(),A.end(),B.begin(),B.end(),insert_iterato
r<CIntSet>(E,E.begin()));
```

Trong đó, C,D và E là các biến kiểu `CIntSet`. Kết quả chương trình sẽ không khác nếu ta in nội dung của C, D và E ra màn hình với khuôn hình giải thuật `copy()` như đã làm đối với A và B.

2.3.2 Lớp map

Lớp `map` thực sự là một bộ chứa liên kết. Nó được thiết kế để lưu trữ quan hệ giữa hai tập đối tượng. Mỗi phần tử của `map` là một cặp đối tượng. Đối tượng thứ nhất thuộc lớp `Key` và đối tượng thứ hai thuộc lớp `Data`. Hai lớp `Key` và `Data` là hai tham số bắt buộc khi khai báo một lớp `map`. Lớp `Key` phải hỗ trợ một số toán tử phục vụ việc sắp xếp các phần tử trong bộ chứa. Ràng buộc này đối với tham số `Key` của `map` cũng giống như đối với tham số `Key` của lớp `set`. Ví dụ, trong trường hợp tham số `Compare` nhận giá trị mặc định, các phần tử trong `map` được sắp xếp dựa trên quan hệ "nhỏ hơn" giữa các khoá. Tham số `Data` có thể là một kiểu bất kỳ, không có ràng buộc gì. Ví dụ sau đây minh hoạ cách tạo ra một bảng chỉ số dựa trên số hiệu sinh viên. Bảng này cho phép tạo ra các tham chiếu nhanh trên một tập các sinh viên dựa theo số hiệu của họ.

```
#include <map>
#include "../Student.h"

using namespace std;

int main(int argc, char* argv[])
{
     typedef map<int,student> IdStudentMap;
     IdStudentMap student_index_table;
     for(int i = 0;i < 10;i++)

     student_index_table.insert(pair<int,student>(i,student(i)));
```

```
    IdStudentMap::iterator it;
    for(it = student_index_table.begin();it !=
student_index_table.end();it++)
        cout << it->second << endl;
    return 0;
}
```

Ban đầu, bảng chỉ số được tạo ra trên một tập các sinh viên có số hiệu từ 0 tới 25. Các cặp (chỉ số, sinh viên) được bổ sung dần bảng bằng lệnh `insert()`. Cũng giống như các bộ chứa khác, hàm `insert()` bổ sung thêm một phần tử vào bộ chứa và nhận đối số có kiểu là kiểu phần tử. Kiểu phần tử của `map<Key,Data>` là `pair<Key,Data>`. Do vậy, trước khi bổ sung một cặp gồm khoá và dữ liệu, ta phải tạo ra một đối tượng kiểu `pair<Key,Data>` từ hai đối tượng trên kiểu `Key` và `Data`. Trong ví dụ trên, ta sử dụng cấu tử của lớp `pair<Key, Data>`. Đoạn chương trình tạo bảng chỉ số trong ví dụ trên có thể được viết lại tường minh hơn như sau:

```
pair<int,student> a_pair;
for(int i = 0;i < 10;i++){
    a_pair = make_pair(i,student(i));
    student_index_table.insert(a_pair);
}
```

Hàm `make_pair()` cho phép tạo ra một cặp từ hai đối tượng thành phần. Ngoài hàm này, lớp `pair<_Ty,_Tx)` còn có hai biến thành phần được sử dụng khá nhiều đó là `first` và `second`. Biến `first` tham chiếu tới đối tượng thứ nhất trong bộ đôi (cặp) Biến `second` tham chiếu tới đối tượng thứ hai. Cách sử dụng hai biến này có thể xem trong đoạn duyệt bảng chỉ số và in ra danh sách sinh viên trong bảng của ví dụ trên.

```
#include <map>
#include "../Student.h"

int main(int argc, char* argv[])
{
    typedef map<int,student> IdStudentMap;
    IdStudentMap student_index_table;
    for(int i = 0;i < 10;i++)
        student_index_table[i] = student(i);
    IdStudentMap::iterator it;
    for(it = student_index_table.begin();it !=
student_index_table.end();it++)
```

```
            cout << it->second << endl;
    return 0;
}
```

Nhìn qua đoạn mã ta cảm tưởng có gì đó không hợp lý vì `it` là bộ duyệt của `map<Key, Data>` còn `first` và `second` là biến thành phần của lớp `pair<_Ty, _Tx>` Tuy nhiên, nếu nhìn kỹ hơn, ta thấy hoàn toàn hợp lý. Bộ duyệt của một bộ chứa thực chất có thể xem là con trỏ của kiểu phần tử, kiểu phần tử của `map<Key, Data>` là `pair<Key, Data>`. Do vậy, con trỏ của `map<>` được nhìn nhận như một con trỏ trỏ tới một đối tượng kiểu `pair<>`

Mặc dù, hai cách làm trên có vẻ chính quy, nhưng nó lại ít được sử dụng. Cách được dùng phổ biến là sử dụng toán tử `[]` của `map`. Toán tử `[]` của `map` linh hoạt hơn rất nhiều so với các toán tử `[]` của `vector` hay `deque`. Đối số của nó không chỉ đơn thuần là chỉ số kiểu nguyên mà là một đối tượng của lớp `Key`. Hơn thế, nó vừa cho phép truy nhập phần dữ liệu của một phần tử thông qua khoá vừa cho phép tạo ra một phần tử hoặc sửa đổi phần dữ liệu của phần tử có khoá, trùng với giá trị của đối số. Việc tạo ra hay sửa đổi một phần tử được thực hiện với cùng một phép gán. Thao tác nào được thực hiện tuỳ thuộc vào phần tử đã tồn tại trong `map` hay chưa. Nếu phần tử với khoá trùng với giá trị với đối số của toán tử `[]` đã có trong bộ chứa thì phép gán được hiểu như việc sửa đổi nội dung của phần tử. Trong trường hợp ngược lại, một phần tử mới được thêm vào. Phần tử này có khoá là đối số của phép gán và dữ liệu là vế phải của phép gán. Để in ra danh sách của toàn bộ sinh viên trong bảng chỉ số trên, ta có thể làm theo cách sau:

```
for(i = 0;i < student_index_table.size();i++)
    cout << student_index_table[i] << endl;
```

Lưu ý rằng, ta không sử dụng hàm `copy()` để hiển thị một phần nội dung của `map` như giống như các bộ chứa khác vì bộ duyệt của lớp `map` chỉ tới một cặp chứ không phải chỉ tới một phần tử. Trong trường hợp muốn sử dụng hàm `copy()` như đã làm với các bộ chứa trên, ta phải xây dựng toán tử `<<` cho lớp `pair<int,student>`. Khi đó, ví dụ trên có thể viết lại như sau mà không hề làm thay đổi kết quả hiển thị trên màn hình.

```
#include "stdafx.h"
#include <map>
#include <iostream>
```

```
#include "../Student.h"

using namespace std;

ostream& operator<<(ostream& out,const pair<int,student>&
student_desc)
{
     out << student_desc.second;
     return out;
}
int main(int argc,char* argv[])
{
     typedef map<int,student> IdStudentMap;
     IdStudentMap student_index_table;
     for(int i = 0;i < 10;i++)
            student_index_table[i] = student(i);
     copy(student_index_table.begin(),student_index_table.end(),
            ostream_iterator<pair<int,student> >(cout,"\n"));
}
```

Trong ví dụ trên, bạn có thể tổng quát hoá toán tử << để có thể sử dụng cho các map khác theo cách sau:

```
template <typename _Ty1,typename _Ty2>
ostream& operator<<(ostream& out,const pair<_Ty1,_Ty2>& a_pair)
{
     out << "(" << a_pair.first << "," << a_pair.second << ")";
     return out;
}
```

Có một vấn đề nhỏ khi ta triển khai toán tử << . Mặc dù đã viết toán tử << như trên, nhưng nếu ta dùng hàm `copy()` để hiển thị nội dung của một `map<int,int>`, vẫn gặp thông báo lỗi: "toán tử << với vế phải là `const std::pair<_Ty1,_Ty2>` với `_Ty1 = int, _Ty2 = int` chưa được định nghĩa ...", thậm chí ngay cả khi ta định nghĩa toán tử << với vế phải là `pair<int,int>`, vẫn nhận được thông báo lỗi trên. Đây có thể là do lỗi của trình dịch hoặc thư viện STL trong Visual Studio.NET. Để hiển thị nội dung của một ánh xạ từ tập số nguyên sang tập số nguyên, ta làm một thủ thuật nhỏ: định nghĩa lại kiểu nguyên bằng lớp `CInt`. Ví dụ sau minh hoạ sẽ cho điều này:

```
#include "stdafx.h"
#include <map>
#include <iostream>
```

```
using namespace std;

template <typename _Ty1,typename _Ty2>
ostream& operator<<(ostream& out,const pair<_Ty1,_Ty2>& a_pair)
{
    out << "(" << a_pair.first << "," << a_pair.second << ")";
    return out;
}

class CInt
{
private:
    int _core;
public:
    CInt(int int_number):_core(int_number){};
    CInt(){};
    CInt(const CInt& other):_core(other._core){};
    friend ostream& operator<<(ostream& out,const CInt&
int_number)
    {
        out << int_number._core;
        return out;
    }
};

int main(int argc,char* argv[])
{
    typedef pair<int,int> int_int;
    pair<int,CInt> int_pair_array[10] =
{pair<int,CInt>(0,1),pair<int,CInt>(2,3),

    pair<int,CInt>(4,5),pair<int,CInt>(6,7),pair<int,CInt>(8,9)}
;
    map<int,CInt> int2int_map(int_pair_array, int_pair_array +
5);
    copy(int2int_map.begin(),int2int_map.end(),
        ostream_iterator<pair<int,CInt> >(cout," "));
    return 0;
}
```

Chi tiết hơn về lỗi này sẽ được trình bày trong phần lưu ý ở cuối chương.

Nói chung, rất ít khi người ta thực hiện thao tác duyệt toàn bộ map như các ví dụ trên. Các thao tác chủ yếu được sử dụng khi làm việc với map là tìm kiếm và truy nhập dữ liệu dựa trên khóa. Và cũng rất ít khi người ta sử dụng

map với khoá là kiểu số nguyên. Đối với những tập dữ liệu có nhu cầu đánh chỉ số trên tập số nguyên, ta nên sử dụng các bộ chứa tuần tự như vector hay deque, chúng được tối ưu hoá cho mục đích này. Thế mạnh của map chỉ nổi bật với các ứng dụng đánh chỉ số dựa trên một kiểu dữ liệu nào đó. Ta xét một ứng dụng cho phép người dùng nhập các lệnh vào dấu nhắc. Nếu lệnh này có trong bộ lệnh của chương trình, nó sẽ được thực hiện. Ngược lại, thông báo lỗi sẽ được đưa ra. Trong ứng dụng này, ta sẽ tạo ra một bộ lệnh mà thực chất là một ánh xạ giữa mã lệnh (là một chuỗi ký tự) với con trỏ lệnh.

```
#ifndef _COMMAND__H
#define _COMMAND__H

#include <vector>
#include <map>
#include <string>
#include <iostream>

using namespace std;

class CCommand
{
protected:
     string code;
     int argc;
     vector<string> argv;
public:
     CCommand();
     CCommand(string cmd_code, vector<string> cmd_argv);
     string GetCode();
     virtual int Execute(vector<string> cmd_argv =
vector<string>())=0;
     virtual ~CCommand();
};

class CCommandSet:public map<string,CCommand*>
{
     vector<string> current_cmd_argv;
     string current_cmd_code;
     void Extract(string cmd_str);
public:
     typedef map<string,CCommand*>::iterator iterator;
public:
     bool isRun;
```

```
    CCommandSet();
    void InsertCommand(CCommand* p_command);
    void RemoveCommand(string cmd_code);
    int Execute(string cmd_str);
    virtual ~CCommandSet();
};

#endif
```

Phần định nghĩa của hai lớp trên như sau:

```
#include "Command.h"

CCommand::CCommand(){}

CCommand::CCommand(string cmd_code, vector<string>
cmd_argv):code(cmd_code),argc(cmd_argv.size()){};

string CCommand::GetCode(){
    return code;
};

CCommand::~CCommand(){};

//======== CCommandSet definition ================

CCommandSet::CCommandSet(){
};

void CCommandSet::InsertCommand(CCommand* p_command){
    (*this)[p_command->GetCode()] = p_command;
};

void CCommandSet::RemoveCommand(string cmd_code)
{
    erase(cmd_code);
};

void CCommandSet::Extract(string cmd_str){
    current_cmd_argv.clear();
    string::iterator str_it = cmd_str.begin();
    while(str_it != cmd_str.end())
    {
        string word="";
        while((*str_it == ' ') && (str_it != cmd_str.end()))
```

```
                    str_it++;
            while((*str_it != ' ') && (str_it != cmd_str.end()))
                    word = word + *str_it++;
            current_cmd_argv.push_back(word);
    }
    current_cmd_code = current_cmd_argv[0];
    current_cmd_argv.erase(current_cmd_argv.begin());
};

int CCommandSet::Execute(string cmd_str){
    Extract(cmd_str);
    iterator cmd_it = find(current_cmd_code);
    if(cmd_it != end())
            return (cmd_it->second)->Execute(current_cmd_argv);
    cout << "Khong co lenh " << current_cmd_code << " trong tap
lenh\n";
    return -1;
};

CCommandSet::~CCommandSet(){
};
```

Đoạn mã trên sẽ được sử dụng cho bất kỳ chương trình có giao diện dòng lệnh nào. Tuy nhiên, nếu mới chỉ như vậy thì chương trình chưa làm được gì. Để xây dựng bộ lệnh cho ứng dụng của mình, ta phải viết các lớp lệnh cho từng tác vụ. Các lớp lệnh này được kế thừa từ lớp `Ccommand`, sau đó khai báo một đối tượng của lớp `CCommandSet` và lần lượt bổ sung các lệnh mong muốn cho tập lệnh của mình. Việc viết các lớp lệnh kế thừa từ lớp `CCommand` là điều bắt buộc vì `CCommand` là một lớp trừu tượng, các đối tượng không thể khai báo thuộc lớp này. Lưu ý rằng, bộ lệnh phải là một ánh xạ mã lệnh và con trỏ lệnh chứ không phải giữa mã lệnh và lệnh. Thiết kế này đảm bảo hai điều. Thứ nhất, không gây ra dư thừa trong lưu trữ các lệnh trong bộ nhớ, và điều quan trọng hơn là đảm bảo cho người dùng xây dựng bộ lệnh riêng cho mình một cách linh hoạt mà vẫn đảm bảo tính đúng đắn khi gọi hàm `Execute()` từ một con trỏ lệnh lớp `CCommand`. Lớp `CCommandSet` cung cấp các hàm cho phép người dùng xây dựng và sửa đổi tập lệnh của mình. Các hàm này bao gồm `InsertCommand()`, `RemoveCommand()` và thậm chí người dùng có thể sử dụng ngay các hàm hoặc toán tử `[]` của lớp `map` để thực hiện tác vụ này. Hàm quan trọng nhất của lớp `CCommandSet` là hàm `Execute()`. Hàm này nhận đối số là một dòng lệnh, dòng lệnh này sẽ được tách thành mã lệnh và tập tham số bằng hàm `Extract()`. Sau khi có được

mã lệnh (lưu trữ trong biến `current_cmd_code`) cùng tập tham số lệnh (lưu trữ trong biến `current_cmd_argv`), chương trình sẽ thực hiện việc xác định con trỏ lệnh dựa trên mã lệnh bằng hàm `find()` của lớp `map<>`. Nếu việc tìm kiếm có kết quả, tác vụ của lệnh tương ứng được thực hiện bằng cách gọi hàm `Execute()` của nó. Trong trường hợp ngược lại, thông báo lỗi sẽ được gửi tới người dùng.

Ví dụ sau sẽ tạo ra một bộ lệnh đơn giản với hai lệnh `print` và `quit`. Lệnh `print` sẽ đưa ra màn hình một thông báo `Yeu cau in danh sach da hoan thanh`. Lệnh `quit` sẽ thoát khỏi chương trình. Để thực hiện điều này, trước hết ta xây dựng hai lớp lệnh tương ứng với hai lệnh mà ứng dụng yêu cầu. Hai lớp này được kế thừa từ lớp `CCommand`. Sau đó, ta viết các hàm `Execute()` để thực hiện các tác vụ riêng cho từng lệnh.

```
#ifndef _APPCOMMAND__H
#define _APPCOMMAND__H

#include "../command.h"


class PrintStudentListCmd:public CCommand
{
public:
     PrintStudentListCmd();
     int Execute(vector<string> cmd_argv = vector<string>());
     virtual ~PrintStudentListCmd();
};

class QuitCmd:public CCommand
{
public:
     static bool isRun;
public:
     QuitCmd();
     int Execute(vector<string> cmd_argv = vector<string>());
     virtual ~QuitCmd();
};

#endif
```

Phần định nghĩa hai lớp lệnh được viết như sau:

```
#include "AppCommand.h"
```

```
//========= QuitCmd definition ===============

PrintStudentListCmd::PrintStudentListCmd():CCommand("print",vecto
r<string>()){
};

int PrintStudentListCmd::Execute(vector<string> cmd_argv){
    if(cmd_argv.size() > 0){
            cout << "Tham so khong hop le" << endl;
            return 1;
    }
    cout << "Yeu cau in danh sach da hoa thanh" << endl;
    return 0;
};

PrintStudentListCmd::~PrintStudentListCmd(){
};

//========= QuitCmd definition ===============

QuitCmd::QuitCmd():CCommand("quit",vector<string>()){
};

int QuitCmd::Execute(vector<string> cmd_argv){
    if(cmd_argv.size() > 0){
            cout << "Tham so khong hop le" << endl;
            return 1;
    }
    cout << "Tam biet" << endl;
    isRun = false;
    return 0;
};

QuitCmd::~QuitCmd(){
};
```

Sau khi có được các lớp lệnh, ta tạo ra các thực thể của các lớp lệnh này. Các thực thể này sẽ được bổ sung vào tập lệnh với vai trò là các lệnh. Chương trình thực sự bắt đầu với vòng lặp `while`, vòng lặp này sẽ kết thúc khi biến `QuitCmd::isRun` nhận giá `false`. Điều này chỉ xảy ra khi người dùng thực hiện lệnh `quit`. Chương trình sau sẽ được khai báo để sử dụng bộ lệnh với hai lệnh trên:

```
#include "AppCommand.h"
```

```
PrintStudentListCmd prn_cmd;
QuitCmd quit_cmd;
CCommandSet command_set;

bool QuitCmd::isRun = true;

void Initialize()
{
     command_set.InsertCommand(&prn_cmd);
     command_set.InsertCommand(&quit_cmd);

};

int main(int argc, char* argv[])
{
     Initialize();
     string command_str;
     vector<string> str_vec;
     while(QuitCmd::isRun)
     {
            cout << "Lenh>";
            cin >> command_str;
            command_set.Execute(command_str);
     }
     return 0;
}
```

Cũng giống như `set`, `map` cũng xuất phát từ KHÁI NIỆM Sorted Associative Container. Điều đó có nghĩa là các phần tử được sắp xếp dựa trên so sánh các khoá của mình. Trật tự sắp xếp phụ thuộc vào hàm so sánh được xác định bởi tham số thứ ba trong các tham số khuôn hình của `map<>`. Với cùng một tập phần tử, hàm so sánh khác nhau cho ta các trật tự khác nhau. Và với quy ước như vậy, hàm so sánh không đảm bảo tạo ra các phần tử với khoá từng đôi một khác nhau theo nghĩa thông thường. Ví dụ minh hoạ cho điều này có thể làm tương tự như ví dụ đã nêu trong phần trình bày về lớp `set`.

2.3.3 Lớp multiset và lớp multimap

Trong thực tế sử dụng, có rất nhiều trường hợp ta cần lưu trữ các mục dữ liệu mà khoá của chúng có thể trùng nhau. Ví dụ, khi lưu trữ một từ điển các từ đồng nghĩa, với một từ có thể có nhiều hơn một từ đồng nghĩa. Khi đó, ta có thể tổ chức bộ chứa lưu trữ từ điển như một ánh xạ từ một chuỗi ký tự sang một chuỗi ký tự. Ánh xạ này cho phép có nhiều hơn một mục có chung một khoá.

```
class SynonymDictionary
{
    multimap<string,string> _dictionary;
public:
    deque<string> GetSynonymWords(string a_word)
    {
            deque<string> result;
            result.clear();

    pair<multimap<string,string>::const_iterator,multimap<string
,string>::const_iterator> find_result =
_dictionary.equal_range(a_word);
            for(multimap<string,string>::const_iterator it =
find_result.first;
                    it != find_result.second;
                    it++)
                    result.push_back(it->second);
            return result;
    };
};
```

Ví dụ trên định nghĩa một lớp đại diện cho một từ điển các từ đồng nghĩa. Dữ liệu chính của lớp này là một đối tượng của `multimap<string,string>`. Mỗi phần tử trong biến này lưu trữ một từ gốc và từ đồng nghĩa của nó. Lớp từ điển cung cấp hàm thành phần `GetSynonymWords()` cho phép lấy ra các từ đồng nghĩa với một từ cho trước. Hàm thành phần này sử dụng một hàm đặc trưng của các bộ chứa liên kết phức (Multiple Associative Container) là `equal_range()`. Hàm `equal_range()` trả về một khoảng chứa các bộ có khoá trùng với đối số của hàm. Khoảng này được định nghĩa bởi một cặp bộ duyệt. Ngoài hàm này, các bộ chứa liên kết dạng này còn có hai hàm đặc trưng khác là `upper_bound()` và `lower_bound()`. Hàm `upper_bound()` và `lower_bound()` trả về hai cận tương ứng với hàm `equal_range()`

```
upper_bound() = equal_range().second
lower_bound() = equal_range().first
```

Xét về mặt ý nghĩa, nếu vị từ quan hệ dùng để sắp xếp các phần tử dựa trên khoá trong một bộ chứa liên kết là quan hệ nhỏ hơn thì hàm `lower_bound(a_key)` sẽ trả về vị trí của phần tử đầu tiên có khoá không nhỏ hơn (lớn hơn hoặc bằng) `a_key`. Còn `upper_bound(a_key)` trả về phần tử đầu tiên có khoá lớn hơn `a_key`. Ví dụ, thực hiện chương trình sau:

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

int main(int argc,char* argv[])
{
    multimap<int,int> int_m_map;
    for(int i = 0; i < 10;i++)
            int_m_map.insert(pair<int,int>(i % 3,i));
    multimap<int,int>::iterator it;
    int a_key = 1;
    cout << "Toan bo map:";
    for(it = int_m_map.begin();
            it != int_m_map.end();
            it++)
            cout << "(" << it->first << "," << it->second << ")
";
    cout << "\nCac phan tu co khoa nho hon "<< a_key << ":";
    for(it = int_m_map.begin();
            it != int_m_map.lower_bound(a_key);
            it++)
            cout << "(" << it->first << "," << it->second << ")
";
    cout << "\nCac phan tu co khoa bang "<< a_key << ":";
    for(it = int_m_map.equal_range(a_key).first;
            it != int_m_map.equal_range(a_key).second;
            it++)
            cout << "(" << it->first << "," << it->second << ")
";
    cout << "\nCac phan tu co khoa lon hon "<< a_key << ":";
    for(it = int_m_map.upper_bound(a_key);
            it != int_m_map.end();
            it++)
            cout << "(" << it->first << "," << it->second << ")
";
    cout << endl;
    return 0;
}
```

Kết quả chương trình:

```
Toan bo map:(0,0) (0,3) (0,6) (0,9) (1,1) (1,4) (1,7) (2,2)
(2,5) (2,8)
Cac phan tu co khoa nho hon 1:(0,0) (0,3) (0,6) (0,9)
Cac phan tu co khoa bang 1:(1,1) (1,4) (1,7)
Cac phan tu co khoa lon hon 1:(2,2) (2,5) (2,8)
```

Ví dụ trên gồm có 5 vòng lặp chính. Vòng lặp thứ nhất xây dựng nội dung cho bộ chứa `int_m_map`. Khoá của các phần tử trong bộ chứa là số dư trong phép chia của phần dữ liệu với 3. Vòng lặp thứ hai hiển thị toàn bộ nội dung của bộ chứa bằng việc duyệt qua toàn bộ bộ chứa. Vòng lặp thứ ba duyệt từ đầu bộ chứa tới phần tử đứng trước vị trí được trả về bởi hàm `lower_bound()`. Vòng lặp thứ tư duyệt qua khoảng được trả về bởi hàm `equal_range()`. Khoảng này chứa các phần tử có khoá bằng với `a_key`. Còn vòng lặp cuối cùng hiển thị các phần tử có khoá lớn `a_key`. Phép duyệt được thực hiện từ vị trí `upper_bound(a_key)` cho tới hết bộ chứa.

Để ý rằng, nếu `set` cho phép lưu trữ các phần tử trùng nhau thì xét về mặt ý nghĩa lưu trữ, nó không khác gì so với một bộ chứa tuần tự. Điều đó có nghĩa là nếu một tập các đối tượng được lưu trữ trong các bộ chứa tuần tự cũng có thể được lưu trữ bằng `multiset` và ngược lại (điều này hoàn toàn không đúng đối với `set`). Tuy nhiên, với cùng một tập các đối tượng, trật tự các đối tượng được lưu trữ trên bộ chứa tuần tự là do người dùng quy định, còn trật tự này trên `multiset<>` lại bị quy định trước bởi một tham số được ấn định khi khai báo.

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

using namespace std;
int main(int argc,char* argv[])
{
    vector<int> int_vector;
    for(int i = 0;i < 10;i++)
        int_vector.push_back(i % 3);
    multiset<int>
int_m_set(int_vector.begin(),int_vector.end());
    cout << "vector:";
    copy(int_vector.begin(),int_vector.end(),ostream_iterator<in
t>(cout," "));
    cout << "\nset:";
    copy(int_m_set.begin(),int_m_set.end(),ostream_iterator<int>
(cout," "));
    cout << endl;
    return 0;
}
```

```
vector:0 1 2 0 1 2 0 1 2 0
set:0 0 0 0 1 1 1 2 2 2
```

Do tổ chức lưu trữ khác nhau nên hai bộ chứa này cũng có khác biệt trên một số khía cạnh. Ví dụ, việc tìm kiếm trên `multiset<>` thường nhanh hơn do các phần tử đã được sắp xếp. Việc bổ sung một phần tử vào bộ chứa trên `multiset<>` lại thường lâu hơn khi thực hiện trên bộ chứa tuần tự do phải xác định vị trí đúng của phần tử mới. Chính vì thế, cũng xin được nhắc lại là việc lựa chọn bộ chứa nào cho chương trình là điều cần thiết. Điều này giúp chương trình hiệu quả hơn. Việc lựa chọn giữa `multiset<>` hay các bộ chứa tuần tự không khó do hai dạng bộ chứa này đã được thiết kế cho các mục đích rất khác nhau. Các bộ chứa liên kết luôn cần đến khoá. Do vậy, nó cho phép các tác vụ tìm kiếm thực hiện rất nhanh, Ngược lại, các bộ chứa tuần tự lại có nhiều lợi thế trong các thao tác cập nhật, sửa đổi. Đó mới chỉ là hai khía cạnh lớn được xét tới, bạn đọc có thể dựa trên những yêu cầu cụ thể để có được những phân tích sâu sắc hơn.

2.4 Một số bộ chứa hữu dụng khác

Trong phần này, chúng ta sẽ đề cập tới một số bộ chứa khác. Mặc dù, chúng không được xếp vào các lớp bộ chứa cơ sở nhưng vẫn được sử dụng khá phổ biến. Các bộ chứa được đề cập tới trong phần này bao gồm các bộ chứa thích nghi, lớp `string` và `bit_set`. Các bộ chứa thích nghi là các bộ chứa được xây dựng từ các bộ chứa cơ sở, chúng được bổ sung các ràng buộc hay phương thức để phù hợp với một số mục đích sử dụng cụ thể nhất định. Ta sẽ cùng tìm hiểu về ba bộ chứa thích nghi là `stack`, `queue` và `priority_queue`.

Hai lớp `string` và `bit_set` là các lớp cụ thể chứ không phải là các khuôn hình lớp như các bộ chứa đã giới thiệu trong các phần trước. Lớp `string` dùng để lưu trữ một chuỗi ký tự còn lớp `bit_set` đại diện cho một xâu nhị phân.

2.4.1 Các bộ chứa thích nghi

Các bộ thích nghi được hiểu là các lớp được xây dựng trên các lớp cơ bản. Khi đó, các lớp cơ bản sẽ được gọi là lớp nền. Lưu ý rằng, mối quan hệ giữa lớp bộ thích nghi và lớp nền không phải là sự kế thừa. Thông thường, các lớp nền bị thêm các ràng buộc để tạo ra lớp bộ thích nghi. Cơ chế này gọi là cơ chế thích nghi. Sự khác biệt giữa cơ chế thích nghi và cơ chế kế thừa còn thể hiện ở chỗ với cùng một lớp bộ thích nghi, người dùng có thể chọn lớp nền khác nhau mà không làm thay đổi giao diện của lớp bộ thích nghi.

Trong STL, các bộ thích nghi được phân vào 3 loại là: bộ thích nghi trên bộ chứa (gọi tắt là bộ chứa thích nghi), bộ thích nghi trên bộ duyệt (gọi tắt là bộ duyệt thích nghi) và bộ thích nghi trên đối tượng hàm (gọi tắt là đối tượng hàm thích nghi). Các lớp bộ chứa thích nghi sử dụng các lớp bộ chứa làm nền, các lớp này bao gồm `stack`, `queue` và `priority_queue`. Các thích nghi đối tượng hàm thích nghi và bộ duyệt thích nghi sẽ được bàn tới trong các chương sau.

Lớp stack

Lớp `stack` là một bộ chứa thích nghi mô tả lại cấu trúc dữ liệu ngăn xếp. Việc tổ chức dữ liệu cho `stack` có thể sử dụng `vector`, `deque` hay `list`. Lựa chọn bộ chứa cơ bản nào tuỳ thuộc vào bài toán cụ thể. Trong trường hợp người dùng không chỉ ra lớp bộ chứa cơ bản muốn sử dụng, bộ chứa cơ bản sẽ được lấy là `deque`. Để tạo ra một ngăn xếp với các phần tử là số nguyên, ta thực hiện theo ví dụ sau:

```
#include <stack>
#include <stdlib.h>

using namespace std;

int main(int argc, char* argv[])
{
     stack<int> int_stack;
     int_stack.push(rand());
     while(!int_stack.empty()&&((ticks = clock() - ticks) <
2000))      {
            int choice = rand() % 2;
            if(choice ==0)
                   int_stack.push(rand());
            else
            {
                   cout << int_stack.top() << " ";
                   int_stack.pop();
            }
     };
     cout << endl;
     return 0;

}
```

Ví dụ trên minh hoạ cho cách sử dụng `stack` với các thao tác chính như thêm vào một phần tử, lấy ra phần tử đầu. Ban đầu `stack` được khởi tạo

rỗng, sau đó một phần tử bất kỳ được thêm vào bằng lệnh:

```
int_stack.push(rand());
```

Sau đó, tại mỗi bước lặp, các thao tác thêm vào hoặc lấy ra một phần tử được chọn một cách ngẫu nhiên dựa trên giá trị biến `choice`. Vòng lặp kết thúc khi `stack` rỗng hoặc thời gian thực hiện vượt quá 2000 xung nhịp.

Ngoài các hàm đặc trưng là `pop()`, `push()` và `top()`, lớp `stack` trong STL còn hỗ trợ một số toán tử so sánh các `stack`. Các toán tử này bao gồm toán tử <, toán tử <=, toán tử ==, toán tử >=, toán tử > và toán tử !=. Các phép toán so sánh trên `stack` được định nghĩa như các phép toán trên xâu ký tự Lúc này mỗi `stack` đóng vai trò một xâu, mỗi phần tử trong `stack` tương ứng với mỗi ký tự trong xâu. Do vậy, để thực hiện các phép toán so sánh trên `stack`, các phép toán này cũng phải được định nghĩa cho lớp phần tử. Đối với các phép toán ,<, <=, >, >= , ta chỉ cần định nghĩa toán tử < cho lớp phần tử đủ đảm bảo chương trình có thể chạy được nhưng không đảm bảo chương trình đúng về mặt logic. Đối với hai toán tử != và ==, lớp phần tử phải định nghĩa toán tử == đủ đảm bảo cho tính đúng đắn của chương trình. Chú ý là các toán tử so sánh trên `stack` chỉ làm việc các lớp `stack` được thích nghi từ cùng một bộ chứa.

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

using namespace std;

int main(int argc, char* argv[])
{
     stack<int> stack_int_1;

     stack<student> stack_std_2;
     for(int i = 0;i < 5;i++)
     {
            stack_int_1.push(student(i+5);
            stack_int_2.push(student
     cout << (stack_int_1 != stack_int_2) << endl;
     return 0;

}
```

Thiết kế lớp `stack` của STL có một khác biệt nhỏ so với ngăn xếp thông thường. Đó là hàm `pop()` không trả về giá trị phần tử trên đầu ngăn xếp mà chỉ đơn thuần loại bỏ phần tử trên đầu ra khỏi ngăn xếp và không trả về giá trị nào. Để tham chiếu tới giá trị phần tử trên đầu ngăn xếp, lớp `stack` cung cấp hàm `top()`. STL tách tác vụ lấy phần tử trên đỉnh của ngăn xếp thành hai hàm riêng biệt mà không gộp lại thành một vì nếu `pop()` trả về phần tử đầu tiên trong ngăn xếp, nó phải trả về giá trị thay vì trả về tham chiếu tới phần tử đó, còn nếu trả về tham chiếu tới phần tử đầu thì nó sẽ tạo ra một con trỏ vu vơ do phần tử đầu bị tách ra khỏi ngăn xếp. Trong trường hợp trả về giá trị, hàm `pop()` sẽ không thực sự hiệu quả do phải gọi cấu tử sao chép. Do vậy, thiết kế tối ưu nhất là để hàm `pop()` chịu trách nhiệm tách phần tử đầu khỏi ngăn xếp mà không trả về bất cứ giá trị nào, còn `top()` trả về tham chiếu tới phần tử trên đỉnh mà không tách phần tử này ra khỏi ngăn xếp. Nguyên tắc này cũng được áp dụng cho các bộ chứa thích nghi khác.

Lớp `stack` không hỗ trợ thao tác duyệt bộ chứa cũng như truy nhập ngẫu nhiên. Điều này là hiển nhiên do các thao tác này không cần thiết đối với một ngăn xếp. Trong trường hợp vẫn muốn áp đặt các ràng buộc của ngăn xếp cho bộ chứa song lại cũng muốn cung cấp thao tác duyệt hoặc truy nhập ngẫu nhiên nhằm tăng tính linh hoạt cho bộ chứa, ta có thể viết một khuôn hình lớp ngăn xếp cho riêng mình. Lớp này kế thừa từ lớp `stack`. Trong lớp `stack` có một biến tên là `c` tham chiếu tới bộ chứa nền. Biến này được khai báo trong phạm vi `protected`, nghĩa là cho phép truy nhập bởi các hàm thành phần của lớp con. Khuôn hình của lớp ngăn xếp mới có thể viết như sau

```
#include <stack>

using namespace std;

template <typename _Ty,typename _C = deque<_Ty> >
class my_stack:public stack<_Ty,_C>
{
public:
     typedef _C::iterator iterator;
     iterator begin(){return c.begin();};
     iterator end(){return c.end();};

     _Ty & operator[](int index)

     {
            return c[index];
     }
};
```

Lớp `my_stack` hỗ trợ duyệt qua `stack` với bộ duyệt và truy nhập ngẫu nhiên. Tuy nhiên, thao tác truy nhập ngẫu nhiên chỉ hợp lệ với các lớp `my_stack` được thích nghi từ các lớp nền hỗ trợ truy nhập ngẫu nhiên như `vector` hay `deque`. Các bộ bộ chứa thích nghi có `list` là lớp nền không hỗ trợ toán tử `[]`. Do vậy, dùng toán tử `[]` với các đối tượng thuộc lớp này sẽ gây lỗi. Với các bổ sung này, bạn còn có thể hiển thị toàn bộ nội dung ngăn xếp ra màn hình với hàm `copy()` như thường làm với các bộ chứa trong các ví dụ trước:

```
#include <iostream>
#include <algorithm>
#include "MyStack.h"

int main(int argc, char* argv[])
{
    my_stack<int> int_my_stack;
    for(int i = 0;i < 10;i++)
    {
          int_my_stack.push(i);
    }
    cout << "Phan tu thu tu: " << int_my_stack[3] << endl;
    cout << "Toan bo ngan xep:" << endl;
    copy(int_my_stack.begin(),int_my_stack.end(),ostream_iterato
r<int>(cout," "));
    cout << endl;
    return 0;
}
```

Kết quả thu được trên màn hình như sau:

```
Phan tu thu tu: 3
Toan bo ngan xep:
0 1 2 3 4 5 6 7 8 9
```

Lớp queue

Lớp `queue` là một bộ chứa thích nghi mô phỏng hoạt động của một hàng đợi: các phần tử được thêm vào tại một đầu và được lấy ra ở đầu còn lại. Có thể thấy `queue` chỉ là một dạng đặc biệt của `deque`. Do vậy, chỗ nào có nhu cầu sử dụng `queue` thì đều có thể sử dụng `deque`. Sử dụng `deque` cho những công việc cần đến `queue` không những thoả mãn các yêu cầu của bài toán, mà nó còn cho phép sử dụng cả các tính năng của `deque`. Tuy nhiên,

đối với các bài toán mà tính chất FIFO (First In First Out) là quan trọng và chỉ cho phép thực hiện các thao tác của một hàng đợi trên bộ chứa thì queue được sử dụng thay cho deque.

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

using namespace std;

class PrintRequest
{
    int content;
    int print_time;
public:
    PrintRequest(){srand(time(0));content = rand() % 100;}
    PrintRequest(int cont):content(cont){};
    friend ostream& print(ostream& prn_des,const PrintRequest&
prn_req){
        prn_des << "Noi dung: " << prn_req.content << endl;
        return prn_des;
    }
};
```

```
#include "PrintRequest.h"
class PrintQueue:public queue<PrintRequest>
{
private:
    bool isActive;
public:
    PrintQueue():queue<PrintRequest>(),isActive(false){};
    void Active(){
        isActive = true;
        while (isActive && !empty())
        {
            print(cout,front());
            pop();
        }
    };
    void Deactive(){
        isActive = false;
    };
    void AcceptRequest(PrintRequest prn_req){
```

```
            push(prn_req);

    };
    void Clear(){
            while (!empty())
                    pop();
    };
};


int main(int argc, char* argv[])
{
    PrintQueue print_queue;
    srand(time(0));
    for(int i = 0;i < (rand() % 25);i++)
    {
            print_queue.AcceptRequest(PrintRequest(rand()));
    }
    print_queue.Active();
    return 0;
}
```

Cũng giống như `stack`, `queue` không có bộ duyệt và cũng không có cơ chế duyệt hay truy nhập tới một phần tử bất kỳ. Để bổ sung thêm các tính năng này, ta có thể viết lớp `my_queue` như sau :

```
#include <queue>

using namespace std;

template <typename _Ty,typename _C = deque<_Ty> >
class my_queue:public queue<_Ty,_C>
{
public:
    typedef _C::iterator iterator;
    iterator begin(){return c.begin();};
    iterator end(){return c.end();};
    _Ty & operator[](int index)
    {
            return c[index];
    }
};
```

Khi đó, ta có thể bổ sung chức năng in các yêu cầu in hiện tại trong hàng đợi. Trước tiên ta định nghĩa toán tử <<. Toán tử này thực hiện nhiệm vụ in thông tin về yêu cầu in ra một luồng ra nào đó.

```
friend ostream& operator<<(ostream& out,const PrintRequest&
prn_req)
{
    out << "Thoi gian thuc hien yeu cau " << print_time << "
giay";
} ;
```

Lớp `PrintQueue` được viết lại như sau:

```
#include "MyQueue.h"
#include "../queue01/PrintRequest.h"

class PrintQueue:public my_queue<PrintRequest>
{
private:
    bool isActive;
public:
    PrintQueue():my_queue<PrintRequest>(),isActive(false){};
    void Active(){
        isActive = true;
        while (isActive && !empty())
        {
            print(cout,front());
            pop();
        }
    };
    void Deactive(){
        isActive = false;
    };
    void AcceptRequest(PrintRequest prn_req){
        push(prn_req);

    };
    void Clear(){
        while (!empty())

            pop();
    };
    void ListAllRequest(ostream des){
        for(iterator it = begin();it != end();it++)
        {
```

```
                    des << *it << endl;
            }
    };
};
```

```
#include "PrintQueue.h"

int main(int argc, char* argv[])
{
    PrintQueue print_queue;
    srand(time(0));
    for(int i = 0;i < (rand() % 25);i++)
    {
            print_queue.AcceptRequest(PrintRequest(rand()));
    }
    print_queue.ListAllRequest(cout);
    return 0;
}
```

Việc kết xuất các thông tin về các yêu cầu in ấn không cần lấy các yêu cầu ra khỏi hàng đợi, do vậy ta vẫn có được kết quả (phần minh hoạ thực hiện yêu cầu in) giống như ví dụ trước khi bổ sung thêm dòng mã sau trước khi kết thúc chương trình.

```
print_queue.Active();
```

Lớp priority_queue

Lớp `priority_queue` cho phép lưu trữ các phần tử cùng kiểu theo trật tự được xác định bởi một hàm so sánh cho trước. Hàm so sánh được người sử dụng đưa vào để ấn định trật tự lưu trữ mong muốn, trong trường hợp người dùng không chỉ ra thì hàm mặc định được chỉ định. Hàm này là khuôn hình hàm `less<>`. Ví dụ sau chỉ ra cách lưu trữ các số nguyên theo trật tự từ lớn tới nhỏ.

```
#include <iostream>
#include <queue>
#include <cstdlib>
#include <ctime>
using namespace std;
```

```
int main() {
  priority_queue<int> int_priority_queue;
  srand(time(0)); // Tạo ra một số ngẫu nhiên
  for(int i = 0; i < 100; i++)
    int_priority_queue.push(rand() % 25);
  while(!int_priority_queue.empty()) {
    cout << int_priority_queue.top() << ' ';
    int_priority_queue.pop();
  }
}
```

Biến `int_priority_queue` sẽ lưu trữ 100 số nguyên từ 0 đến 24. Các số nguyên này được sinh ra một cách ngẫu nhiên và được bổ sung dần dần vào `int_priority_queue`. Với trật tự lưu trữ như trên, hàm `top()` sẽ luôn lấy ra số lớn nhất trong bộ chứa.

Trong ví dụ trên, `int_priority_queue` được khai báo là một `priority_queue` sử dụng tổ chức lưu trữ kiểu `vector` và có hàm xác định mức ưu tiên là `less<int>`. Đây là hai giá trị mặc định cho hai tham số của khuôn hình `priority_queue`. Khuôn dạng đầy đủ của khuôn hình `priority_queue` như sau:

```
priority_queue<typename Ty,typename _C = vector<Ty>, typename _Pr
= less<_C::value_type> >
```

với `_C` nhận một trong ba giá trị là `vector`, `deque` hoặc `list`, `_Pr` là một hàm hoặc một đối tượng hàm. Ví dụ sau sử dụng `priority_queue` để lưu trữ các từ theo thứ tự từ điển.

```
#include <iostream>
#include <queue>
#include <string>
#include <cstdlib>

using namespace std;
```

```
int main(int argc, char* argv[])
{
    char* line[] = {"acgd","afadf","gfsdfg","twert"};
    priority_queue<string,deque<string>,greater<string> >
str_priority_queue;
    string word;
```

```
    srand(0);
    for(int i = 0;i < 10;i++)
    {
        str_priority_queue.push(string(line[rand() % 4]));
    }
    while(!str_priority_queue.empty()) {
        cout << str_priority_queue.top() << ' ';
        str_priority_queue.pop();
    }
    return 0;
}
```

Ví dụ trên tạo ra một bộ chứa lưu trữ các từ theo thứ tự từ điển. Các từ được lấy ra ngẫu nhiên từ mảng `line`.

Cũng như hai bộ chứa thích nghi trên bộ chứa trên, `priority_queue` không có bộ duyệt. Tuy nhiên, trong trường hợp bộ chứa muốn xây dựng có yêu cầu này thì ta có thể mô phỏng lại các đặc trưng của `priority_queue` dựa trên một trong ba bộ chứa cơ sở là `vector`, `deque` hay `list`. Điều này đồng nghĩa với việc bạn phải viết lại các hàm `push()`, `pop()` và `top()`. Đồng thời, mỗi khi có một phần tử được thêm vào hay lấy ra, ta phải thực hiện thao tác vun đống trên bộ chứa cơ sở để đảm bảo trật tự của các phần tử trong hàng đợi ưu tiên. Thao tác vun đống được thực hiện bằng khuôn hình giải thuật `make_heap()`.

```
#include <iostream>
#include <vector>
#include <deque>
#include <queue>
#include <string>
#include <iterator>

using namespace std;

template<typename _Ty,typename _C = vector<_Ty>,typename _Pr =
less<_C::value_type> >
class my_priority_queue:public _C
{
private:
    _Pr comp;
public:
    const _Ty& top(){
        return front();
```

```
    };
    void push(const _Ty x){
            push_back(x);
            push_heap(begin(),end(),comp);
    };
    void pop(){
            pop_heap(begin(),end(),comp);
            pop_back();
    };

};

int main(int argc, char* argv[])
{
    my_priority_queue<string,deque<string>,greater<string> >
str_my_pq;
    typedef
my_priority_queue<string,deque<string>,greater<string>
>::iterator pq_iterator;
    string word;
    char* line[] = {"gfsdfg","afadf","twert","acgd"};
    greater<string> comp;
    for(i = 0;i < 4;i++)
    {
            str_my_pq.push(string(line[i]));
    }
    cout << "Duyet theo thu tu tuyen tinh: " << endl;
    for(pq_iterator it = str_my_pq.begin();it !=
str_my_pq.end();it++)
    {
            cout << *it << " ";
    }
    cout << endl;
    cout << "Duyet theo thu tu do hang doi co uu tien quy dinh:"
<< endl;
    while(!str_my_pq.empty())
    {
            cout << str_my_pq.top() << " ";
            str_my_pq.pop();
    }
    cout << endl;
    return 0;
}
```

Lưu ý rằng thứ tự duyệt khi sử dụng bộ duyệt không đảm bảo giống với trật tự của `priority_queue`.

Một cách khác để duyệt qua priority_queue là sử dụng bộ duyệt của bộ chứa nền của priority_queue như đã làm đối với các bộ chứa thích nghi trước. Trong ví dụ trên, bộ duyệt được sử dụng là deque<string>::iterator. Cũng như cách trên, thứ tự các phần tử khi duyệt tuần tự từ đầu đến cuối không đảm bảo theo đúng trật tự mong muốn của priority_queue. Đoạn mã khai báo khuôn hình my_priority_queue trong chương trình trên được thay thế bởi đoạn mã sau:

```
template<typename _Ty, typename _C = vector<_Ty>, typename _Pr =
less<_C::value_type> >
class my_priority_queue:public priority_queue<_Ty,_C,_Pr>{
public:
     typedef _C::iterator iterator;
     iterator begin(){return c.begin();};
     iterator end(){return c.end();};
};
```

2.4.2 Lớp string

Xử lý các chuỗi ký tự là nhu cầu trong hầu hết các ứng dụng. Để phục vụ nhu cầu này, thư viện STL cung cấp lớp string cho phép người dùng lưu trữ và thực hiện các thao tác trên chuỗi ký tự. Lớp string được cụ thể hoá từ khuôn hình lớp basic_string<> với kiểu phần tử là char. Tuy nhiên, chúng tôi sẽ không đề cập nhiều tới lớp basic_string<> mà sẽ trình bày chủ yếu trên lớp string để các bạn nhanh chóng quen với lớp string và không phải quá bận tâm tới một số KHÁI NIỆM khác liên quan tới basic_string<>

```
typedef basic_string<char, char_traits<char>, allocator<char> >
     string;
```

Với định nghĩa trên, có thể thấy string là một bộ chứa với kiểu phần tử là char. Do vậy, lớp string cũng có các hàm thành phần chung của bộ chứa mà cụ thể hơn là bộ chứa tuần tự cho phép truy nhập ngẫu nhiên như toán tử [], empty(), size()... Ngoài các hàm thành phần trên, lớp string có những hàm thành phần đặc trưng cho các thao tác trên chuỗi như tìm chuỗi con, kéo dài chuỗi ... Ví dụ sau minh hoạ việc tạo ra một chuỗi và một số thao tác đơn giản trên chuỗi đó.

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

using namespace std;

int main(int argc, char* argv[])
{
     string str_1("Hello");
     string str_2 = "World";
     str_1.append(" ");
     str_1.append(str_2);
     cout << str_1 << endl;
     return 0;
}
```

Lớp `string` cung cấp khá nhiều các cấu tử cũng như hàm gán cho phép người dùng khởi tạo các chuỗi ký tự rất linh hoạt. Các cấu tử, hàm gán và toán tử = cho phép khởi tạo giá trị cho một đối tượng `string` từ một chuỗi ký tự hoặc chỉ một phần của chuỗi ký tự, từ một mảng ký tự và thậm chí từ nội dung của một bộ chứa các ký tự.

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

using namespace std;

int main(int argc, char* argv[])
{
     vector<char> char_vector(49,'*');
     string str_1("High Performance Computing Center");
     string str_2(char_vector.begin(),char_vector.end());
     string str_3((char_vector.size() - str_1.size())/2,'*');
     cout << str_2 << endl << str_3 << str_1 << str_3 << endl <<
str_2 << endl;
}
```

Lưu ý rằng, cấu tử dùng để gán giá trị cho `str_2` là khuôn hình hàm thành phần nên không phải trình dịch nào cũng cho phép.

Cùng với các hàm cho phép ấn định nội dung của chuỗi từ các nguồn khác nhau, lớp string cũng cho phép chuyển nội dung về một kiểu lưu trữ chuỗi phổ biến là `char*`. Lớp `string` có hai hàm cho phép thực hiện việc này là `c_str()` và `data()`.

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

using namespace std;

int main(int argc, char* argv[])
{
    string str_1("High Performance Computing");
    str_1.push_back(char(NULL));
    str_1 = str_1 + "Center";
    cout << "Chuoi thu nhat: " << str_1 << " do dai: " <<
str_1.length() << endl;
    string str_2 ((char*)str_1.c_str());
    cout << "Chuoi thu hai: " << str_2 << " do dai: " <<
str_2.length() << endl;
    string str_3 = (char*)str_1.data();
    cout << "Chuoi thu ba: " << str_3 << " do dai: " <<
str_3.length() << endl;
    return 0;
}
```

```
Chuoi thu nhat: High Performance Computing Center do dai: 33
Chuoi thu hai: High Performance Computing do dai: 26
Chuoi thu ba: High Performance Computing do dai: 26
```

Hai hàm thành phần `c_str()` và `data()` được thiết kế phục vụ cho hai mục đích khác nhau, `c_str()` trả về vị trí đầu của một mảng các ký tự lưu trữ nội dung của chuỗi, mảng này được kết thúc bởi một ký tự NULL. Còn `data()` trả về vị trí đầu của một mảng các ký tự lưu trữ nội dung chuỗi, mảng này không nhất thiết phải kết thúc bằng ký tự NULL. Tuy nhiên, kết quả khi thực hiện chương trình trên với Visual Studio.NET hai chuỗi `str_2` và `str_3` không hề khác nhau bởi hàm `basic_string<>::data()` trong STL do Visual Studio.NET cung cấp chỉ đơn thuần là gọi lại `c_str()`:

```
const _Elem *data() const
{   // return pointer to nonmutable array
    return (c_str());
}
```

Lớp `string` cung cấp khá nhiều các hàm thành phần thực hiện các thao tác cơ bản trên chuỗi như tìm kiếm, thay thế, lấy chuỗi con, sao chép.... Mỗi một thao tác được hỗ trợ bằng một số hàm thành phần với các đối số khác nhau làm cho việc sử dụng đơn giản và tiện lợi. Chi tiết về các hàm có

thể tra cứu trong bất cứ một tài liệu trợ giúp nào của STL. Do vậy, chúng tôi sẽ không bàn thêm về các hàm này.

Ngoài ra, thư viện STL cung cấp một số phương thức cho phép đọc ghi chuỗi từ các luồng vào ra như toán tử >>, toán tử << và hàm `getline()`. Lưu ý rằng, các phương thức này không phải là các hàm thành phần của lớp `string`, nó là các hàm toàn cục. Toán tử `operator>>(basic_stream<> is, basic_string<> str)` sẽ ghi vào `str` nội dung của luồng `is`, toán tử << thực hiện công việc theo chiều ngược lại. Có một số lưu ý nhỏ khi làm việc với các toán tử >>. Việc đọc các ký tự từ luồng vào/ra và lưu vào một đối tượng kiểu `string` chỉ được thực hiện khi gặp một ký tự hợp lệ trong luồng. Đồng thời trong quá trình đọc, nếu gặp dấu cách các ký tự kết thúc dòng, xuống dòng hoặc khi đã đọc hết nội dung của luồng vào ra thì việc đọc bị dừng lại. Điều đó có nghĩa là toán tử >> không cho phép đọc các chuỗi có chứa dấu cách từ các luồng vào ra. Ví dụ sau minh hoạ cho điều này.

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

using namespace std;

nt main(int argc,char* argv[])
{
     string str1,str2;
     cout << "Enter a string:";
     cin >> str1;
     cout << "Content of string load from cin:" << str1 << endl;
     ifstream f;
     f.open("F:\\C-programs\\OOPtest\\swapping04\\test.txt");
     f >> str2;
     f.close();
     cout << "Content of string load from file test.txt:" << str2
<< endl;
     return 0;
}
```

```
Enter a string:Trung tam tinh toan hieu nang cao
Content of string load from cin:Trung
Content of string load from file test.txt:Trung
```

Nếu muốn lấy cả dòng, ta dùng hàm `getline()`. Hàm này sẽ đọc chép nội dung của luồng vào/ra vào đối tượng kiểu `string` cho tới khi gặp ký tự

kết thúc dòng hoặc kết thúc luồng vào/ra. Hàm `getline()` có hai phiên bản. Phiên bản thứ nhất mặc định ký tự kết thúc dòng là '\n', còn phiên bản thứ hai cho phép người dùng định nghĩa ký tự kết thúc dòng.

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

using namespace std;

int main(int argc,char* argv[])
{
    string str1,str2;
    cout << "Enter a string:";
    getline(cin,str1);
    cout << "Content of string load from cin:" << str1 << endl;
    ifstream f;
    f.open("F:\\C-programs\\OOPtest\\swapping04\\test.txt");
    getline(f,str2);
    f.close();
    cout << "Content of string load from file test.txt:" << str2
<< endl;
    return 0;
}
```

```
Enter a string:Trung tam tinh toan hieu nang cao
Content of string load from cin:Trung tam tinh toan hieu nang
cao
Content of string load from file test.txt:Trung tam tinh toan
hieu nang cao
```

2.4.3 Lớp bitset và bit_vector

STL cũng cung cấp cho ta một số lớp làm việc với các bit, chuỗi bit. Các lớp này đặc biệt hữu dụng khi người lập trình phải xây dựng các chương trình làm việc trực tiếp với phần cứng, với các cổng vào ra hoặc với các chương trình sử dụng khái niệm mặt nạ. Để minh hoạ việc sử dụng lớp `bitset`, chúng ta đưa ra đây một ví dụ nhỏ. Chương trình đơn giản này sẽ xây dựng một lớp đại diện cho một thanh ghi và một số thao tác đơn giản trên thanh ghi như các phép toán số học, phép toán logic và các thao tác dịch trái, dịch phải.

```
class Register
{
private:
    bitset<_length> _content;
    int _haft_length;
public:
    Register(void):_content(0),_haft_length(_length / 2)
    {
    }

    ~Register(void)
    {
    }

    friend ostream& operator<<(ostream& out,const Register&
reg);
    friend void MOV(Register& first_reg,const Register&
second_reg);
    friend void STOR(Register& reg,int number);
    friend void ADD(Register& first_reg,const Register&
second_reg);
    friend void SUB(Register& first_reg,const Register&
second_reg);
    friend void AND(Register& first_reg,const Register&
second_reg);
    friend void OR(Register& first_reg,const Register&
second_reg);
    friend void XOR(Register& first_reg,const Register&
second_reg);
    friend void NOT(Register& reg);
};
```

Lưu ý là nếu chỉ chép nguyên đoạn mã trên và dịch thì chắc chắn sẽ gặp lỗi LINK đối với các hàm nhưng chưa được định nghĩa, vì tất cả các thao tác đều được định nghĩa dưới dạng các hàm toàn cục và là hàm bạn của lớp `Register`. Tuy nhiên, do tránh sự thêm thắt rườm rà trong cuốn sách, ở đây chỉ đưa ra khuôn dạng của các hàm này. Phần định nghĩa các hàm được mô tả đầy đủ trong đĩa CD đi cùng cuốn sách.

Với một tập thanh ghi thuộc lớp `Register` và các phép toán được hỗ trợ, có thể viết chương trình mô phỏng hoạt động của một bộ vi xử lý đơn giản. Ví dụ sau mô phỏng lại hoạt động của bộ vi xử lý PIC16F84 dùng điều khiển các đèn LED. Trước hết, ta định nghĩa lớp `MicroChip` ở mức đơn giản nhất:

```
#include <map>
#include <vector>
#include "Register.h"

template<int bits> class MicroChipGenerator;

template<int bits>
class MicroChip
{
friend class MicroChipGenerator<bits>;
private:
    int _memory_size;
    string _name;
    vector<Register<8> > RAM;
public:
    map<string,Register<bits> > GR;
public:
    MicroChip(void)
    {
    }

    ~MicroChip(void)
    {
    }
};
```

Chúng ta sẽ giả lập một bộ vi xử lý theo khía cạnh đơn giản nhất. Nó sẽ gồm một tập các thanh ghi và một bộ nhớ. Tập các thanh ghi được biểu diễn bởi một đối tượng của lớp `map<>` là ánh xạ giữa tên thanh ghi và một đối tượng kiểu `Register`. Bộ nhớ được biểu diễn bởi một tập các thanh ghi 8 bit. Ngoài hai phần chính này, một bộ vi xử lý còn được lưu trữ một số thông tin như dung lượng bộ nhớ, tên ... Trên đây mới chỉ đưa ra định nghĩa cho lớp `MicroChip` ở mức đơn giản nhất. Có thể bổ sung thêm các thành phần khác để lớp này hữu dụng hơn.

Để định nghĩa một bộ vi xử lý cụ thể, ta xây dựng ra một lớp đóng vai trò là bộ sinh ra các bộ vi xử lý cụ thể (các lớp làm công việc này gọi là lớp sinh).Với lớp này, khi muốn định nghĩa một vi xử lý cụ thể thay vì khai báo một đối tượng kiểu `MicroChip` sau đó ấn định các thông số của nó một cách trực tiếp qua các hàm thành phần được `MicroChip` cung cấp, ta sẽ truyền các thông số của bộ vi xử lý cần định nghĩa cho lớp sinh này. Lớp sinh này được định nghĩa như sau:

```
template<int bits>
class MicroChipGenerator
{
public:
     static MicroChip<bits>* Generate(char* name,int
memory_size,vector<string> register_name)
     {
            MicroChip<bits>* result = new MicroChip<bits>;
            result->_memory_size = memory_size;
            result->RAM.reserve(memory_size / 8);
            result->_name = string(name);
            for(vector<string>::iterator it =
register_name.begin();
                  it != register_name.end();
                  it++)
             result->GR[*it] = Register<bits>();
            return result;
     }
};
```

Đoạn chương trình mô tả việc khai báo một bộ vi xử lý và một đoạn
lệnh lập trình để bộ vi xử lý điều khiển các đèn LED gắn với các chân dữ
của nó.

```
#include "MicroChip.h"
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
     char* str_array[2] = {"STATUS","DATA"};
     vector<char*> register_name(str_array,str_array+1);
     MicroChip<8>* PIC16F84 =
MicroChipGenerator<8>::Generate("PIC16F84",256,register_name);
     STOR(PIC16F84->GR["DATA"],"00000011");
     SF(PIC16F84->GR["STATUS"],7);
     return 0;
}
```

Đoạn mã lập trình cho bộ vi xử lý chỉ đơn giản là hai lệnh gán gi
cho các thanh ghi dùng chung của bộ vi xử lý. Sau khi thanh ghi DATA
gán giá trị nhị phân 00000011 (giá trị 3), bít cuối cùng của STATUS
thiết lập. Bit này báo cho bộ xử lý biết nội dung của thanh ghi DATA
ghi ra cổng. Kết quả là hai đèn LED cuối trong chuỗi 8 đèn sẽ sáng. Với

này, người đọc có thể thấy lợi thế khi sử dụng các lớp của thư viện STL trong lập trình với các xâu nhị phân.

Xét về mặt sử dụng, `vector<bool>` không khác các `vector<>`. Tuy nhiên, lớp này được cụ thể hoá nhằm tối ưu không gian lưu trữ. Với bất kỳ một lớp `vector` với kiểu phần tử khác `bool`, hệ thống cần cấp phát ít nhất một byte cho một phần tử, `vector<bool>` chỉ yêu cầu một bit cho một phần tử. Một lưu ý khi sử dụng lớp này là định nghĩa kiểu:

```
typedef vector<bool> bit_vector ;
```

sẽ không còn trong các phiên bản sau của STL. Do vậy, nếu khai báo với `bit_vector` bị báo lỗi bạn có thể khai báo trực tiếp với `vector<bool>`.

2.5 Tìm hiểu sâu hơn về các bộ chứa

2.5.1 Thao tác hoán đổi nội dung trên các lớp bộ chứa

Trong thư viện STL có một thiết kế dường như đi ngược với tiêu chí "xây dựng các hàm, giải thuật càng tổng quát càng tốt". Đây là trường hợp của khuôn hình giải thuật `swap()`. Mặc dù khuôn hình giải thuật tổng quát `swap<typename _Ty>` vẫn làm việc tốt trên các bộ chứa, nhưng tất cả các bộ chứa cơ bản đều có hàm `swap()` của riêng mình. Điều khác thường này bắt nguồn từ yếu tố tổ chức lưu trữ của từng bộ chứa. Với mỗi một cách tổ chức lưu trữ vật lý khác nhau, người ta đề xuất một hàm thành phần hoán vị khác nhau chứ không đơn thuần là sử dụng các phép gán như giải thuật tổng quát:

```
template<class _Ty> inline
    void swap(_Ty& _Left, _Ty& _Right)
    {       // exchange values stored at _Left and _Right
    _Ty _Tmp = _Left;
    _Left = _Right, _Right = _Tmp;
    }
```

Ví dụ, đối với `vector`, các phần tử được tổ chức lưu trữ trong một khối nhớ theo trật tự tuyến tính. Khi đó, nếu hai `vector` sử dụng cùng một cơ chế cấp phát bộ nhớ thì việc chuyển đổi chỉ đơn giản là hoán đổi thông tin điều khiển của hai `vector`. Trong trường hợp ngược lại, việc hoán đổi chỉ đơn giản là hoán vị hai con trỏ. Hàm thành phần `swap()` của lớp `vector`

được định nghĩa như sau:

```
void swap(_Myt& _Right)
     {     . // exchange contents with _Right
     if (this->_Alval == _Right._Alval)
     {      // same allocator, swap control information
            std::swap(_Myfirst, _Right._Myfirst);
            std::swap(_Mylast, _Right._Mylast);
            std::swap(_Myend, _Right._Myend);
            }
     else
     {      // different allocator, do multiple assigns
     _Myt _Ts = *this; *this = _Right, _Right = _Ts;
     }
}
```

Rõ ràng, nếu thực hiện theo giải thuật tổng quát, số lần gọi các cấu tử sao chép cũng như huỷ tử của lớp phần tử là rất lớn. Trên thực tế, không bao giờ hai `vector` được hoán vị bằng hàm `swap()` tổng quát. Nó luôn sử dụng hàm thành phần `swap()` của riêng mình, ngay cả khi ta sử dụng lời gọi tới hàm `swap` toàn cục như trong ví dụ sau:

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

using namespace std;

int main(int argc,char* argv[])
{
     int int_array[10] = {0,1,2,3,4,5,6,7,8,9};
     vector<int> int_vector_1(int_array, int_array + 4);
     vector<int> int_vector_2(int_array + 5,int_array + 9);
     cout << "Vector 1:";
     copy(int_vector_1.begin(),int_vector_1.end(),ostream_iterato
r<int>(cout," "));
     cout << "\nVector 2:";
     copy(int_vector_2.begin(),int_vector_2.end(),ostream_iterato
r<int>(cout," "));
     swap(int_vector_1,int_vector_2);
     cout << "\nVector 1:";
     copy(int_vector_1.begin(),int_vector_1.end(),ostream_iterato
r<int>(cout," "));
     cout << "\nVector 2:";
```

```
    copy(int_vector_2.begin(),int_vector_2.end(),ostream_iterato
r<int>(cout," "));
    return 0;
}
```

Có thể xem đây là một thủ thuật nhỏ để luôn đảm bảo tính tối ưu của chương trình. Thực chất lúc này khuôn hình hàm `swap<typename _Ty>` tổng quát đã được cụ thể cho lớp `vector` bằng hàm `swap()` được định nghĩa trong lớp `vector` như sau:

```
friend void swap(_Myt& _Left, _Myt& _Right)
{   // swap _Left and _Right deques
    _Left.swap(_Right);
}
```

Để kiểm chứng điều này, có thể sử dụng các chương trình gỡ rối (debug) để thực hiện từng dòng lệnh trong chương trình. Tư tưởng thiết kế trên được thực hiện trên tất cả các bộ chứa. Tuy nhiên, nếu như hàm `swap()` toàn cục được gọi để hoán vị hai đối tượng của lớp `set` hoặc `map` thì hàm được sử dụng lại là giải thuật khái quát chứ không phải là giải thuật được đưa ra cho riêng `set` hay `map`.

Nguyên nhân của điều này bắt nguồn từ cơ chế tự phát hiện tham số khuôn hình của chương trình dịch dựa trên các đối số của hàm. Khi gọi `swap(int_vector_1,int_vector_2)`, chương trình dịch sẽ xác định kiểu của `int_vector_1`, từ đó xác định được tham số `_Ty` trong khuôn hình hàm `swap<typename _Ty>` là kiểu `vector<int>` và do khuôn hình hàm này đã được cụ thể hoá với lớp `vector` nên hàm cụ thể được gọi. Trong trường hợp đối với `set`, khi chương trình dịch xác định tham số `_Ty` là lớp `set`, nó không tìm được hàm `swap()` nào cụ thể hoá cho lớp `set` do các hàm thành phần `swap()` cũng như hàm bạn `swap()` của `set` được định nghĩa trong lớp `_Tree <typename _Traits>`. Để hàm `swap()` tổng quát gọi đến hàm thành phần `swap()` đã được cụ thể ta phải chỉ tường minh tham số khuôn hình như ví dụ sau:

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

using namespace std;
```

```
int main(int argc,char* argv[])
{
    int int_array[10] = {0,1,2,3,4,5,6,7,8,9};
    set<int> int_set_1(int_array, int_array + 4);
    set<int> int_set_2(int_array + 5,int_array + 9);
    cout << "Tap 1:";
    copy(int_set_1.begin(),int_set_1.end(),ostream_iterator<int>
(cout," "));
    cout << "\nTap 2:";
    copy(int_set_2.begin(),int_set_2.end(),ostream_iterator<int>
(cout," "));
    swap<_Tree<_Tset_traits<int,less<int>,allocator<int>,false>
> >(int_set_1,int_set_2);
    cout << "\nTap 1:";
    copy(int_set_1.begin(),int_set_1.end(),ostream_iterator<int>
(cout," "));
    cout << "\nTap 2:";
    copy(int_set_2.begin(),int_set_2.end(),ostream_iterator<int>
(cout," "));
    return 0;
}
```

Nếu không muốn làm theo cách rườm rà trên, ta có thể gọi trực tiếp hàm thành phần `swap()` của lớp `set`. Ngoài ra, ta cũng có thể viết lại một lớp kế thừa từ lớp `set` và định nghĩa thêm một hàm bạn `swap()` của lớp này.

```
#ifndef _MY_SET__H
#define    _MY_SET__H

#include <set>

using namespace std;

template<class _Kty,
    class _Pr = less<_Kty>,
    class _Alloc = allocator<_Kty> >
class my_set:public set<_Kty,_Pr,_Alloc>
{
public:
    template<class _Iter>
    my_set(_Iter _First, _Iter _Last)
          : set<_Kty,_Pr,_Alloc>(_First,_Last){};

    friend void swap(my_set& _Left,my_set& _Right)
    {
```

```
            _Left.swap(_Right);
        }
};

#endif
```

Lưu ý rằng, ta có định nghĩa thêm cấu tử khởi tạo từ một bộ chứa khác (để cho tiện sử dụng trong chương trình). Cấu tử này không kế thừa được từ lớp `set`. Đoạn chương trình sau sử dụng lớp `my_set` vừa định nghĩa ở trên.

```
#include "MySet.h"
#include <algorithm>
#include <iostream>

int main(int argc,char* argv[])
{
    int int_array[10] = {0,1,2,3,4,5,6,7,8,9};
    my_set<int> int_set_1(int_array, int_array + 4);
    my_set<int>  int_set_2(int_array + 5,int_array + 9);
    cout << "Tap 1:";
    copy(int_set_1.begin(),int_set_1.end(),ostream_iterator<int>
(cout," "));
    cout << "\nTap 2:";
    copy(int_set_2.begin(),int_set_2.end(),ostream_iterator<int>
(cout," "));
    swap(int_set_1,int_set_2);
    cout << "\nTap 1:";
    copy(int_set_1.begin(),int_set_1.end(),ostream_iterator<int>
(cout," "));
    cout << "\nTap 2:";
    copy(int_set_2.begin(),int_set_2.end(),ostream_iterator<int>
(cout," "));
    return 0;
}
```

Điều này tương tự với `map`. Ví dụ sau minh hoạ cho điều này.

```
#include "stdafx.h"
#include <map>
#include <iostream>

using namespace std;

int main(int argc,char* argv[])
{
```

```
    typedef pair<int,int> int_int;
    pair<int,int> int_pair_array[10] =
{pair<int,int>(0,1),pair<int,int>(2,3),

    pair<int,int>(4,5),pair<int,int>(6,7),pair<int,int>(8,9)};
    map<int,int> int2int_map_1(int_pair_array, int_pair_array +
2);
    map<int,int> int2int_map_2(int_pair_array + 2,int_pair_array
+ 5);
    swap(int2int_map_1,int2int_map_2);
    swap<_Tree<_Tmap_traits<int,int,less<int>,allocator<pair<con
st int,int> >,false> > >
            (int2int_map_1,int2int_map_2);
    return 0;
}
```

2.5.2 Xây dựng các bộ chứa phức hợp dựa trên các bộ chứa cơ sở

Trong các phần trên của chương này, chúng ta đã làm quen với các bộ chứa cơ sở. Và trong hầu hết các ví dụ, chúng ta mới chỉ sử dụng các bộ chứa để chứa một tập các phần tử đơn. Tuy nhiên, trong thực tế sử dụng, nhiều khi phải lưu trữ một tập các đối tượng mà mỗi đối tượng là một tập các đối tượng khác, chẳng hạn như ma trận hay từ điển. Ma trận có thể xem là một vectơ hàng các vectơ cột hoặc một vectơ cột các vectơ hàng (vectơ ở đây là khái niệm toán học, không nên nhầm với lớp `vector` của STL). Từ điển được xem là một tập các tập mục từ, mỗi tập mục từ là tập các mục từ có cùng chung chữ cái đầu.

Trong phần này, ta xây dựng các bộ chứa phức hợp dựa trên các bộ chứa cơ sở được thư viện STL cung cấp. Ta có thể tạo ra một bộ chứa từ hai bộ chứa tuần tự, từ một bộ chứa tuần tự và một bộ chứa liên kết hoặc từ hai bộ chứa liên kết, thậm chí có thể xây dựng một bộ chứa trong đó mỗi phần tử của nó là một bộ chứa phức hợp. Chúng ta sẽ đề cập tới vấn đề này thông qua hai ví dụ đơn giản là ma trận và từ điển đồng nghĩa. Những ví dụ được đưa ra ở đây cũng chỉ hạn chế ở mức độ minh hoạ cho cách tổ chức lưu trữ và cơ chế vào ra trên bộ chứa phức hợp được xây dựng.

Xây dựng bộ chứa lưu trữ một ma trận

Như đã nói trong phần đầu, ma trận có thể được nhìn nhận theo hai cách: là vectơ hàng trong đó mỗi phần tử là một vectơ cột hoặc ngược lại. Với cách nhìn này, ta sẽ xây dựng bộ chứa lưu trữ ma trận từ hai bộ chứa tuần tự.

Cách thức lưu trữ này hoàn toàn trong suốt đối với người sử dụng ma trận. Cùng với việc trong suốt hoá cách thức lưu trữ của ma trận, ta phải cung cấp cho người dùng các cơ chế truy nhập tới các hàng, các cột và các phần tử.

```
#ifndef _MATRIX__H
#define _MATRIX__H

#include <deque>
#include <vector>

using namespace std;
enum MatrixKind {RowMatrix,ColumnMatrix};

template <typename _Ty,MatrixKind _kind = ColumnMatrix>
class Matrix
{
    deque<deque<_Ty> > _core;
    int _rows,_cols;
public:
    typedef deque<_Ty>::iterator cell_iterator;
    typedef deque<_Ty> one_dim_storage;
    typedef deque<deque<_Ty> >::iterator one_dim_iterator;
};
```

Trong định nghĩa trên, ta quy ước mỗi ma trận được lưu trữ bằng một hàng đợi hai chiều, mỗi phần tử của hàng đợi này lại là một hàng đợi. Từ tổ chức lưu trữ như vậy, ta đưa ra các định nghĩa về các cơ chế vào ra bao gồm bộ duyệt cho phép duyệt trên từng phần tử `cell_iterator`, bộ duyệt trên từng hàng hay từng cột (tuỳ thuộc vào ma trận được định nghĩa là ma trận hàng hay ma trận cột) `one_dim_iterator`. Phép giải tham chiếu trên bộ duyệt `cell_iterator` trả về giá trị của phần tử được bộ duyệt chỉ tới. Phép giải tham chiếu trên bộ duyệt `one_dim_iterator` trả về một hàng hoặc một cột mà bộ duyệt đang chỉ tới.

Cùng với việc định nghĩa các cơ chế vào/ra dựa trên ba bộ duyệt, ta sẽ định nghĩa các hàm thành phần thực hiện thao tác vào/ra. Các hàm thành phần vào ra bao gồm các hàm cho phép duyệt trên các phần tử của ma trận, tham chiếu tới một hàng hay cột của ma trận và tham chiếu tới một phần tử trong ma trận.

```
public:
    one_dim_iterator begin();
    one_dim_iterator end();
```

```
    one_dim_storage& operator[](int index);
    _Ty& operator()(int row,int column);
```

Để ý rằng, lớp ma trận trên không cung cấp các hàm thành phần phục vụ việc duyệt trên từng phần tử trong hàng hay cột. Các hàm thành phần này do bộ chứa kiểu `one_dim_storage` quy định. Các hàm thành phần phục vụ việc duyệt và truy nhập tới các phần tử của ma trận được định nghĩa như sau:

```
template<typename _Ty, MatrixOrientation _orient>
Matrix<_Ty,_orient>::one_dim_iterator
Matrix<_Ty,_orient>::begin()
{
    return _core.begin();
}

template<typename _Ty, MatrixOrientation _orient>
Matrix<_Ty,_orient>::one_dim_iterator Matrix<_Ty,_orient>::end()
{

    return _core.end();
}

template<typename _Ty, MatrixOrientation _orient>
Matrix<_Ty,_orient>::one_dim_storage&
Matrix<_Ty,_orient>::operator[](int index)
{
    return _core[index];
}

template<typename _Ty, MatrixOrientation _orient>
_Ty& Matrix<_Ty,_orient>::operator()(int row,int col)
{
    int one_dim_index,other_dim_index;
    if(_orient == RowMatrix)
    {
        one_dim_index = col;
        other_dim_index = row;
    }
    else
    {
        one_dim_index = row;
        other_dim_index = col;
    }
    return _core[other_dim_index][one_dim_index];
}
```

Với định nghĩa như trên, việc truy xuất vào hàng, cột hay phần tử có chỉ số vượt quá số hàng, số cột hiện thời của ma trận sẽ gây ra lỗi. Tiếp tới chúng ta sẽ cùng làm một số ví dụ đơn giản thao tác trên các đối tượng của lớp ma trận vừa được định nghĩa. Nhưng trước hết, ta cần định nghĩa một số cấu tử, huỷ tử và hàm thành phần nhằm thuận tiện hơn khi sử dụng.

```
template<typename _Ty, MatrixOrientation _orient>
Matrix<_Ty,_orient>::Matrix()
{
    _rows = 0;
    _cols = 0;
    _core.clear();
}

template<typename _Ty, MatrixOrientation _orient>
Matrix<_Ty,_orient>::Matrix(int rows,int cols)
{
    _rows = rows;
    _cols = cols;
    _core.clear();
    int one_dim_size,other_dim_size;
    if(_orient == RowMatrix)
    {
        one_dim_size = _cols;
        other_dim_size = _rows;
    }
    else
    {
        one_dim_size = _rows;
        other_dim_size = _cols;
    }
    for(int i = 0;i < other_dim_size;i++)
        _core.push_back(deque<_Ty>(one_dim_size));

}

template<typename _Ty, MatrixOrientation _orient>
Matrix<_Ty,_orient>::~Matrix()
{
}
```

Ta sẽ sử dụng ma trận vừa được xây dựng trong một ví dụ đơn giản, ví dụ thực hiện ấn định và hiện thị ma trận ra màn hình. Ma trận được tạo ra có

kích thước 10x10 và được ấn định các giá trị từ 0 đến 99. Hàm hiển thị ma trận được viết như một khuôn hình hàm.

```
#include <iostream>
#include "../matrix.h"

template<typename _Ty,MatrixOrientation _orient>
ostream& operator<<(ostream& out,Matrix<_Ty,_orient>& matrix)
{
    for (int row = 0;row < matrix.GetRows();row++)
    {
        for(int col = 0;col < matrix.GetCols();col++)
            out << matrix(row,col) << " ";
        out << endl;
    }
    return out;
}

int main(int argc, char* argv[])
{
    Matrix<int,RowMatrix> int_row_matrix(10,10);
    for(int i = 0;i < 10;i++)
        for(int j = 0;j < 10;j++)
            int_row_matrix(i,j) = i*10 + j;
    cout << int_row_matrix;
    return 0;
}
```

Xây dựng từ điển đồng nghĩa

Trong ví dụ thứ hai, minh hoạ cho việc xây dựng các bộ chứa phức hợp, chúng ta sẽ xây dựng một từ điển đồng nghĩa. Trong từ điển này, mỗi từ sẽ có một tập các từ đồng nghĩa với nó. Như vậy một từ điển đồng nghĩa thực chất là một ánh xạ giữa một từ với một tập các từ. Trong tập các từ đồng nghĩa với một từ không có hai từ nào trùng nhau. Dựa trên phân tích này, ta xây dựng lớp `SynonymDictionary` như sau:

```
class SynonymDictionary
{
    map<string,set<string> > _dictionary;
public:
    typedef pair<string,set<string> > word_entry;
    typedef set<string> synonym_words;
```

```
    typedef map<string,set<string> >::iterator
word_entry_iterator;
    typedef set<string>::iterator word_iterator;
public:
    SynonymDictionary(void);
    ~SynonymDictionary(void);
public:
    word_entry_iterator begin();
    word_entry_iterator end();
    synonym_words& operator[](string a_word);
    word_entry_iterator find(string a_word);
public:
    void AddSynonymWord(string original_word,string
synonym_word);
};
```

Trong phần định nghĩa lớp `SynonymDictionary` trên, ta chỉ đưa ra các khai báo về các cơ chế và các hàm thành phần truy nhập từ điển. Định nghĩa cụ thể của các hàm thành phần này chủ yếu dựa trên các hàm thành phần do lớp `map` và `set` cung cấp. Phần định nghĩa này bạn đọc có thể tham khảo trong đĩa CD đi cùng cuốn sách. Các cơ chế truy nhập từ điển bao gồm các bộ duyệt cho phép duyệt trên từng mục từ - `word_entry_iterator`, bộ duyệt trên từng từ của một mục từ - `word_iterator`. Ngoài ra, lớp cũng chứa hai định nghĩa kiểu khác nữa là `word_entry` và `synonym_words`. Mỗi `word_entry` gồm hai phần, phần đầu chứa từ gốc còn phần thứ hai chứa tập các từ đồng nghĩa. Phép giải tham chiếu trên bộ duyệt `word_entry_iterator` trả về một đối tượng thuộc kiểu `word_entry`. Kiểu `synonym_words` đại diện cho tập các từ đồng nghĩa, đối tượng thuộc kiểu này thường được dùng để lưu trữ kết quả của phép tham chiếu từ điển dựa trên một từ đã biết.

Với định nghĩa đơn giản như trên, ta có thể xây dựng một ứng dụng có sử dụng từ điển đồng nghĩa. Tuy nhiên, trên thực tế, các từ điển thường được tổ chức phức tạp hơn đôi chút. Từ điển sẽ được chia thành nhiều tập mục từ, mỗi tập mục từ tương ứng với một chữ cái, chữ cái này là chữ đầu của các từ gốc trong tập mục từ này. Lúc này ta có thể coi từ điển là một ánh xạ giữa một chữ cái và tập các mục từ và tập các mục từ lúc này tương đương với một từ điển của phiên bản đầu. Ta định nghĩa lớp đại diện cho tập các mục từ - `SetOfWordEntries` như sau:

```
class SetOfWordEntries
{
    map<string,set<string> > _dictionary;
public:
    typedef pair<string,set<string> > word_entry;
    typedef set<string> synonym_words;
    typedef map<string,set<string> >::iterator
word_entry_iterator;
    typedef set<string>::iterator word_iterator;
public:
    SetOfWordEntries(void);
    ~SetOfWordEntries(void);
public:
    word_entry_iterator begin();
    word_entry_iterator end();
    synonym_words& operator[](string a_word);
    word_entry_iterator find(string a_word);
public:
    void AddSynonymWord(string original_word,string
synonym_word);
};
```

còn lớp `SynonymDictionary` được khai báo như sau:

```
class SynonymDictionary
{
    map<char,SetOfWordEntries> _dictionary;
public:
    typedef map<char,SetOfWordEntries>::iterator
set_of_word_entries_iterator;

    typedef SetOfWordEntries::synonym_words synonym_words;
    typedef SetOfWordEntries::word_entry_iterator
word_entry_iterator;
public:
    SynonymDictionary(void);
    ~SynonymDictionary(void);
public:
    set_of_word_entries_iterator begin();
    set_of_word_entries_iterator end();
    synonym_words& operator[](string a_word);
    SetOfWordEntries& operator[](char first_char);
```

```
    word_entry_iterator find(string a_word);
public:
    void AddSynonymWord(string original_word,string
synonym_word);
};
```

So với định nghĩa trong phiên bản đầu, lớp `SynonymDictionary` không khác nhiều. Lớp này được bổ sung bộ duyệt trên tập các từ mục và toán tử `[]` cho phép tham chiếu tới một tập từ mục tương ứng với một chữ cái. Mặc dù xét về mặt giao diện, các hàm thành phần của lớp `SynonymDictionary` trong phiên bản thứ hai không khác so với phiên bản đầu, nhưng phần định nghĩa lại khác khá nhiều. Điều này là do `SynonymDictionary` trong phiên bản thứ nhất chỉ là bộ chứa phức hợp với hai mức, trong khi đó ở phiên bản thứ hai, nó là ba mức.

2.6 Tóm tắt

2.6.1 Ghi nhớ

Bộ chứa là một cấu trúc dữ liệu cho phép lưu trữ một tập các phần tử cùng kiểu. Các bộ chứa được xây dựng với những đặc trưng riêng và dành cho các mục đích khác nhau.

Các bộ chứa trong STL được xếp vào hai lớp chính là các bộ chứa tuần tự và các bộ chứa liên kết. Bộ chứa tuần tự lưu trữ và truy xuất các phần tử theo trật tự tuyến tính. Các bộ chứa tuần tự cơ bản trong STL bao gồm `vector`, `deque` và `list`. Bộ chứa liên kết truy xuất các phần tử dựa trên giá trị khoá. Thư viện STL có bốn bộ chứa liên kết cơ bản là `set`, `map`, `multiset` và `multimap`. Ngoài hai dạng bộ chứa cơ sở trên, STL còn cung cấp các bộ chứa khác dưới dạng các bộ chứa thích nghi, `string` và `bitset`.

Lựa chọn một bộ chứa thích hợp cho chương trình không phải đơn giản. Trước hết, phải xét tới khả năng hỗ trợ các tác vụ bài toán yêu cầu đối với từng dạng bộ chứa. Tiếp đến xét tới độ phức tạp tính toán trên phương diện thời gian. Độ phức tạp tính toán sẽ được đánh giá đối với từng thao tác cơ bản như bổ sung, sửa đổi, tìm kiếm. Tóm lại, việc lựa chọn bộ chứa cho chương trình đòi hỏi phải nắm rõ được đặc trưng của từng bộ chứa và yêu cầu bài

toán.

Việc chỉ sử dụng đơn thuần các bộ chưa cơ sở do STL cung cấp không phải lúc nào cũng thoả mãn yêu cầu bài toán. Trong những trường hợp như vậy, phải xây dựng một bộ chứa mới. Tuy nhiên, các bộ chứa mới được xây dựng dựa trên các bộ chứa cơ sở của STL. Các bộ chứa mới có được sinh ra dựa cơ chế kế thừa, thích nghi hoặc kết hợp từ các bộ chứa cơ sở.

2.6.2 Một số lưu ý khi lập trình

- Để sử dụng các lớp, hàm của STL bạn cần khai báo không gian tên std trước khi sử dụng, khai báo này thực hiện như sau
  - `using namespace std;`
  - Trong trường hợp không khai báo như vậy, ta phải chỉ tường minh không gian tên khi khai báo từng lớp, ví dụ:
    - `std::vector<int> int_vector;`
    - hay
    - `std::copy(..);`
- Sử dụng `iostream.h` có thể sẽ gây lỗi.
- Các chương trình ví dụ minh hoạ khái niệm `khuôn hình` hàm thành phần được viết trên môi trường lập trình Visual Studio.Net. Nếu bạn sử dụng các môi trường lập trình không hỗ trợ khái niệm này như Visual Studio, chương trình sẽ gặp lỗi.
- Việc ép kiểu từ iterator sang long ở ví dụ trang 49 có thể gây lỗi trên một số môi trường phát triển. Ví dụ nầy khi thực hiện trên Visual C++ không gây ra lỗi nhưng khi dịch ở Visual C. NET thì báo lỗi.
- Trong ví dụ `swapping04`, chương trình không thể sử dụng kiểu `map<int,int>`, nếu sử dụng kiểu này khi dịch chương trình sẽ nhận được một thông báo lỗi: "Toán tử << với vế phải là `const std::pair<_Ty1,_Ty2>` với `_Ty1 = int, _Ty2 = int` không được định nghĩa" mặc dù ta đã định nghĩa toán tử << với vế phải là một `pair<_Ty1, _Ty2>` như trong ví dụ hoặc thậm chí định nghĩa một cách cụ thể cho `pair<int,int>`. Để minh hoạ ví dụ cho việc hoán đổi trên `map`, chúng ta phải định nghĩa lại một lớp `CInt` với vai trò như kiểu `int`. Đây có thể do lỗi của trình dịch hoặc lỗi của bộ thư viện STL trong Visual Studio.NET

2.6.3 Bài tập

Bài tập 2.1:

Viết bổ sung một số thao tác cơ bản cho lớp ma trận vừa xây dựng. Các thao tác này bao gồm:

Các phép toán với số thực

Các phép toán với ma trận, trước khi thực hiện cần kiểm tra điều kiện thực hiện phép toán.

Chương 3

BỘ DUYỆT

Mục đích chương này:

- Làm quen với bộ duyệt
- Hướng dẫn sử dụng một số bộ duyệt tiện ích trong STL
- Tìm hiểu về tư tưởng thiết kế bộ duyệt nói chung và trong STL nói riêng
- Hướng dẫn cách xây dựng một bộ duyệt

3.1. Giới thiệu chung về bộ duyệt

Bộ duyệt là một khái niệm hoàn toàn mới trong thư viện STL nói riêng và trong kỹ thuật lập trình khái lược nói chung. Đối với một lập trình viên có kinh nghiệm, khi làm việc với STL, họ chỉ cần một ít thời gian để quen với việc sử dụng các lớp bộ duyệt được thư viện cung cấp. Tuy nhiên, để thực sự làm chủ các bộ duyệt và xây dựng các bộ duyệt cho riêng mình, đó không phải là điều đơn giản.

Thực chất, bộ duyệt không phải là khái niệm bắt nguồn từ thư viện STL. Nó xuất phát từ một tư tưởng thiết kế các hệ thống gọi là "Thiết kế mẫu". Bộ duyệt được đề xuất nhằm tách biệt giữa tổ chức lưu trữ dữ liệu (các bộ chứa) và các cơ chế xử lý dữ liệu (các giải thuật). Khởi điểm về ý tưởng bộ duyệt là hoàn toàn dễ hiểu. Ví dụ, khi ta muốn tìm phần tử "lớn nhất" trong một tập các đối tượng theo khía cạnh nào đó thì việc lưu trữ tập các đối tượng bằng `vector` hay `deque` vẫn không thay đổi thuật toán xác định phần tử lớn nhất. Hơn thế, thuật toán tìm phần tử lớn nhất cũng không cần quan tâm tới kiểu phần tử của bộ chứa là gì mà chỉ xem xét các phần tử có khả năng so sánh hay không. Nói cách khác, giải thuật tìm phần tử lớn nhất phải làm việc được trên một bộ chứa bất kỳ, kiểu phần tử bất kỳ (tất nhiên kiểu phần tử này phải hỗ trợ các phép toán so sánh lớn hơn). Đối với người dùng, lợi ích cụ thể nhất của bộ duyệt là việc không phải sửa lại mã lệnh khi thay đổi bộ chứa. Ví dụ:

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

```
using namespace std;

int main(int argc,char* argv[])
{
     int int_array[10] = {0,1,2,3,4,5,6,7,8,9};
     vector<int> set_of_int_num(int_array,int_array + 10);
     cout << "Phan tu lon nhat" <<
*(max_element(set_of_int_num.begin(),set_of_int_num.end())) <<
endl;
     return 0;
}
```

Đối với bài toán đơn giản trên, nếu muốn lưu trữ tập các số nguyên trong bằng deque, người dùng chỉ đơn giản thay đổi lại khai báo bộ chứa được sử dụng:

```
vector<int> set_of_int_num(int_array,int_array + 10);
```

và đương nhiên là cả khai báo

```
#include <deque>
```

Có hai điểm trong đoạn mã giúp bạn không phải sửa mã lệnh khi thay đổi bộ chứa. Trước hết đó là cơ chế sử dụng bộ duyệt để truy nhập các phần tử trong bộ chứa của các thuật toán và thứ hai là toán tử giải tham chiếu (*) được định nghĩa chung cho các bộ duyệt.

Xét về mặt lập trình, bộ duyệt là sự khái quát hoá khái niệm con trỏ. Nó cho phép truy nhập tới các phần tử trong bộ chứa mà không cần quan tâm tới cấu trúc bên trong của bộ chứa. Ví dụ khi muốn duyệt qua toàn bộ bộ chứa

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

using namespace std;

int main(int argc,char* argv)
{
     typedef vector<int> SeqOfInt;
     SeqOfInt seq_of_int;
     for(int i = 0;i < 10;i++)
           seq_of_int.push_back(rand() % 100);
     for(SeqOfInt::iterator it = seq_of_int.begin();
```

```
            it != seq_of_int.end();
            it++)
            cout << *it << " ";
    cout << endl;
    return 0;
}
```

Kết quả chương trình là không đổi nếu thay mã lệnh khai báo kiểu

```
typedef vector<int> SeqOfInt;
```

bằng

```
typedef deque<int> SeqOfInt;
```

hay

```
typedef list<int> SeqOfInt;
```

Các KHÁI NIỆM liên quan tới bộ duyệt cũng được xây dựng theo hướng làm mịn dần (tinh chỉnh dần). Các bộ duyệt càng phức tạp thì cung cấp càng nhiều phương thức và có càng nhiều ràng buộc. Theo cách nhìn này, STL đưa ra 6 KHÁI NIỆM liên quan tới bộ duyệt:

- Bộ duyệt đơn giản – Trivial Iterator
- Bộ duyệt nhập – Input Iterator
- Bộ duyệt xuất – Output Iterator
- Bộ duyệt đơn chiều – Forward Iterator
- Bộ duyệt hai chiều – Bidirctional Iterator
- Bộ duyệt truy nhập ngẫu nhiên – Random Access Iterator

Chi tiết về các KHÁI NIỆM liên quan tới bộ duyệt được trình bày trong phần phụ lục.

Các KHÁI NIỆM trên được cụ thể hoá bằng các lớp bộ duyệt. Các lớp bộ duyệt có thể chia vào hai nhóm: các lớp bộ duyệt cơ sở và các lớp bộ duyệt chuyên dụng hay các bộ duyệt thích nghi. Từ góc độ người sử dụng, ít khi ta sử dụng bộ duyệt độc lập mà hầu hết sử dụng các bộ duyệt gắn với một bộ chứa nào đó ngoại trừ một số trường hợp sử dụng các bộ duyệt trên các luồng vào/ra `ostream_iterator`, bộ duyệt để bổ sung phần tử `insert_iterator`. Tuy nhiên nếu nhìn kỹ hơn, các bộ duyệt đi cùng bộ chứa không phải được định nghĩa riêng cho từng bộ chứa mà nó chỉ là các lớp bộ duyệt cơ sở được

cụ thể hoá cho bộ chứa. Ta sẽ đề cập điều này một cách rõ ràng hơn trong cuối chương.

Với mong muốn giúp các bạn làm quen với bộ duyệt, phần đầu chương sẽ chỉ giới thiệu một số lớp bộ duyệt. Đây là các lớp bộ duyệt thường được sử dụng độc lập cho các mục đích khác nhau. Các lớp bộ duyệt chuyên dụng bao gồm:

- Các bộ duyệt phục vụ việc bổ sung phần tử vào bộ chứa: `insert_iterator`, `front_insert_iterator` và `back_insert_iterator`.
- Các bộ duyệt trên các luồng vào/ra: `istream_iterator`, `ostream_iterator`, `istreambuf_iterator` và `ostreambuf_iterator`.
- Bộ duyệt ngược: `reverse_iterator`. Bộ duyệt này cho phép duyệt một bộ chứa từ cuối trở lại đầu.
- Bộ duyệt `raw_storage_iterator`.

Đây thực chất là các bộ duyệt thích nghi. Nó được sử dụng chủ yếu với vai trò đối số cho các giải thuật mà kết quả trả về là một tập phần tử.

3.2. Các bộ duyệt bổ sung phần tử

Dựa theo tên gọi ta cũng có thể thấy các bộ duyệt này được sử dụng để bổ sung thêm một hoặc nhiều phần tử vào bộ chứa. Nhưng rõ ràng, bộ chứa nào cũng cung cấp các hàm thành phần cho phép bổ sung các phần tử. Ví dụ hai cách bổ sung phần tử như sau là tương đương:

```
#include <iterator>
#include <list>
#include <iostream>

using namespace std;

int main(int argc,char* argv[])
{
     list<int> int_list;
     int_list.push_front(3);
     front_insert_iterator<list<int> > ii(int_list);
     ii = 0;
     copy(int_list.begin(),int_list.end(),
ostream_iterator<int>(cout, " "));
    return 0;
}
```

Kết quả thu được

```
0 3
```

Đến đây, ta sẽ làm rõ hơn về nhận định ở cuối phần trước và cũng là để giải thích cho điều khó hiểu trên. Các bộ duyệt bổ sung không mấy khi được sử dụng để bổ sung phần tử trực tiếp như ví dụ trên. Rõ ràng là so với cách gọi hàm `push_front()`, dùng bộ duyệt bổ sung rườm rà hơn rất nhiều. Bộ duyệt bổ sung nói riêng và các bộ duyệt thích nghi nói chung chỉ được sử dụng với vai trò đối số cho các giải thuật. Điều này có thể lý giải như sau: bộ duyệt được đề xuất với vai trò là giao diện giữa các giải thuật và bộ chứa, do vậy, tất cả các giải thuật được thiết kế chỉ làm việc trên bộ duyệt mà không cần quan tâm gì tới bộ chứa. Tuy nhiên, có nhiều giải thuật cho kết quả là một tập phần tử, cụ thể hơn là trong quá trình thực hiện giải thuật lần lượt bổ sung phần tử vào bộ chứa lưu trữ kết quả trả về. Do vậy, các bộ duyệt bổ sung được sử dụng làm đối số cho các hàm này.

Các bộ duyệt bổ sung phần tử được sử dụng khi người dùng sử dụng các hàm với giá trị trả về là một bộ chứa, ví dụ như hàm `copy()`

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

using namespace std;

int main(int argc,char* argv[])
{
    list<int> int_list;
    int_list.push_front(3);
    int_list.push_front(2);
    vector<int> int_vector;
    copy(int_list.begin(),int_list.end(),
back_insert_iterator<vector<int> >(int_vector));
    cout << "list ban dau: ";
    copy(int_list.begin(),int_list.end(),
ostream_iterator<int>(cout," "));
    cout << "\nvector ket qua: ";
    copy(int_vector.begin(),int_vector.end(),
ostream_iterator<int>(cout," "));
    return 0;
}
```

```
list ban dau: 2 3
vector ket qua: 2 3
```

Ví dụ trên thực hiện việc sao chép nội dung của một `list` sang một `vector`. Để ý rằng, trong khuôn dạng của hàm `copy()`, đối số thứ ba chỉ cần yêu cầu là một bộ duyệt xuất, nghĩa là một bộ duyệt kiểu `vector<int>::iterator` có thể truyền vào làm đối số của hàm này, nhưng nếu ta thay dòng lệnh

```
copy(int_list.begin(),int_list.end(),back_insert_iterator
<vector<int> >(int_vector));
```

bằng hai lệnh

```
vector<int>::iterator it = int_vector.begin()
copy(int_list.begin(),int_list.end(),it);
```

thì chương trình dịch được nhưng khi chạy sẽ báo lỗi. Đây là lỗi truy nhập tới vùng nhớ chưa được cấp phát của `vector` do hàm `copy()` thực hiện việc dịch chuyển bộ duyệt được truyền vào trong quá trình thực hiện. Tuy nhiên, nếu ta có thực hiện việc cấp phát trước cho `vector` với hàm `reserve()` (lưu ý rằng, không phải bộ chứa nào cũng có khả năng cấp phát trước như `vector`) như sau

```
int_vector.reserve(int_list.size());
    vector<int>::iterator it = int_vector.begin();
    copy(int_list.begin(),int_list.end(),it);
```

thì kết quả vẫn sai. Kết quả thu được sẽ như sau:

```
list ban dau: 2 3
vector ket qua:
```

Điều này có nghĩa là `int_vector` không được bổ sung thêm phần tử nào. Với ví dụ trên, bạn đọc đã thấy được phần nào vai trò của các bộ duyệt bổ sung.

Ta lại quay trở lại với chủ đề chính là các bộ duyệt bổ sung. Trong phần này, ta bàn về ba bộ duyệt bổ sung là `front_insert_iterator<>`, `back_insert_iterator<>` và `insert_iterator<>`. Các bộ duyệt này tương ứng với các thao tác bổ sung vào đầu bộ chứa, bổ sung vào cuối bộ chứa và bổ sung vào một vị trí bất kỳ trên bộ chứa.

3.2.1. Bộ duyệt front_insert_iterator

Bộ duyệt `front_insert_iterator` được sử dụng để bổ sung các phần tử vào đầu của một bộ chứa. Ví dụ sau thực hiện việc sao chép nội dung của một `vector` vào một `deque`, đồng thời thứ tự các phần tử được đảo ngược lại.

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

using namespace std;

int main(int argc,char* argv[])
{
    vector<int> int_vector;
    for(int i = 0;i < 10;i++)
            int_vector.push_back(i);
    deque<int> int_deque;
    copy(int_vector.begin(),int_vector.end(),front_insert_iterat
or<deque<int> >(int_deque));
    cout << "vector ban dau: ";
    copy(int_vector.begin(),int_vector.end(),ostream_iterator<in
t>(cout," "));
    cout << "\ndeque ket qua: ";
    copy(int_deque.begin(),int_deque.end(),ostream_iterator<int>
(cout," "));
    return 0;
}
```

```
vector ban dau: 0 1 2 3 4 5 6 7 8 9
deque ket qua: 9 8 7 6 5 4 3 2 1 0
```

Việc đảo ngược nội dung của dãy số được sao chép chỉ đơn giản là do sử dụng bộ duyệt `front_insert_iterator<>`. Đối tượng `vector` ban đầu được duyệt theo chiều thuận trong khi các phần tử của `deque` lại luôn được thêm vào đầu dãy. Việc `deque_int` rỗng hay đã có phần tử không ảnh hưởng gì tới các phần tử được bổ sung với `front_insert_iterator<deque<int> >`. Giả sử, ta thêm vào `deque_int` phần tử -1 trước khi gọi hàm `copy()`:

```
int_deque.push_back(-1);
```

thì thu được kết quả như sau:

```
vector ban dau: 0 1 2 3 4 5 6 7 8 9
deque ket qua: 9 8 7 6 5 4 3 2 1 0 -1
```

Việc bổ sung vào bộ chứa được thực hiện với toán tử = của lớp `front_insert_iterator<>`. Thực ra, toán tử này không làm gì khác là gọi hàm `push_front()` của bộ chứa

```
front_insert_iterator<_Container>& operator=(
            typename _Container::const_reference _Val)
            {       // push value into container
            container->push_front(_Val);
            return (*this);
            }
```

Do vậy, để có thể sử dụng bộ duyệt này cho mục đích bổ sung phần tử vào bộ chứa, các bộ chứa phải hỗ trợ hàm thành phần `push_front()`. Trong bộ duyệt `front_insert_iterator<>`, ngoài trừ toán tử = và cấu tử là có ý nghĩa, còn các toán tử còn lại như `*`, `++` ... đều chỉ định nghĩa để tránh gây ra lỗi khi sử dụng. Thực chất, chúng không làm bất cứ một việc gì. Cũng xin nói thêm rằng, toán tử * của `front_insert_iterator` trả về một bộ duyệt chứ không trả về giá trị của phần tử nó đang trỏ tới. Nó được định nghĩa như sau:

```
front_insert_iterator<_Container>& operator*()
            {       // pretend to return designated value
            return (*this);
            }
```

Chính vì vậy, thực hiện các toán tử trên không làm ảnh hưởng gì với việc bổ sung phần tử, ví dụ:

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

using namespace std;

int main(int argc,char* argv[])
{
     vector<int> int_vector;
     back_insert_iterator<vector<int> > ii(int_vector);
     ii = 0;
     ii++;
     *ii = 1;
```

```
    ii++= 2;
    copy(int_vector.begin(),int_vector.end(),
ostream_iterator<int>(cout," "));
    return 0;
}
```

```
0 1 2
```

3.2.2. Bộ duyệt back_insert_iterator

Bộ duyệt `back_insert_iterator` dùng để thêm các phần tử vào cuối một bộ chứa. Cũng tương tự như bộ duyệt `front_insert_iterator`, bộ duyệt `back_insert_iterator` chỉ làm việc với các bộ chứa hỗ trợ hàm `push_back()`. Điều này có nghĩa là chỉ có thể sử dụng được với bộ chứa cơ sở là `vector`, `deque` hay `list`. Và đương nhiên, ta cũng có thể sử dụng bộ duyệt này với các bộ chứa do tự xây dựng nếu các bộ chứa này được định nghĩa hàm `push_back()`. Kết quả của việc bổ sung khi sử dụng bộ duyệt `back_insert_iterator` tuỳ thuộc vào việc định nghĩa hàm `push_back()`. Sau đây là một ví dụ đơn giản để minh hoạ cách sử dụng bộ duyệt này:

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

using namespace std;

int main(int argc,char* argv[])
{
    deque<int> int_deque;
    for(int i = 0;i < 10;i++)
        int_deque.push_back(i);
    vector<int> int_vector;
    copy(int_deque.begin(),int_deque.end(),back_insert_iterator<
vector<int> >(int_vector));
    cout << "Deque ban dau: ";
    copy(int_deque.begin(),int_deque.end(),ostream_iterator<int>
(cout," "));
    cout << "\nVector ket qua: ";
    copy(int_vector.begin(),int_vector.end(),ostream_iterator
<int>(cout," "));
    return 0;
}
```

```
deque ban dau: 0 1 2 3 4 5 6 7 8 9
vector ket qua: 0 1 2 3 4 5 6 7 8 9
```

Ví dụ trên thực hiện việc sao chép nội dung của một deque sang vector bằng hàm copy() và bộ duyệt back_insert_iterator. Trật tự của dãy số vẫn được giữ nguyên sau thao tác sao chép. Nếu muốn sao chép nội dung và đảo ngược trật tự các phần tử, ta làm thế nào? Rõ ràng, không thể sử dụng được front_insert_iterator vì lớp vector không có hàm push_front(). Hướng giải quyết của chúng ta là duyệt deque từ cuối lên đầu trong quá trình sao chép. Ta sẽ thay lời gọi hàm copy() trong đoạn mã trên như sau:

```
copy(int_deque.rbegin(),int_deque.rend(),
back_insert_iterator<vector<int> >(int_vector));
```

Kết quả thu được sẽ đúng như mong muốn của chúng ta:

```
deque ban dau: 0 1 2 3 4 5 6 7 8 9
vector ket qua: 9 8 7 6 5 4 3 2 1 0
```

Để chỉ cho hàm copy() biết rằng phải duyệt int_deque theo thứ tự ngược trong quá trình sao chép, ta sử dụng bộ duyệt ngược (reverse iterator) và chỉ ra hai cận của đoạn cần sao chép là int_deque.rbegin() (cuối của bộ chứa) và int_deque.rend() (đầu của bộ chứa). Chúng ta sẽ không nói sâu thêm về bộ duyệt ngược ở đây và sẽ dành nó cho phần trình bày về bộ duyệt ngược.

3.2.3. Bộ duyệt insert_iterator

Ngoài hai bộ duyệt bổ sung trên, STL còn cung cấp một bộ duyệt bổ sung nữa là insert_iterator. So với các bộ duyệt front_insert_iterator và back_insert_iterator chỉ cho bổ sung vào bộ chứa tại 2 đầu, bộ duyệt insert_iterator linh hoạt hơn. Nó cho phép người sử dụng xác định vị trí bổ sung phần tử. Hơn thế, insert_iterator còn làm việc được với tất cả bộ duyệt cơ sở mà STL cung cấp vì bộ duyệt này sử dụng hàm insert(iterator where) được hỗ trợ trong tất cả các bộ chứa. Việc sử dụng insert_iterator cũng không khác gì hai bộ duyệt bổ sung trên. Ta có thể làm theo cách sau:

```
#include <vector>
#include <deque>
```

```
#include <iostream>

using namespace std;

int main(int argc,char* argv[])
{
    deque<int> int_deque;
    for(int i = 0;i < 10;i++)
        int_deque.push_back(i);
    vector<int> int_vector;
    int_vector.push_back(-1); int_vector.push_back(-2);
    vector<int>::iterator ii = int_vector.begin();
    ii++;
    cout << "Truoc khi sao chep: " << *ii << endl;
    copy(int_deque.begin(),int_deque.end(),insert_iterator<vecto
r<int> >(int_vector,ii));
    cout << "Sau khi sao chep: " << *ii << endl;
    cout << "deque ban dau: ";
    copy(int_deque.begin(),int_deque.end(),ostream_iterator<int>
(cout," "));
    cout << "\nvector ket qua: ";
    copy(int_vector.begin(),int_vector.end(),ostream_iterator<in
t>(cout," "));
    return 0;
}
```

```
Truoc khi sao chep: -2
Sau khi sao chep: -572662307
deque ban dau: 0 1 2 3 4 5 6 7 8 9
vector ket qua: -1 0 1 2 3 4 5 6 7 8 9 -2
```

Có hai lưu ý khi làm việc với `insert_iterator`. Thứ nhất, trong cấu tử của lớp `insert_iterator`, ngoài việc chỉ ra muốn bổ sung các phần tử vào bộ chứa nào (đối số đầu tiên), ta còn phải chỉ ra vị trí được bổ sung phần tử. Thứ hai, bộ duyệt dùng để xác định vị trí bổ sung phần tử sẽ được tự động tăng lên mỗi khi một phần tử được thêm vào và sau khi thực hiện một thao tác bổ sung, bộ duyệt này không chỉ vào một phần tử nào trong bộ chứa nữa.

3.3. Các bộ duyệt trên các luồng vào/ra

Các luồng vào ra trong STL được chia thành 2 loại, mỗi loại phục vụ cho một mục đích khác nhau. Các bộ duyệt `istream_iterator` và `istreambuf_iterator` được sử khi muốn lấy dữ liệu từ các luồng vào và bộ đệm. Còn `ostream_iterator` và `ostreambuf_iterator` phục vụ việc đẩy dữ liệu ra các luồng ra và bộ đệm.

3.3.1. Bộ duyệt ostream_iterator

Ta đã khá quen với việc sử dụng bộ duyệt trên luồng ra và hàm copy() để đưa nội dung của một bộ chứa ra màn hình. Ta cũng sẽ thực hiện tương tự để có thể kết xuất nội dung của bộ chứa ra tệp tin. Việc này được thực hiện như sau:

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

using namespace std;

int main(int argc,char* argv[])
{
    vector<int> int_vector;
    for(int i = 0;i < 10;i++)
          int_vector.push_back(i);
    ofstream text_file("Output.dat");
    ostream_iterator<int> int_file_out(text_file," ");
    ostream_iterator<string> str_file_out(text_file);
    *str_file_out = "Noi dung cua vector:\n";
    copy(int_vector.begin(),int_vector.end(),int_file_out);
    text_file.close();
    return 0;
}
```

Nội dung của tệp tin Output.dat như sau:

```
Noi dung cua vector:
0 1 2 3 4 5 6 7 8 9
```

So với việc kết xuất nội dung bộ chứa ra màn hình, việc kết xuất nội dung bộ chứa ra tệp tin không khác nhau nhiều. Trước hết, ta khai báo một luồng ra tệp tin. Khai báo này được ví dụ trên thực hiện trong dòng:

```
ofstream text_file("Output.dat");
```

Khai báo này sẽ tạo ra một luồng vào ra tương ứng với tệp tin Output.dat. Sau đó, ta gắn bộ duyệt với luồng ra tệp tin vừa được tạo ra. Cuối cùng, ta gọi hàm copy() với đối số thứ ba chính là bộ duyệt này. Trong ví dụ trên, còn có một số kỹ thuật mới. Đầu tiên đó là việc gắn hai bộ duyệt với

kiểu phần tử khác nhau với cùng một luồng vào ra. Điều này cho phép bạn đẩy các dữ liệu được khai báo theo các kiểu khác nhau ra cùng một tệp tin. Ví dụ như ghi ra tệp tin nội dung của một `vector<int>`. Tiếp đến là nội dung của `deque<float>` v.v. Thậm chí có thể ghi nội dung của một bộ chứa với kiểu phần tử do người sử dụng tự định nghĩa. Trở lại với ví dụ ban đầu về bộ chứa lưu các thông tin về sinh viên, thay vì hiển thị nội dung của các sinh viên ra màn hình, ta sẽ lưu vào tệp tin StudentProfile.dat.

```
#include <vector>
#include <iterator>
#include <fstream>
#include <string>
#include "../Student.h"

using namespace std;

int main(int argc,char* argv[])
{
     vector <student> student_vector;
     for(int i = 0;i < 10;i++)
            student_vector.push_back(student());
     ofstream text_file("StudentProfile.dat");
     ostream_iterator<student> std_file_out(text_file,"\n");
     copy(student_vector.begin(),student_vector.end(),
std_file_out);
     text_file.close();
     return 0;
}
```

Lưu ý là phải định nghĩa toán tử >> với đối số thứ nhất có kiểu là `ostream` và đối số thứ hai có kiểu là lớp mà ta vừa định nghĩa. Điểm mới thứ hai trong ví dụ đề cập ở trên là sử dụng toán tử = để ghi nội dung của biến ra tệp tin. Lưu ý rằng, cũng giống như đối với các bộ duyệt bổ sung phần tử, toán tử giải tham chiếu (`operator*`), toán tử ++ của bộ duyệt này trả về một bộ duyệt và cụ thể hơn là trả về chính nó (`*this`). Do vậy, ta viết

```
*str_file_out = "Content of a vector:\n";
```

hay

```
str_file_out = "Content of a vector:\n";
```

kết quả không hề thay đổi, đồng thời thực hiện lệnh

```
str_file_out++;
```

sẽ không có ý nghĩa gì.

3.3.2. Bộ duyệt trên luồng vào istream_iterator

Giữa bộ duyệt trên luồng ra và bộ duyệt trên luồng vào có một sự khác biệt quan trọng. Sự khác biệt này bắt nguồn từ yếu tố: các luồng vào có điểm cuối, trong khi đó, các luồng ra thì không. Điều đó có nghĩa là luôn ghi thêm vào luồng ra, trong khi đó việc đọc từ luồng vào có những giới hạn nhất định. Tuy nhiên, trên các luồng vào không định nghĩa các hàm cho phép đưa bộ duyệt tới các vị trí đầu hay cuối như đối với các bộ chứa. Chính vì vậy, STL xác định điểm cuối của một luồng vào qua một bộ duyệt đặc biệt. Bộ duyệt này được khai báo mà không gán với bất cứ một luồng vào nào. Khai báo này như sau:

```
istream_iterator<T> end_of_istream;
```

Ví dụ sau đây thực hiện đọc các từ của một tệp tin văn bản Input.dat với nội dung chỉ gồm một dòng "High Performance Computing Center" vào một `vector<string>`, sau đó hiển thị nội dung của bộ chứa này.

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

using namespace std;

int main(int argc,char* argv[])
{
    ifstream text_file("Input.dat");
    istream_iterator<string> str_file_in(text_file),end_of_file;
    vector<string> str_vector;
    copy(str_file_in,end_of_file,back_insert_iterator<vector<str
ing> >(str_vector));
    text_file.close();
    copy(str_vector.begin(),str_vector.end(),ostream_iterator<st
ring>(cout,"\n"));
    return 0;
}
```

Kết quả chạy chương trình:

```
High
Performance
Computing
Center
```

Đối với các luồng vào trên tệp tin, bộ duyệt sẽ được gán giá trị `end_of_stream` khi đọc đến cuối tệp tin hoặc khi đọc được một ký tự không mong muốn. Còn đối với luồng vào chuẩn (bàn phím), bộ duyệt sẽ được gán giá trị `end_of_stream` khi bạn nhập vào một ký tự không mong muốn, đúng hơn là khi chương trình đọc được ở bộ đệm bàn phím một ký tự không mong muốn. Các ký tự không mong muốn ở đây được hiểu là các ký tự không thể chuyển về thành một phần tử do bộ duyệt trỏ tới. Ví như khi bạn muốn nhập vào một dãy số nguyên bằng một bộ duyệt `in_it` với `in_it` được khai báo như sau:

```
istream_iterator<int> in_it(cin)
```

thì `in_it` sẽ nhận giá trị `istream_iterator<int>()` khi bạn nhập vào một ký tự bất kỳ khác các ký tự 0, 1, ...9 và ký tự trắng (các khoảng trắng giữa các số được nhập vào đóng vai trò là phân cách giữa các số). Ví dụ, ta muốn nhập vào một dãy số nguyên với chiều dài tuỳ ý (không biết trước độ dài của chuỗi cần nhập) và người dùng sẽ nhập vào một ký tự bất kỳ để kết thúc việc nhập. Đoạn chương trình đó được thực hiện như sau:

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

using namespace std;

int main(int argc,char* argv[])
{
    cout << "Nhap vao mot day so!\n
(Viec nhap se ket thuc khi dua vao mot ky tu bat ky va nhan
Enter)\n";
    istream_iterator<int> in_it(cin);
    istream_iterator<int> end;
    vector<int> int_vector;
    copy(in_it,end,back_insert_iterator<vector<int>
>(int_vector));
```

```
    cout << "Chuoi so da nhap: " << endl;
    copy(int_vector.begin(),int_vector.end(),
ostream_iterator<int>(cout," "));
    return 0;
}
```

Kết quả của chương trình trên như sau:

```
Nhap vao mot day so!
(Viec nhap se ket thuc khi dua vao mot ky tu bat ky va nhan
Enter)
1 23   456
7890 q 12 34
Chuoi so da nhap:
1 23 456 7890 Press any key to continue
```

Trong kết quả hiện lên màn hình trên, phần sau dấu nhắc cho tới dòng "Chuoi so da nhap:" là phần nhập vào từ bàn phím. Rõ ràng, là chỉ có các số được nhập trước ký tự "q" mới được lưu vào biến `int_vector`. Cách nhập liệu trên linh hoạt và tự nhiên hơn so với việc yêu cầu người dùng đưa vào trước số lượng phần tử của dãy. Điều này cũng đúng khi luồng vào thực hiện trên tệp tin.

Khác với các bộ duyệt bổ sung phần tử cũng như bộ duyệt trên luồng ra, các toán tử ++ và toán tử * đều được định nghĩa cho những mục đích riêng. Toán tử ++ sẽ chuyển tới phần tử kế tiếp trong luồng, còn toán tử * sẽ trả về giá trị của phần tử hiện tại mà bộ duyệt trỏ tới. Cần chú ý rằng, sự khác biệt trên là là sự khác biệt giữa KHÁI NIỆM Input Iterator và KHÁI NIỆM Output Iterator. Rõ ràng, các bộ duyệt được mô hình hoá từ KHÁI NIỆM Output Iterator dùng để đẩy dữ liệu ra bộ chứa nên "phần tử" nó trỏ tới là chưa tồn tại. Do vậy, toán tử * hay ++ đều không được định nghĩa. Ngược lại, các bộ duyệt mô hình KHÁI NIỆM Input Iterator được sử dụng để đọc dữ liệu ra từ các bộ chứa, nên tại mỗi thời điểm, bộ duyệt đều chỉ tới một phần tử xác định. Do vậy, toán tử * được định nghĩa để lấy giá trị của phần tử và ++ để dịch bộ duyệt tới phần tử tiếp theo.

Đối với bộ duyệt trên luồng vào `istream_iterator<>`, toán tử ++ được định nghĩa lại theo mục đích sử dụng. Khi thực hiện toán tử ++, bộ duyệt sẽ sử dụng toán tử của luồng vào tương ứng để trích ra phần tử tiếp theo trong luồng và lưu trữ giá trị phần tử này trong một biến thành phần của lớp `istream_iterator<>`. Nếu đó là luồng vào từ tệp, phần tử tiếp theo trong tệp được trích ra. Nếu đó là luồng vào từ thiết bị vào chuẩn (bàn phím), phần tử kế tiếp trong bộ đệm của thiết bị chuẩn sẽ được trích ra. Trong trường hợp

bộ đệm này rỗng, chương trình sẽ đợi cho tới khi bạn nhập liệu vào. Do vậy, có thể nhập liệu liên tục mà vẫn không thể kết thúc với chương trình sau:

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

using namespace std;

int main(int argc,char* argv[])
{
     // Chương trình này bị lỗi,
     // nó sẽ chạy mãi không dừng
     istream_iterator<char> in_it(cin),end;
     vector<char> char_vec;
     while(in_it != end)
            char_vec.push_back(*++in_it);
     return 0;
}
```

3.3.3. Các bộ duyệt istreambuf_iterator và ostreambuf_iterator

Các bộ duyệt `istreambuf_iterator` và `ostreambuf_iterator` là các bộ duyệt trên vùng đệm của các luồng vào ra. Điều đó có nghĩa là nó cho phép trích các thông tin từ các luồng này dưới dạng thô nhất. Cụ thể hơn nó không có sự phân biệt giữa các phần tử trong luồng, không có khái niệm phân cách (các khoảng trắng) giữa các phần tử trong luồng. Chính vì vậy, mặc dù được định nghĩa dưới dạng khuôn hình hàm, nhưng các bộ duyệt này chỉ chấp nhận hai kiểu là `char` và `w_char` cho kiểu phần tử. Ví dụ sau minh hoạ cho sự khác biệt giữa `istream_iterator<char>` và `istreambuf_iterator<char>` khi được sử dụng để đọc một tệp tin văn bản Input.dat chỉ gồm một dòng là "Trung tâm tính toán hiệu năng cao".

```
#include <iterator>
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc,char* argv[])
{
```

```
    ifstream in_1("Input.dat");
    istreambuf_iterator<char> istb_it(in_1), istb_end;
    ostream_iterator<char> ost_it(cout);
    while(istb_it != istb_end)
          *ost_it++ = *istb_it++;
    ost_it = '\n';
    ifstream in_2("Input.dat");
    istream_iterator<char> ist_it(in_2), ist_end;
    while(ist_it != ist_end)
          *ost_it++ = *ist_it++;
    ost_it = '\n';
    return 0;
}
```

```
Trung tam tinh toan hieu nang cao
Trungtamtinhtoanhieunangcao
```

3.4. Một số bộ duyệt hữu dụng khác

3.4.1. Bộ duyệt ngược reverse_iterator

Bộ duyệt này được sử dụng khi người dùng có mong muốn duyệt ngược một bộ chứa. Tuy nhiên, trước hết chúng ta cùng đưa ra một ví dụ đơn giản để biết cách sử dụng các bộ duyệt ngược. Giả sử, ta cần kiểm tra một dãy số có phải là đối xứng hay không. Một cách đơn giản là cho hai biến trỏ duyệt dãy số theo hay chiều ngược nhau bắt đầu từ hai đầu của dãy số và tại mỗi bước ta kiểm tra xem hai phần tử được trở bởi biến trỏ. Nếu giá trị này bằng nhau thì tiếp tục tăng hai biến trỏ theo chiều duyệt của chúng và chuyển sang bước lặp tiếp theo. Quá trình duyệt kết thúc khi gặp hai phần tử không bằng nhau hoặc duyệt qua toàn bộ dãy (thực chất chỉ cần duyệt qua nửa dãy). Thủ tục kiểm tra này được thể hiện như sau:

```
template <typename BidirectionalIterator>
bool Symetric(BidirectionalIterator first,
                    BidirectionalIterator last)
{
    reverse_iterator<BidirectionalIterator> rfirst(last);
    bool found = false;
    for(;(first != last)&&!found;
          first++,rfirst++)
        found = (*first != *rfirst);
    return (!found);
}
```

Đoạn chương trình sau sử dụng này để kiểm tra tính đối xứng của dãy số lưu trong deque:

```
#include <deque>
#include <iostream>
#include <iterator>

int main(int argc,char* argv[])
{
     deque<int> int_deque;
     for(int i = 0;i < 10;i++)
     {
            int_deque.push_back(i);
            int_deque.push_front(i);
     }
     copy(int_deque.begin(),int_deque.end(),ostream_iterator<int>
(cout," "));
     if( Symetric(int_deque.begin(),int_deque.end()))
            cout << "\nDay so doi xung\n";
     else
            cout << "\nDay so bat doi xung\n";
     return 0;
}
```

```
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
Day so doi xung
```

Bây giờ để thấy được điều thú vị khi dùng bộ duyệt ngược, ta viết hàm `Symetric()` nhưng không dùng bộ duyệt ngược, có thể tham khảo cách sau:

```
template <typename BidirectionalIterator>
bool Symetric(BidirectionalIterator first,
                    BidirectionalIterator last)
{
     bool found = false;
     for(;(first != last)&&!found;
          first++,last--)
        found = (*first != *last);
     return (!found);
}
```

Xem ra có vẻ đúng và hiệu quả hơn cách trên do chỉ phải duyệt trên hai nửa dãy. Tuy nhiên, nếu ta chạy chương trình trên, kết quả sẽ không như mong muốn.

```
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
Day so bat doi xung
```

Nguyên nhân là do hàm thành phần `end()` trong các bộ chứa trả về một bộ duyệt không chỉ vào bất cứ một phần tử trong nào trong mảng. Nó chỉ mang ý nghĩa đại diện. Chính vì điều này, khi duyệt đến cuối bộ chứa chỉ cần kiểm tra bộ duyệt hiện tại có trùng với bộ duyệt do hàm thành phần `end()` trả về không. Và sau đây là một ví dụ khác để thấy sự hữu ích của bộ duyệt ngược. Ví dụ này yêu cầu ta hiển thị nội dung của một bộ chứa theo thứ tự ngược. Chắc chắn không thể sử dụng hàm `copy()` như vẫn thường làm. Hàm `copy()` sẽ sử dụng toán tử ++ để duyệt bộ chứa thay vì toán tử --. Vậy ta sẽ duyệt bộ chứa một cách thủ công. Theo suy nghĩ thông thường, ta thực hiện như sau (trong đoạn mã sau đã cố gắng tránh hiển thị nội dung của "phần tử" chỉ bởi `end()`):

```
#include <deque>
#include <iostream>

using namespace std;

int main(int argc,char* argv[])
{
    deque<int> int_deque;
    for(int i = 0;i < 10;i++)
        int_deque.push_back(i);
    deque<int>::iterator it = int_deque.end();
    for(it--;it != int_deque.begin();it--)
        cout << *it <<" ";
    cout << endl;
    return 0;
}
```

```
9 8 7 6 5 4 3 2 1
```

Kết quả không đúng như mong muốn. Phần tử đầu tiên của bộ chứa không được xem xét. Có thể nói, ta không có cách nào thay đổi tình hình, ngoại trừ làm theo một cách chắp vá là thêm dòng sau vào sau vòng lặp `for`:

```
cout << *it <<" ";
```

Tuy nhiên, đây không phải cách những người lập trình chuyên nghiệp giải quyết vấn đề. Ta sẽ sử dụng bộ duyệt ngược. Đoạn duyệt và hiển thị nội dung `int_deque` được sửa lại như sau:

```
reverse_iterator<deque<int>::iterator> r_it;
for(r_it =
reverse_iterator<deque<int>::iterator>(int_deque.end());
    r_it !=
reverse_iterator<deque<int>::iterator>(int_deque.begin());
    r_it++)
    cout << *r_it << " ";
```

```
9 8 7 6 5 4 3 2 1 0
```

Có thể sử dụng hàm `copy()` cho mục đích này. Hàm `copy()` được gọi như sau:

```
copy(reverse_iterator<deque<int>::iterator>(int_deque.end()),
reverse_iterator<deque<int>::iterator>(int_deque.begin()),
ostream_iterator<int>(cout," "));
```

Lưu ý rằng, để có thể sử dụng được các đoạn mã trên trong chương trình, phải thêm khai báo

```
#include <iterator>
```

vào đầu chương trình. Một cách khác để thực hiện việc trên khá đơn giản và đã được nhắc tới trong ví dụ về các bộ duyệt bổ sung phần tử:

```
copy(int_deque.rbegin(),int_deque.rend(),
ostream_iterator<int>(cout," "));
```

Dùng như vậy, nhưng thực chất phải hiểu là `deque<>::reverse_iterator` chẳng qua chỉ là định nghĩa cụ thể của `reverse_iterator<>` với tham số là `deque<int>::iterator`, còn `rbegin()` và `rend()` được định nghĩa đơn giản như sau:

```
reverse_iterator rbegin()
{
    return (reverse_iterator(end()));
}
```

và

```
reverse_iterator rend()
{
    return (reverse_iterator(begin()));
}
```

3.4.2. Con trỏ và bộ duyệt raw_storage_iterator

Sử dụng con trỏ như một bộ duyệt

Mảng là một bộ chứa đơn giản, quen thuộc nhất với hầu hết các lập trình viên. Việc khai báo và sử dụng mảng cũng khá đơn giản. Tuy nhiên, từ góc độ sử dụng STL, mảng lại là bộ chứa tuần tự phức tạp nhất. Nó được xếp vào lớp các bộ chứa cho phép truy nhập ngẫu nhiên. Bộ duyệt trên mảng là con trỏ. Con trỏ có các hàm thành phần của một bộ duyệt truy nhập ngẫu nhiên. Tuy nhiên, kiểu đơn vị tính khoảng cách giữa các phần tử chỉ là kiểu nguyên. Để minh hoạ cho việc sử dụng con trỏ mảng như bộ duyệt, cách trực quan nhất là truyền nó vào cho đối số của một khuôn hình giải thuật. Ở đây xin đưa ra một khuôn hình giải thuật tương đối phức tạp để minh chứng rằng, STL xem con trỏ như một bộ duyệt truy nhập ngẫu nhiên.

```
#include <iterator>
#include <vector>
#include <deque>
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main(int argc,char* argv[])
{
     vector<int> int_vector;
     for(int i = 0;i < 10;i++)
            int_vector.push_back(i);
     deque<int> int_deque;
     for(i = 0;i < 10;i++)
            int_deque.push_back(2*i);
    int* int_array = new int[int_vector.size()];
 transform(int_vector.begin(),int_vector.end(),
int_deque.begin(),int_array,plus<int>());
     cout << "Day thu nhat: ";
     copy(int_vector.begin(),int_vector.end(),
ostream_iterator<int>(cout," "));
     cout << "\nDay thu hai: ";
     copy(int_deque.begin(),int_deque.end(),
ostream_iterator<int>(cout," "));
     cout << "\nDay tong ";
     copy(int_array,int_array + 10, ostream_iterator<int>(cout,"
"));
```

```
    return 0;
}
```

```
Day thu nhat: 0 1 2 3 4 5 6 7 8 9
Day thu hai: 0 2 4 6 8 10 12 14 16 18
Day tong 0 3 6 9 12 15 18 21 24 27
```

Khuôn hình giải thuật được dùng để minh hoạ là khuôn hình giải thuật `transform()`. Một cách đơn giản, `transform()` thực hiện biến đổi một nội dung của một tập phần tử (truyền vào qua đối số thứ ba) dựa trên hai dãy đầu vào và một phép toán hai ngôi (tương ứng với hai tham số đầu và tham số cuối cùng). Chi tiết hơn về khuôn hình giải thuật này xem trong chương sau. Trong ví dụ trên, mảng `int_array` gồm các phần tử là tổng của hai phần tử ở cùng vị trí trên `int_vector` và `int_deque`. Sau khi thực hiện biến đổi, nội dung của ba dãy được in ra màn hình bằng ba lời gọi hàm `copy()`. Lưu ý rằng, trước khi thực hiện việc biến đổi trên mảng `int_array`, mảng này phải được cấp phát bộ nhớ đủ cho số phần tử sẽ được tạo ra (trong ví dụ là `int_vector.size()`).

Bộ duyệt raw_storage_iterator

Ngoài cách trên, khi muốn đẩy dữ liệu ra một vùng nhớ, STL còn dùng bộ duyệt `raw_storage_iterator`. Để thực hiện công việc như ví dụ trên với `raw_storage_iterator`, chỉ việc thay lời gọi hàm `transform()` như sau:

```
transform(int_vector.begin(),int_vector.end(),
int_deque.begin(),raw_storage_iterator<int*,int>(int_array),plus<
int>());
```

Lưu ý rằng, ngay cả khi sử dụng `raw_storage_iterator`, ta vẫn phải cấp phát trước bộ nhớ cho vùng nhớ để xuất dữ liệu ra.

Không nên nghĩ rằng, `raw_storage_iterator` chỉ đơn giản dùng để đẩy dữ liệu ra một mảng. Nó có thể dùng để đẩy dữ liệu vào một bộ chứa bất kỳ. Điều khác biệt so với các bộ duyệt bổ sung phần tử là nó không thêm các phần tử vào bộ chứa, mà chỉ thay đổi nội dung của các phần tử trong bộ nhớ hiện có. Đây là lý do vì sao luôn yêu cầu bộ chứa đã được cấp phát một dung lượng nhớ cần thiết. Chúng ta trở lại với hàm `copy()` quen thuộc để minh hoạ việc sử dụng `raw_storage_iterator` với một bộ chứa.

```
#include <iterator>
#include <vector>
#include <deque>
```

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

using namespace std;

int main(int argc,char* argv[])
{
     vector<int> int_vector;
     for(int i = 0;i < 10;i++)
            int_vector.push_back(i);
     deque<int> int_deque(int_vector.size());
     copy(int_vector.begin(),int_vector.end(),
raw_storage_iterator<deque<int>::iterator,int>(int_deque.begin())
);
     cout << "Day thu nhat: ";
     copy(int_vector.begin(),int_vector.end(),
ostream_iterator<int>(cout," "));
     cout << "\nDay thu hai: ";
     copy(int_deque.begin(),int_deque.end(),
ostream_iterator<int>(cout," "));
     return 0;
}
```

```
Day thu nhat: 0 1 2 3 4 5 6 7 8 9
Day thu hai: 0 1 2 3 4 5 6 7 8 9
```

Nhìn bề ngoài, việc dùng bộ duyệt này có vẻ không đơn giản. Song trên thực tế nó cũng không qua phức tạp. Ta sẽ bắt đầu tìm hiểu từ tham số khuôn hình của lớp này. Bộ duyệt này có hai tham số khuôn hình. Tham số thứ nhất là một bộ duyệt đơn chiều. Điều đó có nghĩa là bạn có thể truyền vào tham số này bộ duyệt của `vector`, `deque`, `list` hay thậm chí là của `set` (trong ví dụ là `deque<int>::iterator`). Tham số khuôn hình thứ hai lại tương đối đặc biệt. Nó không phải là kiểu của phần tử, có thể là một kiểu bất kỳ miễn là lớp phần tử có một cấu tử với đối số có kiểu là tham số khuôn hình này. Ví dụ.

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

using namespace std;

int main(int argc,char* argv[])
```

```
{
    vector<char> char_vector;
    for(char c = 'a';c < 'k';c++)
        char_vector.push_back(c);
    deque<int> int_deque(char_vector.size());
    copy(char_vector.begin(),char_vector.end(),
raw_storage_iterator<deque<int>::iterator,char>(int_deque.begin()
));
    cout << "Chuoi ky tu: ";
    copy(char_vector.begin(),char_vector.end(),
ostream_iterator<char>(cout," "));
    cout << "\nChuoi ma ASCII: ";
    copy(int_deque.begin(),int_deque.end(),
ostream_iterator<int>(cout," "));
    return 0;
}
```

```
Chuoi ky tu: a b c d e f g h i j
Chuoi ma ASCII: 97 98 99 100 101 102 103 104 105 106
```

Ví dụ trên thực hiện việc lưu mã ASCII của dãy các ký tự trong `char_vector` vào `int_deque`. Chương trình sẽ chạy tốt nếu lớp `int` có cấu tử với đối số kiểu `char`, nghĩa là các khai báo có dạng:

```
int int_num('a');
```

phải hợp lệ.

Để kết thúc phần nói về bộ duyệt này, ta sẽ nói về các cấu tử của lớp `raw_storage_iterator`. Trong ví dụ trên, đối số thứ 3 của hàm `copy()` là một bộ duyệt được sinh ra bởi lời gọi cấu tử của lớp `raw_storage_iterator`. Cấu tử này có một đối số là bộ duyệt thuộc kiểu tương ứng với tham số khuôn hình thứ nhất. Bộ duyệt này xác định vị trí bắt đầu thực hiện đẩy dữ liệu. Lưu ý rằng, không được truyền một bộ duyệt chỉ tới vị trí cuối cùng của bộ chứa cho đối số này. Ngoài ra phải đảm bảo rằng, dung lượng của bộ chứa bắt đầu từ vị trí này cho tới cuối đủ để chứa dữ liệu mà bộ duyệt `raw_storage_iterator` đẩy vào. Ngoài cấu tử này, `raw_storage_iterator` còn có cấu tử sao chép. Tuy nhiên, ta sẽ không bàn sâu thêm về cấu tử này.

3.5. Xây dựng các bộ duyệt đặc thù

Mục đích chính của bộ duyệt là cung cấp cơ chế truy nhập bộ chứa. Tuy nhiên nhu cầu cần xây dựng một bộ duyệt đặc thù lại thường không xuất phát

từ việc xây dựng các bộ chứa mới. Nguyên nhân sâu xa của điều này nằm trong thiết kế các bộ duyệt của STL.

3.5.1. Thiết kế của bộ duyệt STL

Trước khi nói về thiết kế bộ duyệt trong STL, phần đầu sẽ trình bày về một số khía cạnh trong thiết kế bộ duyệt. Các khía cạnh này được đề cập tới mang nặng tính lý thuyết thiết kế hệ thống. Do vậy, ta chỉ đề cập tới những khía cạnh quan trọng nhất.

Các hướng tiếp cận trong thiết kế bộ duyệt

Như đã nói trong phần đầu, bộ duyệt được đề xuất với vai trò là giao diện giữa các giải thuật và các bộ chứa. Do vậy, khi thiết kế bộ duyệt, phải quan tâm tới các vấn đề khi bộ duyệt làm việc với bộ chứa và các vấn đề khi bộ duyệt làm việc với giải thuật. Có hai thao tác cơ bản của bộ duyệt khi làm việc với bộ chứa đó là các thao tác duyệt (ví dụ toán tử ++, toán tử --, ...) các thao tác giải tham chiếu (toán tử *, toán tử [] ...). Để hỗ trợ cho những thao tác này, việc thiết kế có thể đi theo hai hướng: liên kết chặt và liên kết lỏng. Theo hướng thứ nhất, bộ duyệt lưu trữ thông tin về bộ chứa (mối liên hệ này có thể được thể hiện bằng con trỏ trỏ tới bộ chứa mà nó gắn với). Khi đó bạn có thể viết một đoạn mã như sau:

```
vector<int> vec;
vector<int>::iterator vec_it(vec);
for(vec_it.begin();!vec_it.is_end();vec_it++)
{
            // Thực hiện một công việc gì đó
}
```

Đoạn mã trên thực hiện việc duyệt trên bộ chứa kiểu `vector<int>` `vec` bằng bộ duyệt `vec_it`. Trước tiên, `vec_it` được khai báo và được gắn với bộ chứa `vec`. Việc duyệt qua toàn bộ bộ chứa được thực hiện bởi vòng `for`. Hàm thành phần `begin()` của bộ duyệt sẽ đưa nó về đầu bộ chứa được gắn với nó. Hàm `is_end()` kiểm tra xem bộ duyệt đã chỉ tới điểm cuối của bộ chứa chưa.

Đối với các bộ duyệt được thiết kế theo hướng kết lỏng, bộ duyệt chỉ biết thông tin về phần tử hiện tại trong bộ chứa mà nó trỏ tới. Với thiết kế này, bộ duyệt không cần lưu trữ thông tin về toàn bộ bộ chứa. Đây là thiết kế của các bộ duyệt trong STL. Cũng với công việc như trên, bộ duyệt theo thiết kế này cho phép người thực hiện như sau:

```
vector<int> vec;
vector<int>::iterator vec_it;
for(vec_it = vec.begin();vec_it != vec.end();vec_it++)
{
    // Thực hiện một việc gì đó
}
```

Tuy nhiên, để có thể duyệt được trên bộ chứa, bộ duyệt còn có cần thông tin về cách tổ chức cấp phát cho các phần tử trong bộ chứa. Việc cấp phát các phần tử lại là một KHÁI NIỆM chung nên không được xem là một thành phần của bộ chứa.

Để các giải thuật có thể truy nhập vào các bộ chứa mà không quan tâm tới bộ chứa loại gì, bộ duyệt được truyền vào như là đối số trong hầu hết các giải thuật. Để thực hiện được điều này đòi hỏi phải có một giao diện chung cho các loại bộ duyệt khác nhau. Có hai hướng để giải quyết vấn đề này. Theo cách thứ nhất sử dụng tính kế thừa và đa hình trong lập trình hướng đối tượng. Hướng thứ hai sử dụng khuôn hình hàm để định nghĩa các giải thuật. Với cách tiếp cận đầu, ta định nghĩa một lớp bộ duyệt cơ sở. Đây là một lớp trừu tượng đóng vai trò cung cấp giao diện cho các lớp bộ duyệt. Các lớp bộ duyệt cụ thể được kế thừa từ lớp này và cụ thể hoá các hàm thành phần ảo theo từng loại bộ chứa.

```
template <typename _Ty>
class BaseIterator
{
...
pubic:
    virtual void operator++() = 0;
...
};

template <typename _Ty>
class Iterator:public BaseIterator<_Ty>
{
...
pubic:
    void operator++();
...
}
```

Lúc này đối số của các khuôn hình giải thuật sẽ là con trỏ hoặc tham chiếu của kiểu bộ duyệt cơ sở, ví dụ hàm `min`:

```
template<typenam _Ty>
_Ty min(BaseIterator* first, BaseIterator* last)
```

hoặc

```
template<typenam _Ty>
_Ty min(BaseIterator& first, BaseIterator& last)
```

Thiết kế này đảm bảo bạn có thể truyền cho hàm `min` hai bộ duyệt của `vector`, hai bộ duyệt của `deque` hay hai bộ duyệt của một chứa nào đó, miễn là nó được kế thừa từ lớp `BaseIterator`. Điều đó có nghĩa là hàm `min` ở trên có thể làm việc trên bất cứ bộ duyệt nào. Tuy nhiên, thiết kế này có một yếu điểm là các hàm thành phần của các bộ duyệt mà các hàm có thể sử dụng trong quá trình triển khai giải thuật bị hạn chế trong giao diện mà lớp `BaseIterator` khai báo. Hơn thế, người dùng không có khả năng khắc phục hạn chế này. Điều này có nghĩa là khi định nghĩa một khuôn hình giải thuật mới, chỉ có thể xoay xở bằng các hàm thành phần mà `BaseIterator` cung cấp thậm chí việc định nghĩa một bộ duyệt mới cũng không giúp hơn gì.

STL đi theo hướng tiếp cận thứ hai. Các giải thuật được định nghĩa dưới dạng khuôn hình hàm với đối số khuôn hình là các bộ duyệt. Ví dụ `min` có thể được định nghĩa như sau:

```
template <class _InputIterator>
_InputIterator::value_type min(_InputIterator
first,_InputIterator last)
{
     _InputIterator::value_type min_value = *first;
     for(_InputIterator it = first,it != last;it++)
            if(min_value > *it)
                   min_value = *it;
     return min_value;
}
```

Với định nghĩa như vậy, người dùng có thể truyền vào hàm hai bộ duyệt có kiểu bất kỳ với điều kiện là kiểu này phải hỗ trợ một số yếu tố sau:

- Toán tử ++
- Toán tử =
- Toán tử !=
- Toán tử *
- Có định nghĩa kiểu `value_type`

Khi đó định nghĩa lớp bộ duyệt không cần một lớp cơ sở như thiết kế trên mà vẫn thoả mãn ràng buộc.

Thiết kế bộ duyệt của STL

Nói tóm lại, các bộ duyệt trong STL được thiết kế theo hai phương châm.

Thứ nhất, các bộ duyệt được định nghĩa chung và được cụ thể hoá cho từng bộ chứa (bộ duyệt có liên kết lỏng với bộ chứa), ví dụ bộ duyệt cho lớp `vector` được định nghĩa như sau:

```
template<class _Ty,class _Ax = allocator<_Ty> >
class vector: public _Vector_val<_Ty, _Ax>
{
...
typedef _Ptrit<value_type, difference_type, _Tptr,
            reference, _Tptr, reference> iterator;
...
}
```

Trong định nghĩa kiểu bộ duyệt cho lớp `vector` trên, các tham số truyền cho `_Ptrit` là các thông tin mô tả cách tổ chức cấp phát bộ nhớ cho một phần tử trong bộ duyệt. Các thông tin này được cụ thể hoá như bằng các định nghĩa kiểu của `vector` như `value_type`, `difference_type` .... Sự cụ thể hoá cũng có thể được thể hiện dưới hình thức khai báo lớp bộ duyệt bên trong bộ chứa và kế thừa nó từ lớp bộ duyệt mong muốn. Đây là trường hợp bộ duyệt của các lớp `deque`, `list`, `set` và `map`. Ví dụ đối với `deque`

```
template<class _Ty,      class _Ax = allocator<_Ty> >
class deque: public _Deque_val<_Ty, _Ax>
{
...
     class const_iterator;
     friend class const_iterator;
     class const_iterator
            : public _Ranit<_Ty, _Dift, _Ctptr,const_reference>
     {
     ...
     }

     class iterator;
     friend class iterator;
     class iterator
            : public const_iterator
```

```
    {
    ...
    }
...
};
```

Thứ hai, các bộ duyệt không cần kế thừa từ một lớp cơ sở để có thể truyền vào cho các khuôn hình giải thuật với vai trò đối số. Lưu ý rằng, mặc dù các lớp bộ duyệt trong STL cũng kế thừa từ một lớp chung hoặc từ một lớp dẫn xuất nào đó, nhưng thiết kế này không nhằm mục đích sử dụng tính đa hình vì hầu hết các lớp cơ sở chỉ là các cấu trúc (struct). Các giải thuật trong STL là các khuôn hình hàm với tham số khuôn hình là các bộ duyệt. Ví dụ hàm `copy()` trong STL có khuôn dạng như sau:

```
template<class _InIt,
class _OutIt> inline
_OutIt copy(_InIt _First, _InIt _Last, _OutIt _Dest)
{
    return (_Copy_opt(_First, _Last, _Dest, _Ptr_cat(_First,
_Dest)));
}
```

Nhu cầu xây dựng bộ duyệt mới

Bây giờ chúng ta sẽ giải thích nghịch lý được nói tới ngay trong phần đầu của phần này. Thứ nhất, khi xây dựng một bộ chứa mới, bạn có thể sử dụng bộ duyệt của lớp cơ sở nếu bộ chứa được xây dựng kế thừa từ các lớp bộ chứa trong STL. Trong trường hợp đây là một bộ chứa hoàn toàn mới, cũng có thể định nghĩa bộ duyệt cho bộ chứa bằng cách cụ thể hoá một lớp bộ duyệt cơ sở của STL dựa trên các thông tin về bộ chứa. Do vậy, có thể nói nhu cầu xây dựng bộ duyệt mới thường chỉ nảy sinh khi người dùng mong muốn bộ duyệt hỗ trợ chức năng mà các bộ duyệt cơ sở không có. Ví dụ khi ta xây dựng bộ duyệt trên ma trận, ta mong muốn bộ duyệt này có một hàm thành phần cho biết hàng và cột của phần tử nó đang trỏ tới.

3.5.2. Xây dựng các bộ duyệt mới

Trong các phần tiếp theo, chúng ta sẽ cùng nhau bàn tới cách thức xây dựng một bộ duyệt mới, cụ thể hơn là xây dựng bộ duyệt cho lớp ma trận ta có được nhắc ở cuối chương trước. Phương châm thiết kế là kế thừa tối đa

những gì STL đã làm được, nghĩa là bộ duyệt của chúng ta sẽ dùng các bộ duyệt cơ sở của STL làm cơ sở ban đầu.

Trước hết, chúng ta cùng tìm hiểu cơ chế xác định thông tin về bộ duyệt trong STL hết sức hữu ích cho việc xây dựng bộ duyệt

Lấy thông tin về bộ duyệt

Các thông tin về bộ duyệt bao gồm kiểu dữ liệu, kiểu của đơn vị đo sự khác biệt về vị trí và dạng bộ duyệt. Các cơ chế lấy thông tin về bộ duyệt được STL cung cấp theo hai dạng: dạng thứ nhất thông qua các hàm như `distance_type()`, `value_type()` và `_Iter_cat()`, dạng thứ hai thông qua các lớp `iterator_trait<>`. Thực chất, việc giữ lại các hàm `distance_type()` hay `value_type()` chỉ nhằm đảm bảo tính tương thích ngược với các hệ thống cũ. Hầu hết các giải thuật hiện nay đều sử dụng `iterator_trait<>` để lấy các thông tin về bộ duyệt. Ví dụ sau minh hoạ việc lấy thông tin về bộ duyệt khi thực hiện giải thuật xác định khoảng cách giữa hai phần tử:

```
template<class _InIt,class _Diff> inline
void _Distance(_InIt _First, _InIt _Last, _Diff& _Off)
{    // add to _Off distance between iterators
     _Distance2(_First, _Last, _Off, _Iter_cat(_First));
}
```

`_Distance()` là một hàm được sử dụng cho các dạng bộ duyệt khác nhau. Nó sẽ gọi hàm `_Distance2()` tương ứng với dạng bộ duyệt. Ví dụ hàm `_Distance2()` cho bộ duyệt nhập (xác định thông qua đối số cuối) được định nghĩa như sau:

```
template<class _InIt,class _Diff> inline
void _Distance2(_InIt _First, _InIt _Last, _Diff& _Off,
                    input_iterator_tag)
{    // add to _Off distance between input iterators
     for (; _First != _Last; ++_First)
            ++_Off;
     }
```

Hàm `_Distance()` gọi hàm `_Distance2()` và truyền thông tin về dạng bộ duyệt bằng giá trị trả về bởi `_Iter_cat(_First)`.

Lưu ý là trong số các thông tin về bộ duyệt, dạng bộ duyệt được định nghĩa bởi các cấu trúc đặc biệt gọi là `tag`. Tương ứng với bộ duyệt nhập, STL

định nghĩa lớp `input_iterator_tag`, bộ duyệt xuất tương ứng `output_iterator_tag`. Các lớp `tag` này chỉ đơn thuần là một lớp rỗng (không có hàm, biến thành phần hay bất cứ một định nghĩa bên trong nào), đại diện cho một dạng bộ duyệt. Nó được sử dụng làm kiểu cho giá trị trả về của hàm `_Iter_cat()` hoặc kiểu của `iterator_trait<>::category`.

Ta sẽ xây dựng bộ duyệt `_IndexIterator` theo hướng tiếp cận của bộ duyệt cho `vector`. Ta đi theo hướng này vì bộ duyệt này không chỉ dùng trong ma trận, mà có thể còn dùng cho các bộ duyệt khác. Quá trình xây dựng bộ duyệt mới của chúng ta có thể chia làm hai bước chính:

1. Bước thứ nhất: Xây dựng lớp cơ bản `_IndexIterator`
2. Bước thứ hai: Cụ thể hoá cho bộ duyệt của ma trận

Bộ duyệt `_IndexIterator` không những cho phép người dùng truy nhập ngẫu nhiên vào từng phần tử, mà còn cho phép xác định vị trí trong bộ chứa của phần tử được trỏ tới. Rõ ràng, bộ duyệt này phải được cụ thể hoá từ KHÁI NIỆM Random Access Iterator, nghĩa là nó phải hỗ trợ các phép toán của một bộ duyệt truy nhập ngẫu nhiên như toán tử `+(int)`, toán tử <, toán tử <=. Ngoài ra, bộ duyệt này phải hỗ trợ phép toán xác định vị trí trong bộ chứa của phần tử đang trỏ tới. Ta đặt tên hàm thành phần này là `index()`. Lớp này được định nghĩa như sau:

```
template <class _Ty,
     class _Diff,
     class _Pointer,
     class _Reference>
class _IndexIterator
{
public:
     typedef _IndexIterator<_Ty,_Diff,_Pointer,_Reference> _Myt;
public:
     // constructor
     _IndexIterator(void)
     {
     }

     _IndexIterator(_Pointer Ptr,_Diff index)
     :_current(Ptr),_current_index(index)
     {
     }

     _IndexIterator(_Diff index):_current_indes(index)
```

```
    {
    }

    _IndexIterator(_Pointer Ptr)
    : _current(Ptr),_current_indez(-1)
    {
    }

    _IndexIterator(const _Myt& _iterator)
    :_current(_iterator._current),
    _current_index(_iterator._current_index)
    {
    }
...
    _Diff index()
    {
            return _current_index;
    }

    _Pointer operator->() const
    {
            return (_current);
    }

    bool operator!=(const _Myt& _Right) const
    {
            return (!(*this == _Right));
    }

    _Myt& operator+=(_Diff _Off)
    {
            _current += _Off;
            _current_index += _Off;
            return (*this);
    }
```

```
    _Myt operator+(_Diff _Off) const
    {
            return (_Myt(_current + _Off,_curent_index + _Off));
    }

...
    ~_IndexIterator(void)
    {
```

```
    }
protected:
    _Diff _current_index;
    _Pointer _current;
};
```

Đoạn chương trình trên mới chỉ đưa ra một số cấu tử và hàm thành phần đặc trưng của bộ duyệt. Chi tiết thêm về bộ duyệt này, các bạn tham khảo trong đĩa CD đi cùng sách. Trong định nghĩa của lớp này, chúng ta có một số quy ước. Khi một bộ duyệt chưa xác định được vị trí thì chỉ số của nó bằng -1, các bộ duyệt trỏ vào cuối một bộ chứa thuộc dạng các bộ duyệt này. Việc so sánh khác, nhỏ hơn hay lớn hơn không phụ thuộc vào chỉ số, điều này cũng là dễ hiểu khi ta quy ước chỉ số của con trỏ cuối bằng -1.

Tiếp đến, trong bước hai, ta định nghĩa bộ duyệt cho lớp ma trận được xây dựng trong ví dụ ở cuối chương trước. Lớp ma trận này gồm hai bộ duyệt. Bộ duyệt theo hàng (cột) và bộ duyệt trên từng phần tử. Để có thể định nghĩa các bộ duyệt của ma trận là các bộ duyệt chỉ số, ta đưa vào một số thay đổi nhỏ trong định nghĩa của lớp này. Trước hết, ta định nghĩa một lớp đại diện cho hàng (cột) của ma trận (trong ví dụ của chương trước lớp này không được định nghĩa cụ thể mà coi nó là một `deque` luôn). Lớp này có tên là `OneDimStorage<>`:

```
template<typename _Ty>
class OneDimStorage
{
public:
    typedef deque<_Ty>::difference_type difference_type;
    typedef deque<_Ty>::size_type size_type;
    typedef deque<_Ty>::value_type value_type;
    typedef deque<_Ty>::reference reference;
    typedef deque<_Ty>::const_reference const_reference;
    typedef _IndexIterator<_Ty,
            difference_type,
            deque<_Ty>::iterator,
            reference> iterator;
    typedef _IndexIterator<_Ty,
            difference_type,
            deque<_Ty>::const_iterator,
            const_reference> const_iterator;
protected:
    deque<_Ty> _core;
```

```
public:
    OneDimStorage()
    {
    }
    OneDimStorage(size_type dim_size):_core(dim_size)
    {
    }

    iterator begin()
    {
        return iterator(_core.begin(),0);
    }
    iterator begin() const
    {
        return iterator(_core.begin(),0);
    }
    iterator end()
    {
        return iterator(_core.end(),-1);
    }
    iterator end() const
    {
        return iterator(_core.end(),-1);
    }
    reference operator[](difference_type index)
    {
        return _core[index];
    }
};
```

Điều đáng chú ý trong định nghĩa của lớp `OneDimStorage` là định nghĩa về bộ duyệt của lớp. Bộ duyệt này là sự cụ thể hoá của lớp `_IndexIterator` đã được xây dựng. Định nghĩa của bộ duyệt này gần giống với đinh nghĩa của bộ duyệt cho `vector`. Tuy nhiên, cần lưu ý rằng trong phép cụ thể hoá bộ duyệt này, ta truyền cho tham số `Pointer` của `_IndexIterator<>` giá trị là `deque<_Ty>::iterator` nhằm tận dụng tối đa những gì STL đã làm. Một lý do khác là ta đã xác định cách tổ chức các phần tử trong `OneDimStorage` theo `deque<>`. Điều này hoàn toàn tương tự đối với `const_iterator`. Cùng với việc định nghĩa lại các bộ duyệt, ta cần định nghĩa lại một số hàm thành phần quan trọng như `begin()` và `end()`. Cuối cùng, ta định nghĩa toán tử `[]` cho phép truy nhập vào một phần tử trong bộ chứa.

Sau khi đã định nghĩa lớp `OneDimStorage`, ta định nghĩa lớp `Matrix` như sau:

```
template <typename _Ty,MatrixOrientation _orient = ColumnMatrix>
class Matrix
{
    deque<OneDimStorage<_Ty> > _core;
    int _rows,_cols;
public:
    typedef _Ty elem_type;
    typedef OneDimStorage<_Ty> one_dim_storage;
    typedef one_dim_storage::iterator cell_iterator;
    typedef deque<one_dim_storage>::difference_type
difference_type;
    typedef deque<one_dim_storage>::size_type size_type;
    typedef deque<one_dim_storage>::reference reference;
    typedef deque<one_dim_storage>::const_reference
const_reference;
    typedef _IndexIterator<_Ty,
            difference_type,
            deque<one_dim_storage>::iterator,
            reference> one_dim_iterator;
    typedef _IndexIterator<_Ty,
            difference_type,
            deque<one_dim_storage>::const_iterator,
            const_reference> const_one_dim_iterator;
public:
    Matrix();
    Matrix(int rows,int cols);
    ~Matrix();
public:
    one_dim_iterator begin();
    one_dim_iterator end();
    one_dim_storage& operator[](int index);
    _Ty& operator()(int row,int column);
public:
    void SetOrientation(MatrixOrientation orient);
    MatrixOrientation GetOrientation();
    int GetRows();
    int GetCols();
};
```

So với phiên bản lần trước, lớp `Matrix` lần này có một số thay đổi trong định nghĩa bộ chứa lưu trữ vật lý `_core` và các bộ duyệt `one_dim_iterator` cũng như `cell_iterator`. Việc cụ thể hoá bộ duyệt

one_dim_iterator cũng được thực hiện giống nhu đối với bộ duyệt của OneDimStorage. Cùng với sự thay đổi này là sự thay đổi trên cấu tử và một số hàm thành phần như begin(), end(). Định nghĩa cụ thể của các hàm thành phần này các bạn có thể tham khảo trong đĩa CD đi kèm cuốn sách này.

Bây giờ chúng ta thử nghiệm lớp Matrix mới này với một ví dụ nhỏ. Ví dụ này sẽ in ra chỉ số của hàng và cột theo từng phần tử của một ma trận kích thước 10x10:

```
#include "Matrix.h"
#include <iostream>

int main(int argc,char* argv[])
{
    Matrix<int> int_matrix(10,10);
    Matrix<int>::one_dim_iterator it = int_matrix.begin();
    Matrix<int>::cell_iterator cell_it;
    for(;it != int_matrix.end();it++)
    {
            for(cell_it = it->begin();
                    cell_it != it->end();
                    cell_it++)
             cout << "(" << it.index() << "," << cell_it.index()
<< ") ";
            cout << endl;
    }
    return 0;
}
```

```
(0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7) (0,8) (0,9)
(1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7) (1,8) (1,9)
(2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7) (2,8) (2,9)
(3,0) (3,1) (3,2) (3,3) (3,4) (3,5) (3,6) (3,7) (3,8) (3,9)
(4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7) (4,8) (4,9)
(5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7) (5,8) (5,9)
(6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6) (6,7) (6,8) (6,9)
(7,0) (7,1) (7,2) (7,3) (7,4) (7,5) (7,6) (7,7) (7,8) (7,9)
(8,0) (8,1) (8,2) (8,3) (8,4) (8,5) (8,6) (8,7) (8,8) (8,9)
(9,0) (9,1) (9,2) (9,3) (9,4) (9,5) (9,6) (9,7) (9,8) (9,9)
```

3.6. Tóm tắt

3.6.1. Ghi nhớ

Bộ duyệt là một khái niệm trong lý thuyết "Thiết kế mẫu". Nó được đề xuất như một công cụ để truy xuất, cập nhật các phần tử trong bộ chứa. Với nhiệm vụ này, nó đóng vai trò giao diện giữa các giải thuật và bộ chứa. Do vậy, nó giúp các giải thuật không cần quan tâm chi tiết tới tổ chức của các bộ chứa. Nhờ đó, các giải thuật có tính tổng quát hơn.

Ngoài các bộ duyệt riêng gắn liền với từng bộ chứa, STL còn có rất nhiều các bộ duyệt độc lập khác. Các bộ duyệt này bao gồm các bộ duyệt thích nghi và các bộ duyệt trên luồng vào/ra. Các bộ duyệt thích nghi trên bộ duyệt được xây dựng trên các bộ duyệt cơ sở cho những mục đích đặc biệt chứ không đơn thuần chỉ là truy xuất phần tử trong bộ duyệt. Ví dụ, bộ duyệt `front_insert_iterator` dùng bổ sung phần tử vào một bộ chứa nào đó, bộ duyệt `raw_storage_iterator` dùng để đẩy dữ liệu vào một vùng nhớ đã được cấp phát. Các bộ duyệt trên luồng vào ra cho phép các khuôn hình giải thuật làm việc với các thiết bị vào/ra như màn hình, bàn phím và tệp tin. Thực chất, bộ duyệt trên luồng vào ra cũng có thể coi là bộ duyệt cụ thể của một bộ chứa. Tuy nhiên, do các luồng vào ra là các luồng đặc biệt nên các bộ chứa trên luồng vào ra được nhìn nhận như một bộ duyệt độc lập.

Khi xem xét các thiết kế của bộ duyệt, người ta thường quan tâm đến hai vấn đề lớn. Thứ nhất là là liên kết của bộ duyệt với bộ chứa. Thứ hai là các bộ duyệt phải được thiết kế để các khuôn hình giải thuật có tính tổng quát cao. Đối với vấn đề đầu tiên, có hai hướng tiếp cận để giải quyết là liên kết chặt và liên kết lỏng. Theo hướng thứ nhất, bộ duyệt lưu trữ toàn bộ thông tin về bộ chứa thông qua một con trỏ chỉ tới bộ chứa của nó. Theo hướng thứ hai, bộ duyệt chỉ lưu thông tin về phần tử hiện tại nó đang chỉ tới. Ngoài ra nó cũng sử dụng thêm các thông tin về tổ chức lưu trữ vật lý để có thể tìm ra phần tử kế tiếp của phần tử hiện tại. Vấn đề thứ hai trong thiết kế bộ duyệt liên quan mật thiết tới thiết kế các giải thuật. Vấn đề này cũng có hai cách giải quyết. Cách thứ nhất sử dụng tính đa hình của lập trình hướng đối tượng. Tất cả các bộ duyệt phải được kế thừa từ một bộ duyệt chung đóng vai trò cung cấp giao diện. Cách này cho phép một hàm thành phần được thực hiện theo các cơ chế khác nhau tuỳ thuộc vào bộ duyệt cụ thể. Cách thứ hai cũng với mục đích tương tự nhưng sử dụng `khuôn hình`. Lúc này, các giải thuật được thiết kế dưới dạng các `khuôn hình` hàm. Do vậy, các bộ duyệt không cần kế thừa từ một bộ duyệt chung. Trong STL, các bộ duyệt được thiết kế

theo hướng liên kết lỏng, còn các giải thuật được cung cấp dưới dạng khuôn hình hàm.

Nhu cầu xây dựng bộ duyệt mới đươc phát sinh khi người dùng đòi hỏi một số tác vụ đặc biệt trên bộ duyệt. Việc xây dựng bộ duyệt mới nên tận dụng tối đa những gì STL đã làm được.

3.6.2. Một số lưu ý khi lập trình

- Nếu sử dụng bộ duyệt `front_insert_iterator` với các bộ chứa tự xây dựng thì bộ chứa phải có hàm thành phần `push_front(const value_type var)`. Trong trường hợp không có hàm thành phần này, sẽ có thông báo lỗi "`X` không có hàm `push_front`" với `X` là lớp bộ chứa sử dụng với `font_insert_iterator`.
- Lưu ý này đúng với `back_insert_iterator` và `insert_iterator`. Hai hàm thành phần của bộ được sử dụng trong hai bộ duyệt này là `push_back(var)` và `insert(pos,var)`.

3.6.3. Bài tập

Bài tập 3.1.

Giả thiết rằng, nhìn nhận lớp ma trận như một mô hình của KHÁI NIỆM Ma trận. Viết lại các thao tác được yêu cầu trong bài tập chương trước dưới dạng các hàm toàn cục với đối số là bộ duyệt của lớp ma trận (các giải thuật hàm trên ma trận).

Chương 4

ĐỐI TƯỢNG HÀM

Mục đích chương này

- Giới thiệu về đối tượng hàm
- Biết cách sử dụng đối tượng hàm trong chương trình
- Phân biệt đối tượng hàm với con trỏ hàm
- Thấy được các khả năng của đối tượng hàm trong lập trình khái lược
- Tìm hiểu các dạng đối tượng hàm cơ bản và các đối tượng hàm có sẵn trong STL.
- Tìm hiểu cách kết hợp những đối tượng hàm cơ bản để có được đối tượng hàm mới.

4.1. Đối tượng hàm

4.1.1. Khái niệm đối tượng hàm

Trong lập trình C, để gọi tới các hàm callback người ta thường sử dụng các con trỏ hàm. Tuy nhiên, trong lập trình hướng đối tượng, một hàm có thể được đóng gói trong một đối tượng gọi là đối tượng hàm (`function object` hay `functor`). Với đối tượng hàm ta có một cách tiếp cận khác thay cho con trỏ hàm. Đối tượng hàm tuy là một khái niệm không mới, nhưng nó lại đặc biệt hữu ích trong lập trình khái lược. Một cách đơn giản, đối tượng hàm là một đối tượng có thể được *gọi* như một hàm. Một hàm bình thường hay một con trỏ hàm cũng có thể coi là một đối tượng hàm. Xét trên khía cạnh lập trình khái lược, đối tượng hàm là đối tượng có định nghĩa hoặc nạp chồng toán tử gọi hàm `operator()`. Trong quyển sách này, nếu nhắc đến đối tượng hàm mà không có chú ý gì đặt biệt thì ngầm hiểu là đối tượng có toán tử `operator()`. Xét ví dụ đơn giản sau:

```
class less {
public:
   less (int v) : val (v) {} // Cấu tử
   int operator () (int v)   // Toán tử gọi hàm
```

```
    {
        return v < val;
    }
private:
    int val;
};
```

Đối tượng `less` là một đối tượng hàm đơn giản thực hiện phép so sánh nhỏ hơn giữa hai số nguyên. Để sử dụng, trước tiên phải khởi tạo đối tượng:

```
less less_than_five (5);
```

Cấu tử được gọi với tham số v bằng 5 gán cho thành phần riêng `val`. Khi áp dụng đối tượng hàm, giá trị trả về là kết quả so sánh giữa `val` và giá trị truyền vào cho lời gọi hàm.

```
cout << "2 nho hon 5: " << (less_than_five (2) ? "dung" : "sai");
```

Kết quả:

```
2 nho hon 5: dung
```

Cần chú ý tới một điểm có thể gây nhầm lẫn cho người mới làm quen với đối tượng hàm, đó là có hai lời gọi `less_than_five(5)` và `less_than_five(2)`. Hai lời gọi này hoàn toàn khác nhau. Lời gọi đầu gọi cấu tử để khởi tạo đối tượng `less_than_five`, còn lời gọi sau là gọi đến toán tử gọi hàm `operator()`.

Có thể áp dụng khuôn hình cho đối tượng hàm trong ví dụ trên để có một đối tượng hàm so sánh kiểu bất kì. Khi đó, ta có khái niệm đối tượng hàm khái lược. Trong cuốn sách này, trừ một số ví dụ cơ bản không sử dụng khuôn hình, còn lại các đối tượng hàm đều được viết với kỹ thuật khuôn hình, nên ta quy ước gọi chung là đối tượng hàm.

```
template <class T>
class less {
public:
    less (T v) : val (v) {} // Cấu tử
    int operator () (T v)   // Toán tử gọi hàm
    {
        return v < val;
    }
private:
```

```
    T val;
};
```

Để so sánh một số nguyên hay một số thực ta khởi tạo hai đối tượng tương ứng:

```
less<int> less_than_five(5);
less<float> less_than_Pi(3.14);
```

Sau đó áp dụng trong chương trình như ví dụ đầu:

```
cout << "2 nho hon 5: " << (less_than_five (2) ? "yes" : "no");
cout << "4 nho hon Pi: " << (less_than_Pi (4) ? "dung" : "sai");
```

Kết quả:

```
2 nho hon 5: dung
4 nho hon Pi: sai
```

4.1.2. Sử dụng đối tượng hàm

Hai ví dụ trong mục trước mới chỉ cho người đọc hiểu được khái niệm về đối tượng hàm. Tuy đơn giản, nhưng đối tượng hàm lại đóng một vai trò quan trọng trong lập trình khái lược, đặc biệt là trong các giải thuật của STL sẽ được xem xét đến trong chương sau. Khi lập trình với STL, đối tượng hàm thường được truyền như tham số cho các khuôn hình giải thuật hoặc như các tham số khuôn hình cho khởi tạo các bộ chứa. Nhưng dù được sử dụng trực tiếp trong chương trình hay để truyền tham số, bao giờ đối tượng hàm cũng cần được khởi tạo trước sau đó mới được gọi đến toán tử gọi hàm. Do hai lời gọi này giống nhau về hình thức như đã nói trong mục trước nên người mới làm quen với đối tượng hàm có thể nhầm lẫn và chỉ gọi một lần.

Xét ví dụ đơn giản về cách sử dụng đối tượng hàm để truyền tham số cho hàm. Do đối tượng hàm bản thân là một đối tượng nên việc truyền tham số với đối tượng hàm cũng tương tự như việc truyền tham số với một đối tượng bình thường khác. Tuy nhiên, với đối tượng hàm, mục đích sử dụng chủ yếu là toán tử gọi hàm nên sau khi đối tượng hàm được truyền tham số cho một hàm, hàm đó sẽ gọi tới toán tử gọi hàm của đối tượng hàm đó.

```
#include < iostream.h >

class square {
```

```
public:
    int operator()(int n) {return n*n;} // Tính bình phương
};
class cube {
public:
    int operator()(int n) {return n*n*n;} // Tính lập phương
};
template <class funcObj >
void printEval(int val,funcObj f) { // f được truyền như tham số
    cout << f(val) << endl; // Gọi tới toán tử gọi hàm của f
};

int main() {
    printEval(4, square()); // In ra 16
    printEval(4, cube());   // In ra 64
    return 0;
}
```

Hàm `printEval` nhận hai tham số là số nguyên `val` và đối tượng hàm `f`. Do `printEval` là hàm khuôn hình nên khi sử dụng `f` có thể là hàm `square` hay `cube` đều được. Hàm `f` nhận `val` là tham số cho toán tử gọi hàm và `printEval` in ra kết quả của việc thực hiện toán tử đó.

```
16
64
```

Chú ý rằng trong lời gọi hàm `printEval(4,square())` đối tượng hàm `square` đã được khởi tạo bằng cấu tử không đối `square()`, sau đó trong thân hàm `printEval` nó mới gọi tới toán tử gọi hàm của mình. Để tránh nhầm lẫn ta có thể viết tường minh như sau:

```
square<int> my_square;
printEval(4, my_square);
```

4.1.3. Đối tượng hàm và con trỏ hàm

Trong ví dụ ở trên ta sử dụng đối tượng hàm nhưng có thể thấy nếu sử dụng con trỏ hàm cũng cho kết quả tương tự. Xét hàm `square2`:

```
int square2(int n)
{
    return n*n;
}
```

Trong chương trình có thể dùng hàm `printEval` với tham số thứ hai là con trỏ hàm `square2` và có kết quả tương tự như dùng đối tượng hàm `square`.

```
printEval(4, square2); // In ra 16
```

Đến đây, người đọc có thể hỏi tại sao lại dùng đối tượng hàm trong khi với con trỏ hàm cũng có kết quả tương tự. Ta chú ý rằng trong ví dụ về các đối tượng hàm `square`, `cube` và `square2` chưa sử dụng kỹ thuật khuôn hình. Để có thể tính bình phương của một số kiểu bất kì phải viết lại đối tượng `square` thành khuôn hình lớp:

```
template < class T >
class square {
public:
    T operator()(T n) {return n*n;}
};
```

Khi đó, có thể dùng hàm `printEval` để in ra bình phương của một số kiểu bất kì.

```
printEval(4, square<int>());          // In ra 16
printEval(3.14, square<float>());  // In ra 9.8596
```

Tuy nhiên, khi sử dụng kỹ thuật khuôn hình để viết lại hàm `square2`, ta không có được kết quả tương tự.

```
template <class T>
T square2(T n)
{
    return n*n;
}
```

Giả sử trong chương trình có sử dụng hàm này như đối số cho hàm `printEval`:

```
printEval(4, square2<int>); // Sai
```

Cách viết trên không hợp lệ trong C++ vì không thể khởi tạo một khuôn hình hàm theo cách này. Khi thử viết và biên dịch hàm `square2`, tùy theo trình biên dịch có thể sẽ nhận được thông báo lỗi biên dịch hoặc thông báo lỗi liên kết. Giải pháp đúng đắn duy nhất trong trường hợp này là sử dụng đối tượng hàm.

Ví dụ trên minh họa cho tính khái lược của đối tượng hàm, một điểm mạnh không thể có khi sử dụng con trỏ hàm. Đó cũng là ưu điểm mạnh nhất của đối tượng hàm. Ta sẽ thấy rõ ưu điểm của tính khái lược này khi sử dụng đối tượng hàm trong các khuôn hình giải thuật hoặc trong khởi tạo các bộ chứa. Ngoài ra sử dụng đối tượng hàm còn có các lợi thế khác khi so sánh với con trỏ hàm:

Tính linh hoạt: Đối tượng hàm có thể được sửa đổi mà không gây ảnh hưởng tới việc sử dụng chúng.

Tính inline: Trình biên dịch có thể sinh code dạng inline cho các đối tượng hàm trong lúc đang biên dịch.

Tính đóng gói dữ liệu: Do đối tượng hàm bản thân là đối tượng nên nó mang tính đóng gói dữ liệu. Ta sẽ minh họa tính đóng gói dữ liệu của đối tượng hàm qua ví dụ tính tổng lũy thừa các số. Trong ví dụ trước, ta đã có các đối tượng hàm `square` và `cube` để tính lũy thừa các số. Bây giờ ta xét một đối tượng hàm nữa có nhiệm vụ tính tổng lũy thừa các số.

```
template <class T, class Power>
class sum_power {
private:
     T counter;           // Biến đếm lưu tổng lũy thừa
     Power power;  // Đối tượng hàm tính lũy thừa
public:
     sum_power(): counter(0){}  // Khởi tạo biến đếm bằng 0
     T operator()(T n)
     {
            counter += power(n);        // Tính tổng tích lũy
            return counter;
     }
};
```

Đối tượng hàm `sum_power` có một biến đếm `counter` để lưu tổng tích lũy của các lũy thừa. Trong chương trình, để tính tổng lũy thừa của các số trong một dãy số ta khởi tạo đối tượng hàm `sum_power` và lần lượt gọi toán tử gọi hàm của nó với đối số là các số trong dãy. `sum_power` sẽ lần lượt tính lũy thừa và cộng vào biến đếm của mình.

```
int ress, resc;
     sum_power<int, square<int> > ss;
     sum_power<int, cube<int> > sc;
     for (int i = 1; i < 4; i++)
     {
```

```
            ress = ss(i); // Tính tổng bình phương
            resc = sc(i); // Tính tổng lập phương
    }
    cout << ress << endl << resc;
}
```

```
14
36
```

Sau đây là chương trình hoàn chỉnh:

```
#include <iostream.h>

template <class T>
class square {
public:
    T operator()(T n) {return n*n;}
};
template <class T>
class cube {
public:
    T operator()(T n) {return n*n*n;}
};

template <class T, class Power>
class sum_power {
private:
     T counter;
     Power power;
public:
     sum_power(): counter(0){}
     T operator()(T n)
     {
            counter += power(n);
            return counter;
     }
};

int main()
{
     int ress, resc;
     sum_power<int, square<int> > ss;
     sum_power<int, cube<int> > sc;
     for (int i = 1; i < 4; i++)
     {
            ress = ss(i); // Tính tổng bình phương
```

```
            resc = sc(i); // Tính tổng lập phương
        }
        cout << ress << endl << resc;
    }
```

4.1.4. Ứng dụng đối tượng hàm

Qua những ví dụ đơn giản trình bày ở các mục trên, người đọc đã có được một cái nhìn khá rõ ràng về khái niệm đối tượng hàm, cách sử dụng chúng trong chương trình cũng như ưu điểm của đối tượng hàm so với con trỏ hàm. Ta sẽ tiếp tục xem xét các ví dụ nâng cao để thấy rõ hơn các tính năng của đối tượng hàm cũng như tầm quan trọng của đối tượng hàm trong lập trình khái lược bằng STL.

Trong một mục trước đây, ta đã nhắc đến việc sử dụng đối tượng hàm làm tham số khuôn hình cho khởi tạo các bộ chứa hoặc làm tham số trực tiếp cho các giải thuật của STL. Các bộ chứa của STL đã được giới thiệu đầy đủ trong chương 2, trong đó nhiều bộ chứa như `set`, `map` hay các bộ chứa liên quan như `multiset`, `multimap`, `hash_set`, `hash_map` đều có những hàm khởi tạo sử dụng đối tượng hàm như tham số cho khởi tạo. Các giải thuật của STL sẽ được giới thiệu chi tiết trong chương 5, nhưng ta vẫn có thể đưa ra ví dụ đơn giản để thấy được sự cần thiết của đối tượng hàm, mà không phải đi sâu vào tìm hiểu về giải thuật.

Trước tiên, xem xét ví dụ về sử dụng đối tượng hàm trong khởi tạo một bộ chứa cụ thể là `set`. Chi tiết về `set` người đọc có thể xem lại trong chương 2, ở đây ta chỉ quan tâm đến một hàm khởi tạo của `set` có sử dụng đối tượng hàm `less_than` được cài đặt dưới đây.

```
template <class T>
class less_than
{
public:
    less_than() {}
    bool operator()(const T& x, const T& y) {return x<y;}
};
```

Trong chương trình, ta khởi tạo một tập hợp các số nguyên với `less_than` là tham số khuôn hình thứ hai. Khi đó, mỗi khi thêm một số nguyên vào tập hợp, các số trong tập hợp được sắp xếp theo thứ tự tăng dần bằng cách sử dụng đối tượng hàm so sánh `less_than`.

```
typedef set<int,less_than<int> > int_set; // Định nghĩa tập
nguyên
int_set is;  // Khai báo một tập nguyên
is.insert(2);
is.insert(5);
is.insert(3);
int_set::iterator j;
for (j = is.begin(); j != is.end(); j++)
{
    cout << *j << " ";
}
```

```
2 3 5
```

Chú ý là `less_than` được sử dụng như một đối số khuôn hình nên ta không phải quan tâm tới việc khởi tạo và sử dụng như thế nào trong chương trình. Điều đó đã được làm mặc định trong hàm khởi tạo của `set`. Ta chỉ phải cung cấp một tham số khuôn hình cụ thể hóa của `less_than` là `less_than<int>` để chỉ ra rằng dùng đối tượng hàm so sánh nhỏ hơn cho kiểu `int`.

Tất nhiên, ta có thể dùng `less_than` cho khởi tạo một tập hợp kiểu hoặc bất kì do người dùng định nghĩa miễn là kiểu, đối tượng đó có toán tử so sánh `operator<` sử dụng trong đối tượng hàm `less_than`. Xét đối tượng `person` được cài đặt như sau trong đó có nạp chồng toán tử `operator<`.

```
class person
{
public:
    person(){}
    person(const string& first,const string&
last):m_firstname(first),m_lastname(last){}
    string firstname() const {return m_firstname;}
    string lastname() const {return m_lastname;}

    friend int operator<(const person& p1, const person& p2);
    friend ostream& operator<<(ostream& s, const person& p);
private:
    string m_firstname;
    string m_lastname;
};

int operator<(const person& p1,const person& p2)
```

```
{
    return p1.lastname()<p2.lastname() ||
            (!(p2.lastname()<p1.lastname()) &&
            p1.firstname()<p2.firstname());
}

ostream& operator<< (ostream& s, const person& p)
{
    s << "[" << p.firstname() << " " << p.lastname() << "]";
    return s;
}

person p1("Doan Trung","Tung");
person p2("Dang Cong", "Kien");
person p3("Vuong Minh", "Hoa");
person p4("Cao Tran", "Kien");
typedef set<person,less_than<person> > person_set;
person_set ps;
ps.insert(p1);
ps.insert(p2);
ps.insert(p3);
ps.insert(p4);
person_set::iterator i;
for (i = ps.begin(); i != ps.end(); i++)
{
    cout << *i << endl;
}
```

```
Vuong Minh Hoa
Cao Tran Kien
Dang Cong Kien
Doan Trung Tung
```

Người đọc cũng có thể cài đặt đối tượng hàm `greater_than` và sử dụng để khởi tạo tập hợp các đối tượng được sắp xếp theo thứ tự giảm dần miễn là các đối tượng đó có nạp chồng toán tử `operator>`. Chúng tôi coi đó là bài tập để người đọc tự làm.

Rất nhiều khuôn hình giải thuật trong STL sử dụng đối tượng hàm như một tham số trực tiếp để thực hiện. Một khuôn hình giải thuật rất hay dùng là khuôn hình giải thuật sắp xếp `sort` (chi tiết về `sort` có thể xem trong chương 5). Khuôn hình giải thuật `sort` dùng để sắp xếp một dãy đối tượng theo một tiêu chí sắp xếp quy định bởi một đối tượng hàm xác định. Tiêu chí mặc định là sắp xếp tăng dần dùng `operator<` để so sánh. Tuy nhiên, nếu

muốn ta có thể viết một đối tượng hàm so sánh riêng hoặc sử dụng các đối tượng hàm sẵn có trong STL. Ví dụ dưới đây minh họa cả ba cách sử dụng. Ta vẫn sử dụng lớp `person` trong ví dụ trước có cài đặt thêm toán tử so sánh lớn hơn `operator>` và dùng khuôn hình giải thuật `sort` để sắp xếp `vector` v chứa một dãy các đối tượng `person` theo các tiêu chí khác nhau.

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

using namespace std;

class person
{
public:
    person(){}
    person(const string& first,const string&
last):m_firstname(first),m_lastname(last){}
    string firstname() const {return m_firstname;}
    string lastname() const {return m_lastname;}

    friend int operator<(const person& p1, const person& p2);
    friend int operator>(const person& p1, const person& p2);
    friend ostream& operator<<(ostream& s, const person& p);
private:
    string m_firstname;
    string m_lastname;
};
int operator<(const person& p1,const person& p2)
{
    return p1.lastname()<p2.lastname() ||
           (!(p2.lastname()<p1.lastname()) &&
           p1.firstname()<p2.firstname());
}
int operator>(const person& p1,const person& p2)
{
    return p1.lastname()>p2.lastname() ||
           (!(p2.lastname()>p1.lastname()) &&
           p1.firstname()>p2.firstname());
}
ostream& operator<< (ostream& s, const person& p)
{
    s << "[" << p.firstname() << " " << p.lastname() << "]";
```

```
    return s;
}
template <class T>
class less_than
{
public:
    less_than() {}
    bool operator()(const T& x, const T& y) {return x<y;}
};

int main()
{
    person p1("Doan Trung","Tung");
    person p2("Dang Cong", "Kien");
    person p3("Vuong Minh", "Hoa");
    person p4("Cao Tran", "Kien");
    vector<person> v;
    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    v.push_back(p4);

    sort(v.begin(),v.end());
    cout << "Sap xep mac dinh:" << endl;
    copy(v.begin(),v.end(),ostream_iterator<person>
(cout,"\n"));

    sort(v.begin(),v.end(),greater<person>());
    cout << "Sap xep dung doi tuong ham greater cua STL:" <<
endl;
    copy(v.begin(),v.end(),ostream_iterator<person>
(cout,"\n"));

    less_than<person> mycomp;
    cout << "Sap xep dung doi tuong ham nguoi dung dinh nghia:"
<< endl;
    sort(v.begin(),v.end(),mycomp);
    copy(v.begin(),v.end(),ostream_iterator<person>
(cout,"\n"));
    char c;
    cin >> c;
}
```

Trong chương trình trên cần chú ý các điểm sau. Để sử dụng khuôn hình giải thuật `sort` và khuôn hình giải thuật `copy` cần thêm dẫn hướng biên

dịch #include<algorithm> ở đầu chương trình. Để sử dụng đối tượng hàm greater cần thêm dẫn hướng biên dịch #include<functional>. Khuôn hình giải thuật copy sao chép toàn bộ các đối tượng trong vector v ra màn hình còn greater là đối tượng hàm có sẵn trong STL dùng để so sánh lớn hơn, thực hiện khi thực hiện giải thuật sắp xếp.

Kết quả thực hiện chương trình:

```
Sap xep mac dinh:
[Vuong Minh Hoa]
[Cao Tran Kien]
[Dang Cong Kien]
[Doan Trung Tung]
Sap xep dung doi tuong ham greater cua STL:
[Doan Trung Tung]
[Dang Cong Kien]
[Cao Tran Kien]
[Vuong Minh Hoa]
Sap xep dung doi tuong ham nguoi dung dinh nghia:
[Vuong Minh Hoa]
[Cao Tran Kien]
[Dang Cong Kien]
[Doan Trung Tung]
```

Với khả năng linh hoạt và tính khái lược, đối tượng hàm là một thành phần không thể thiếu trong bộ thư viện STL. Các khuôn hình giải thuật và bộ chứa trong STL sử dụng đối tượng hàm càng tăng thêm tính linh hoạt và tính khái lược của chúng. Đối tượng hàm không chỉ có ý nghĩa trong xây dựng và sử dụng thư viện mà còn có ý nghĩa thực tiễn trong lập trình khái lược. Phong cách lập trình sử dụng đối tượng hàm là một phong cách lập trình tốt cần nắm vững và sử dụng thành thạo.

4.2. Các KHÁI NIỆM về đối tượng hàm

Các KHÁI NIỆM cho đối tượng hàm được đưa ra trong STL nhằm mục đích mô tả các yêu cầu chung nhất về kiểu của các đối tượng hàm. Người đọc cần nắm vững các KHÁI NIỆM này để có thể sử dụng đối tượng hàm cùng với các thành phần khác của STL như giải thuật hay bộ chứa một cách đúng đắn và hiệu quả.

4.2.1. Các KHÁI NIỆM cơ bản

Các KHÁI NIỆM cơ bản trong STL gồm có Generator, Unary Function và Binary Function. Chúng tương ứng là các đối tượng hàm có toán tử gọi

hàm không đối `f()`, một đối `f(x)` và hai đối số `f(x,y)`. Tất nhiên, có thể mở rộng danh sách này lên Ternary Function (đối tượng hàm có toán tử gọi hàm ba đối) hoặc hơn nữa, nhưng thực tế không có khuôn hình giải thuật hoặc bộ chứa nào trong STL lại cần dùng đối tượng hàm có toán tử gọi hàm nhiều hơn hai tham số. Cũng cần chú ý rằng đây là các KHÁI NIỆM do STL định ra để phân loại các đối tượng hàm, do vậy hoàn toàn có thể có các đối tượng hàm đồng thời có toán tử gọi hàm nạp chồng không đối, một đối hay hai đối như sau:

```
#include <iostream>
using namespace std;
class compare {
private:
    int m_val;
public:
    compare(int val) : m_val(val){}
    // toan tu goi ham khong doi
    bool operator()()const
    {
        return m_val > 0;
    }
    // toan tu goi ham mot doi
    bool operator()(int x)const
    {
        return x > m_val;
    }
    // toan tu goi ham hai doi
    bool operator()(int x)const
    {
        return x > y;
    }
};

int main()
{
    compare v(4);               // Khoi tao
    cout << v()<< endl;         // Output: 1
    cout << v(5) << endl;           // Output: 1
    cout << v(5,6) << endl;         // Output: 0
    cout << endl;
    return 0;
}
```

Một đối tượng hàm như trên có thể coi là thuộc vào KHÁI NIỆM Generator, Unary Function hay Binary Function đều được. Đây là ba KHÁI

NIỆM cơ bản nhất của đối tượng hàm, các KHÁI NIỆM còn lại đều là trường hợp đặc biệt của ba KHÁI NIỆM này.

4.2.2. Các KHÁI NIỆM mệnh đề

Các đối tượng hàm có giá trị trả về kiểu `bool` là một trường hợp đặc biệt quan trọng. Một Unary Function có giá trị trả về kiểu `bool` được gọi là một Predicate và một Binary Function có giá trị trả về kiểu `bool` được gọi là một Binary Predicate. Đối tượng hàm `compare` ở trên chính là một ví dụ về Predicate và Binary Predicate.

Một trường hợp đặc biệt của Binary Predicate là KHÁI NIỆM Strict Weak Odering (Quan hệ thứ tự chặt hoặc yếu). Đây là KHÁI NIỆM cho các đối tượng hàm làm nhiệm vụ so sánh hai đối tượng và trả về giá trị `true` nếu đối tượng thứ nhất thỏa mãn điều kiện sánh được với đối tượng thứ hai. Mệnh đề so sánh phải thỏa mãn định nghĩa chuẩn của toán học cho quan hệ thứ tự chặt hoặc yếu:

- Không phản xạ: `f(x,x)` phải trả về giá trị `false`
- Không đối xứng: `f(x,y)` trả về `true` thì `f(y,x)` trả về `false` và ngược lại
- Bắc cầu: `f(x,y)` và `f(y,z)` thì `f(x,z)`
- Tương đương bắc cầu: `x` tương đương `y` và `y` tương đương `z` thì `x` tương đương `z`.

Ba tính chất đầu cho ta quan hệ thứ tự bộ phận. Các đối tượng hàm thuộc KHÁI NIỆM Strict Weak Odering như `less`, `greater` thường được sử dụng trong các giải thuật so sánh, tìm giá trị lớn nhất, nhỏ nhất,... Đối tượng hàm `compare` trong ví dụ trên với toán tử gọi hàm hai đối có thể coi là thuộc KHÁI NIỆM Strict Weak Odering.

4.2.3. Các KHÁI NIỆM khác

Một trường hợp đặc biệt nữa của KHÁI NIỆM Binary Function là KHÁI NIỆM Monoid Operator. Một đối tượng hàm thuộc Monoid Operator phải thỏa mãn một số điều kiện như: kiểu của tham số thứ nhất, kiểu tham số thứ hai và kiểu trả về của phép toán phải giống nhau. Như tên gọi của nó, KHÁI NIỆM này được đưa ra để dành cho các phép toán một ngôi. Các đối tượng hàm thuộc KHÁI NIỆM này là các phép toán cộng `plus` và nhân `multiplies`.

Ngoài ra còn một số KHÁI NIỆM khác không đề cập ở đây. Các KHÁI NIỆM dạng Adaptable là các KHÁI NIỆM cho các đối tượng hàm có thể sử dụng với bộ thích nghi trên đối tượng hàm. Người đọc tạm công nhận các KHÁI NIỆM này và sẽ được biết rõ hơn về bộ thích nghi trên đối tượng hàm cũng như các lớp thuộc KHÁI NIỆM adatable trong các mục 4.3.4 và 4.3.5.

Các KHÁI NIỆM đã được giới thiệu ở trên là những KHÁI NIỆM cơ bản và chung nhất về đối tượng hàm. Hiểu được ý nghĩa các KHÁI NIỆM đó cho ta một cái nhìn tổng quan và rõ ràng về đối tượng hàm trong STL đồng thời giúp ta sử dụng đối tượng hàm khi lập trình một cách dễ dàng hơn.

4.3. Các đối tượng hàm có sẵn

STL xây dựng sẵn rất nhiều đối tượng hàm tiện lợi cho sử dụng với khuôn hình giải thuật, với bộ chứa hay cho các mục đích khác của người lập trình. Các đối tượng hàm có sẵn trong STL được phân thành bốn lớp:

- Các phép toán số học
- Các phép so sánh
- Các phép toán logic
- Các bộ thích nghi trên đối tượng hàm

Ta sẽ lần lượt xem xét các đối tượng hàm của ba lớp đầu về cách sử dụng cũng như phạm vi ứng dụng trong lập trình.

Muốn sử dụng các đối tượng hàm có sẵn trong STL ta cần thêm dẫn hướng biên dịch `#include<functional>` vào đầu chương trình.

4.3.1. Các phép toán số học

4.3.1.1. plus

Đối tượng hàm `plus` thực hiện phép cộng hai đối tượng kiểu `T` và trả về kết quả kiểu `T`. Nếu `f` là một đối tượng hàm của lớp `plus`, `x` và `y` là hai đối tượng kiểu `T` thì `f(x,y)` trả về `x+y`. Các yêu cầu khi sử dụng `plus`:

- Yêu cầu `T` phải nạp chồng toán tử `operator+`
- Kiểu trả về của toán tử `operator+` phải có thể chuyển kiểu được thành kiểu `T`.
- `T` phải là Assignable

Ví dụ về sử dụng plus

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

int main()
{
    int a[5] = {15,12,9,4,79};
    int b[5] = {5,10,28,12,2001};
    plus<int> s;
    ostream_iterator<int> out(cout," ");
    transform(a,a+5,b,out,s);
}
```

```
20 22 37 16 2080
```

Trong chương trình có sử dụng transform là một khuôn hình giải thuật của STL. Hàm transform thực hiện một phép toán định nghĩa bởi tham số cuối trên một dãy các đối tượng và trả về kết quả phép toán lên một tham số khác. Cụ thể trong chương trình hàm transform thực hiện phép toán plus định nghĩa bởi plus<int> s lên các đối tượng của hai dãy nguyên a, b và ghi kết quả ra màn hình ostream_iterator<int> s(cout, " "). Viết như trên là cách viết tường minh dễ theo dõi, ta cũng có thể viết gọn hai lệnh khai báo vào trong hàm transform như sau:

```
transform(a,a+5,b,ostream_iterator<int> (cout, " "), plus<int>
());
```

Đối tượng hàm plus thuộc KHÁI NIỆM Adaptable Binary Function, do vậy bất cứ hàm nào của STL sử dụng tham số khuôn hình thuộc KHÁI NIỆM Adaptable Binary Function đều có thể sử dụng đối tượng hàm plus.

4.3.1.2. minus

Đối tượng hàm minus thực hiện phép toán ngược lại với plus. Nếu f là một đối tượng hàm của lớp minus, x và y là hai đối tượng kiểu T thì f(x,y) trả về x-y. Các yêu cầu khi sử dụng minus:

- Yêu cầu T phải nạp chồng toán tử operator-
- Kiểu trả về của operator- phải chuyển kiểu được thành kiểu T.
- T phải là Assignable

Ví dụ sử dụng minus:

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

int main()
{
     int a[5] = {15,12,9,4,79};
     int b[5] = {5,10,28,12,2001};
     transform(a,a+5,b,ostream_iterator<int>(cout,"
"),minus<int>());
}
```

```
10 2 -19 -8 -1922
```

Hàm transform thực hiện phép toán trừ lên các đối tượng của hai dãy a, b và đưa kết quả ra màn hình.

Đối tượng hàm minus thuộc KHÁI NIỆM Adaptable Binary Function nên bất kì hàm của STL sử dụng đối số là đối tượng hàm kiểu Adaptable Binary Function thì có thể sử dụng được minus.

4.3.1.3. multiplies

Đối tượng hàm multiplies thực hiện phép nhân hai số. Nếu f là một đối tượng của lớp multiplies, x và y là hai đối tượng kiểu T thì f(x,y) trả về x.y. Các yêu cầu khi sử dụng multiplies:

- Yêu cầu T phải nạp chồng toán tử operator*
- Kiểu trả về của operator* phải chuyển kiểu được thành kiểu T.
- T phải là Assignable

Ví dụ về sử dụng multiplies

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

int main()
{
     int a[5] = {15,12,9,4,79};
```

```
    int b[5] = {5,10,28,12,2001};
    transform(a,a+5,b,ostream_iterator<int>(cout," "),multiplies
<int> ());
}
```

```
75 120 252 48 158079
```

Hàm `transform` thực hiện phép nhân trên các đối tượng của hai dãy `a`, `b` và ghi kết quả ra màn hình.

Đối tượng hàm `multiplies` thuộc KHÁI NIỆM Adaptable Binary Function, nên bất kì hàm nào của STL sử dụng đối số là đối tượng hàm kiểu Adaptable Binary Function đều có thể sử dụng `multiplies`.

4.3.1.4. divides

Đối tượng hàm `divides` thực hiện phép chia /. Nếu `f` là một đối tượng của lớp `divides`, `x` và `y` là hai đối tượng kiểu `T` thì `f(x,y)` trả về `x/y`. Các yêu cầu khi sử dụng `divides`:

- Yêu cầu `T` phải nạp chồng toán tử `operator/`
- Kiểu trả về của `operator/` phải chuyển kiểu được thành kiểu `T`.
- `T` phải là Assignable

Ví dụ về sử dụng `divides`

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

int main()
{
    int a[5] = {15,12,9,4,79};
    int b[5] = {5,10,28,12,2001};
    transform(a,a+5,b,ostream_iterator<int>(cout,"
"),divides<int> ());
}
```

```
3 1 0 0 0
```

Hàm `transform` thực hiện phép chia lên các đối tượng của hai dãy `a`,`b` và ghi kết quả ra màn hình.

Đối tượng hàm `divides` thuộc KHÁI NIỆM Adaptable Binary

Function nên bất kì hàm nào của STL sử dụng đối số là đối tượng hàm kiểu Adaptable Binary Function đều có thể sử dụng divides.

4.3.1.5. modulus

Đối tượng hàm modulus thực hiện phép chia %. Nếu f là một đối tượng của lớp modulus, x và y là hai đối tượng kiểu T thì f(x,y) trả về x%y. Yêu cầu khi sử dụng modulus:

- T phải nạp chồng toán tử operator%
- Kiểu trả về của operator% phải chuyển kiểu được thành kiểu T.
- T phải là Assignable

Ví dụ về sử dụng modulus

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

int main()
{
     int a[5] = {15,12,9,4,79};
     int b[5] = {5,10,28,12,2001};
     transform(a,a+5,b,ostream_iterator<int>(cout," "),modulus
<int> ());
}
```

```
0 2 9 4 79
```

Hàm transform thực hiện phép chia % lên các đối tượng của hai dãy a, b và ghi kết quả ra màn hình.

Đối tượng modulus thuộc KHÁI NIỆM Adaptable Binary Function nên bất kì hàm nào của STL sử dụng đối số là đối tượng hàm kiểu Adaptable Binary Function đều có thể sử dụng modulus.

4.3.1.6. negate

Đối tượng hàm negate thực hiện phép toán phủ định. Nếu f là một đối tượng của lớp negate, x là đối tượng kiểu T thì f(x) trả về -x. Các yêu cầu khi sử dụng negate:

- T phải nạp chồng toán tử operator-

- Kiểu trả về của toán tử `operator-` phải chuyển kiểu được thành kiểu `T`
- `T` phải là Assignable

Ví dụ về sử dụng `negate`:

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

int main()
{
    int a[5] = {15,12,9,4,79};
    transform(a,a+5,ostream_iterator<int> (cout,"
"),negate<int>());
}
```

```
-15 -12 -9 -4 -79
```

Chú ý trong ví dụ trên ta sử dụng một hàm nạp chồng khác của `transform` (STL có hai khuôn hình giải thuật `transform`) trong đó đối số cuối là một hàm thuộc KHÁI NIỆM Adaptable Unary Function. Hàm `transform` thực hiện phép toán phủ định lên các đối tượng của dãy `a` và in kết quả ra màn hình.

Đối tượng hàm `negate` thuộc KHÁI NIỆM Adaptable Unary Function nên bất cứ hàm nào của STL sử dụng đối số là đối tượng hàm kiểu Adaptable Unary Function đều có thể sử dụng được `negate`.

4.3.2. Các phép toán so sánh

Đối tượng hàm cho các phép toán so sánh cũng là những đối tượng hàm đơn giản nhưng được sử dụng rất phổ biến trong các giải thuật STL, đặc biệt là các giải thuật liên quan đến sắp xếp hay phân hoạch.

4.3.2.1. equal_to

Đối tượng hàm `equal_to` thực hiện phép so sánh bằng nhau giữa hai đối tượng. Nếu `f` là một đối tượng của lớp `equal_to`, `x` và `y` là hai đối tượng kiểu `T` thì `f(x,y)` trả về `true` nếu `x` và `y` bằng nhau, trả về `false` nếu `x` và `y` khác nhau. Yêu cầu khi sử dụng `equal_to`:

- T phải là Equality Comparable hay T chỉ cần nạp chồng toán tử `operator==` và toán tử này là quan hệ tương đương.

Ví dụ về sử dụng `equal_to`:

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

using namespace std;

template <class InputIterator,class T,class BinaryPredicate>
int object_count(InputIterator first,InputIterator last,T key,int
count=0)
{
     BinaryPredicate comp;
     for(InputIterator i = first;i != last;i++)
            if (comp(*i,key)) count++;
     return count;
}
int main()
{
     vector<int> V;
     V.push_back(1);
     V.push_back(3);
     V.push_back(3);
     V.push_back(3);
     V.push_back(2);
     V.push_back(2);
     V.push_back(1);
     V.push_back(3);
     copy(V.begin(),V.end(),ostream_iterator<int> (cout, " "));

     typedef vector<int>::iterator IntVecItr;
     typedef equal_to<int> eComp;

     cout << "\nSo lan xuat hien cua 4 trong day: "
          << object_count<IntVecItr,int,eComp> (V.begin(),V.end()
,4)
           << endl
            << "So lan xuat hien cua 3 trong day"
            << object_count<IntVecItr,int,eComp>
            (V.begin(),V.end() ,3)
            << endl;
      }
```

```
1 3 3 3 2 2 1 3
So lan xuat hien cua 4 trong day: 0
So lan xuat hien cua 3 trong day: 4
```

Hàm object_count duyệt qua các đối tượng của một dãy và so sánh đối tượng với key theo tiêu chuẩn so sánh Binary Predicate. Cụ thể, trong chương trình object_count được định nghĩa để duyệt trên các đối tượng kiểu vector<int> với hàm so sánh equal_to<int>.

Đối tượng hàm equal_to thuộc KHÁI NIỆM Adaptable Binary Predicate, do đó bất kì hàm nào của STL (hoặc hàm người dùng tự định nghĩa) sử dụng tham số khuôn hình là đối tượng hàm kiểu Adaptable Binary Predicate thì đều sử dụng được với equal_to.

4.3.2.2. not_equal_to

Đối tượng hàm not_equal_to thực hiện phép so sánh bằng nhau giữa hai đối tượng. Nếu f là một đối tượng của lớp not_equal_to, x và y là hai đối tượng kiểu T thì f(x,y) trả về true nếu x và y khác nhau, trả về false nếu x và y bằng nhau. Yêu cầu khi sử dụng equal_to:

- T phải là Equality Comparable hoặc T có nạp chồng toán tử operator!=.

Ví dụ sử dụng not_equal_to:

```
    typedef not_equal_to<int> neComp;

    cout << "\nSo phan tu khac 4 trong day: "
         << object_count<IntVecItr,int,neComp> (V.begin(),
V.end() ,4) << endl
         << "So phan tu khac 3 trong day: "
         << object_count<IntVecItr,int,neComp> (V.begin(),
V.end() ,3) << endl;
```

```
1 3 3 3 2 2 1 3
So phan tu khac 4 trong day: 8
So phan tu khac 3 trong day: 4
```

Đối tượng hàm not_equal_to thuộc KHÁI NIỆM Adaptable Binary Predicate, do đó bất kì hàm nào của STL hoặc người dùng định nghĩa có sử dụng tham số khuôn hình là Adaptable Binary Predicate đều sử dụng được not_equal_to.

4.3.2.3. less

Đối tượng hàm `less` thực hiện phép so sánh nhỏ hơn giữa hai đối tượng. Nếu `f` là đối tượng của lớp `less`, `x` và `y` là hai đối tượng kiểu `T` thì `f(x,y)` trả về `true` nếu `x < y` và ngược lại. Yêu cầu khi sử dụng `less`:

- `T` là kiểu Less Than Comparable hoặc `T` có nạp chồng toán tử `operator<` và toán tử này là quan hệ thứ tự một phần.

Ví dụ về sử dụng `less`:

```
typedef less<int> lComp;

cout << "\nSo phan tu nho hon 4 trong day: "
     << object_count<IntVecItr,int,lComp> (V.begin(),V.end()
,4) << endl
     << "So phan tu nho hon 3 trong day: "
     << object_count<IntVecItr,int,lComp> (V.begin(),V.end()
,3) << endl;
```

```
1 3 3 3 2 2 1 3
So phan tu nho hon 4 trong day: 8
So phan tu nho hon 3 trong day: 4
```

Đối tượng hàm `less` thuộc KHÁI NIỆM Adaptable Binary Predicate, do đó bất kì hàm nào của STL hoặc người dùng định nghĩa có sử dụng tham số khuôn hình là Adaptable Binary Predicate đều sử dụng được `less`.

4.3.2.4. less_equal

Đối tượng hàm `less_equal` hoàn toàn tương tự như `less` nhưng nó thực hiện phép so sánh nhỏ hơn hoặc bằng.

Ví dụ sử dụng `less_equal`:

```
typedef less_equal<int> leComp;

cout << "\nSo phan tu nho hon hoac bang 2 trong day: "
     << object_count<IntVecItr,int,leComp>
(V.begin(),V.end() ,2) << endl;
```

```
1 3 3 3 2 2 1 3
So phan tu nho hon hoac bang 2 trong day: 4
```

4.3.2.5. greater

Đối tượng hàm `greater` thực hiện phép so sánh lớn giữa hai đối tượng. Nếu `f` là một đối tượng của lớp `greater`, `x` và `y` là hai đối tượng kiểu `T` thì `f(x,y)` trả về `true` nếu `x > y` và ngược lại. Yêu cầu khi sử dụng `greater`:

- `T` là kiểu Less Than Comparable hoặc có nạp chồng toán tử `operator>`

Ví dụ sử dụng `greater`:

```
typedef greater<int> gComp;

    cout << "\nSo phan tu lon hon 2 trong day: "
         << object_count<IntVecItr,int,gComp> (V.begin(),V.end()
,2) << endl;
```

```
1 3 3 3 2 2 1 3
So phan tu lon hon 2 trong day: 4
```

Đối tượng hàm `greater` thuộc KHÁI NIỆM Adaptable Binary Predicate, do đó bất kì hàm nào của STL (hoặc hàm người dùng tự định nghĩa) sử dụng tham số khuôn hình là đối tượng hàm kiểu Adaptable Binary Predicate thì đều sử dụng được với `greater`.

4.3.2.6. greater_equal

Đối tượng hàm `greater_equal` cũng tương tự như đối tượng hàm `greater` nhưng thực hiện phép so sánh lớn hơn hoặc bằng giữa hai đối tượng.

Ví dụ sử dụng `greater_equal`:

```
typedef greater_equal<int> geComp;

    cout << "\nSo phan tu lon hon hoac bang 2 trong day: "
         << object_count<IntVecItr,int,geComp>
(V.begin(),V.end() ,2) << endl;
```

```
1 3 3 3 2 2 1 3
So phan tu lon hon hoac bang 2 trong day: 6
```

4.3.3. Các phép toán logic

Các đối tượng hàm biểu diễn các phép toán logic trong STL không được sử dụng trực tiếp một cách rộng rãi như với các phép toán số học hay các phép so sánh. Tuy nhiên, khi sử dụng chúng với các bộ thích nghi của đối tượng hàm chúng lại tỏ ra rất hữu ích.

4.3.3.1. logical_and

Đối tượng hàm `logical_and` biểu diễn phép logic VÀ toán học. Nếu `f` là một đối tượng của lớp `logical_and`, `x` và `y` là hai đối tượng kiểu `T` thì `f(x,y)` trả về `true` khi và chỉ khi cả `x` và `y` là `true`. Yêu cầu khi sử dụng `logical_and`:

- `T` phải chuyển kiểu được thành kiểu `bool`.

Ví dụ sử dụng `logical_and`:

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
#include <stdlib.h>
#include <time.h>
int main( )
{
     using namespace std;
     vector <bool> v1, v2, v3( 7 );


     srand( (unsigned)time( NULL ) );

     for ( int i = 0 ; i < 7 ; i++ )
     {
            v1.push_back((bool) (rand()%2));
            v2.push_back((bool) (rand()%2));
     }

     cout << boolalpha;    // Dat co in gia tri logic

      ostream_iterator<bool> out(cout," ");

     cout << "Vector v1:" << endl;
     copy(v1.begin(),v1.end(),out);
```

```
    cout << "\nVector v2:" << endl;
    copy(v2.begin(),v2.end(),out);

    cout << "\nKet qua phep AND hai vector:" << endl;
    transform(v1.begin(),v1.end(),v2.begin(),out,
              logical_and<bool>());
}
```

```
Vector v1:
true true false false true true false
Vector v2:
false false true false true false false
Ket qua phep AND hai vector:
false false false false true false false
```

Hàm `transform` thực hiện phép toán logic VÀ lên các phần tử của hai `vector` `v1` và `v2` rồi ghi kết quả ra màn hình.

Đối tượng hàm `logical_and` thuộc KHÁI NIỆM Adaptable Binary Predicate, do đó bất kì hàm nào của STL hoặc của người dùng định nghĩa có sử dụng tham số khuôn hình thuộc KHÁI NIỆM Adaptable Binary Predicate đều sử dụng được `logical_and`.

4.3.3.2. logical_or

Đối tượng hàm `logical_or` biểu diễn phép logic HOẶC toán học. Nếu `f` là một đối tượng của lớp `logical_or`, `x` và `y` là hai đối tượng kiểu `T` thì `f(x,y)` trả về `true` khi và chỉ khi `x` hoặc `y` là `true`. Yêu cầu khi sử dụng `logical_or`:

- `T` phải chuyển kiểu được thành kiểu `bool`.

Ví dụ sử dụng `logical_or`: trong ví dụ về `logical_and`, thay hai dòng cuối bằng hai dòng sau:

```
cout << "\nKet qua phep OR hai vector:" << endl;
transform(v1.begin(),v1.end(),v2.begin(),out,logical_or<bool>());
```

```
Vector v1:
false false false true false false false
Vector v2:
true false false false true false false
Ket qua phep OR hai vector:
true false false true true false false
```

Hàm `transform` thực hiện phép toán logic HOẶC lên các phần tử của hai `vector v1` và `v2` rồi ghi kết quả ra màn hình.

Đối tượng hàm `logical_or` thuộc KHÁI NIỆM Adaptable Binary Predicate, do đó bất kì hàm nào của STL hoặc của người dùng định nghĩa có sử dụng tham số khuôn hình thuộc KHÁI NIỆM Adaptable Binary Predicate đều sử dụng được `logical_or`.

4.3.3.3. logical_not

Đối tượng hàm `logical_not` biểu diễn phép logic PHỦ ĐỊNH toán học. Nếu `f` là một đối tượng của lớp `logical_not`, `x` và `y` là hai đối tượng kiểu `T` thì `f(x,y)` trả về `true` khi và chỉ khi `x` hoặc `y` là `true`. Yêu cầu khi sử dụng `logical_not`:

- `T` phải chuyển kiểu được thành kiểu `bool`.

Ví dụ sử dụng `logical_not`: trong ví dụ về `logical_and`, thay hai dòng cuối bằng hai dòng sau:

```
cout << "\nPhu dinh cua v1: " << endl;
transform(v1.begin(),v1.end(),out,logical_not<bool>() );
```

```
Vector v1:
false false false true false false false
Phu dinh cua v1:
true true true false true true true
```

Đối tượng hàm `logical_not` thuộc KHÁI NIỆM Adaptable Predicate, do đó bất kì hàm nào của STL hoặc của người dùng định nghĩa có sử dụng tham số khuôn hình thuộc KHÁI NIỆM Adaptable Binary Predicate đều sử dụng được `logical_not`.

4.3.4. Các bộ thích nghi của đối tượng hàm

Ta đã biết rằng, bộ thích nghi là các lớp trung gian làm nhiệm vụ chuyển các thông điệp tới các lớp muốn nhận thông điệp. Qua các chương về bộ chứa và bộ duyệt ta đã làm quen và thấy được lợi ích của bộ thích nghi đối với chúng như thế nào. Với đối tượng hàm, ta cũng có các bộ thích nghi riêng. Các bộ thích nghi này sẽ đem đến một *diện mạo mới* cho những đối tượng hàm ta đã xem xét ở trên. Thực vậy, những đối tượng hàm cơ bản STL cung cấp chưa thể thỏa mãn hết các yêu cầu của người lập trình. Sử dụng các bộ thích nghi ta sẽ thấy sự tiện lợi và uyển chuyển trong việc kết hợp các đối

tượng hàm cơ bản để thỏa mãn được các yêu cầu từ đơn giản đến phức tạp mà STL không thể đáp ứng hết được.

Ngoài định nghĩa chung về bộ thích nghi trong chương 1, với đối tượng hàm, các bộ thích nghi cho chúng có thể được định nghĩa như sau: *Bộ thích nghi trên đối tượng hàm là các lớp làm nhiệm vụ tạo ra một đối tượng hàm mới từ một hay nhiều đối tượng hàm đã có.*

Lần lượt các bộ thích nghi được giới thiệu dưới đây sẽ cho thấy rõ tất cả các điều ta đã nhắc đến ở trên. Người đọc nên đọc kỹ tất cả các bộ thích nghi, vì có những điều giải thích cho bộ thích nghi này hoàn toàn có thể áp dụng được cho các bộ thích nghi khác nên trong trình bày chúng sẽ không được lặp lại trong các phần sau nữa. Mặt khác, theo quan điểm sư phạm, cũng không thể giải thích mọi điều ngay từ bộ thích nghi đầu tiên vì như vậy sẽ gây khó khăn cho người đọc mới làm quen với STL. Giải pháp đưa ra ở đây là trong mỗi bộ thích nghi lại giải thích một ít và kèm ví dụ minh họa cụ thể cho người đọc dễ nắm bắt, sau đó cuối chương sẽ có mục tổng hợp lại.

4.3.4.1. binder1st

`binder1st` là bộ thích nghi dùng để chuyển một đối tượng hàm hai đối thành một đối tượng hàm một đối. Cụ thể, nếu `f` là một đối tượng của lớp `binder1st`, thì `f(x)` sẽ trả về đối tượng hàm `F(c,x)` trong đó `c` là hằng số, *F là đối tượng hàm hai đối thuộc KHÁI NIỆM adaptable binary function.* Cả `F` và `c` được truyền cho cấu tử của `f` khi khởi tạo. KHÁI NIỆM Adaptable Binary Function sẽ được nói rõ trong mục 3.5, từ góc độ sử dụng ta chỉ cần biết rằng tất cả các đối tượng hàm hai đối mà STL cung cấp đều thuộc KHÁI NIỆM này và do vậy, có thể sử dụng chúng với `binder1st`.

Ta xem xét ví dụ sau để hiểu rõ hơn về `binder1st`, ở đây đối tượng hàm `less` của STL làm nhiệm vụ so sánh hai số. Khi gọi toán tử `less(x,y)` ta có kết quả `true` nếu `x < y` và ngược lại. Bây giờ, giả sử trong chương trình ta cần một đối tượng hàm mệnh đề một đối số có toán tử so sánh một giá trị hằng c với đối số. Cho dù mỗi lần gọi ta truyền cho `x` giá trị `c` ta cũng không thể sử dụng `less` vì `less` là đối tượng hàm hai đối số. Tuy nhiên, nếu sử dụng `binder1st` với `less` ta sẽ có được đối tượng hàm một đối số như yêu cầu.

```
less<int> F;
int c = 5;
binder1st<less<int> > five_less_than(F,c);
```

```
cout << "5 nho hon 6? " << (five_less_than(6) ? "Dung" : "Sai")
<< endl;
cout << "5 nho hon 4? " << (five_less_than(4) ? "Dung" : "Sai")
<< endl;
```

```
5 nho hon 6? Dung
5 nho hon 4? Sai
```

Trong chương trình trên, ta khai báo một đối tượng hàm mệnh đề một đối `five_less_than` là đối tượng thuộc lớp `binder1st<less<int> >`. Đoạn mã `binder1st<less<int> >` cho biết ta sẽ tạo ra một đối tượng hàm một đối từ đối tượng hàm hai đối `less<int>`. (Ở đây, `less<int>` được cung cấp như tham số, khuôn hình cho `binder1st`. Nếu không cung cấp tham số khuôn hình này sẽ gây lỗi vì `binder1st` là lớp khuôn hình, muốn sử dụng phải cung cấp tham số khuông hình). Để sử dụng đối tượng hàm một đối đó, phải cung cấp các tham số cho cấu tử của nó: `five_less_than(F,c)` trong đó, F là một đối tượng của `less<int>` và c là một số nguyên.

Cách viết như trên là để cho rõ hơn, song khi đã quen, ta có thể viết như sau cho ngắn gọn:

```
binder1st<less<int> > five_less_than(less<int> (), 5);
```

`five_lessless_than` có thể truyền vào cho hàm dưới dạng tham số như chương trình sau:

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

template <class InputIterator,class Predicate>
void print_if(InputIterator first,InputIterator last,Predicate
pred)
{
    for (InputIterator i = first;i != last; i++)
    {
        if (pred(*i) == true)
            cout << *i << " ";
    }
}
void main()
```

```
{
    vector<int> v;
    ostream_iterator<int> oi(cout," ");

    v.push_back(10); v.push_back(4); v.push_back(15);
    v.push_back(2); v.push_back(-1); v.push_back(8);
    cout << "v: "; copy(v.begin(),v.end(),oi); cout << endl;

    // tao doi tuong ham menh de 1 doi so
    binder1st<less<int> > five_less_than(less<int> (), 5);
    // su dung doi tuong ham trong ham
    cout << "Cac so lon hon 5: ";
    print_if(v.begin(),v.end(),five_less_than);
}
```

```
v: 10 4 15 2 -1 8
Cac sơ lon hon 5: 10 15 8
```

Hàm `print_if` chỉ in các số trong khoảng `[first,last)` nếu số đó thỏa mãn điều kiện `pred`. Nghĩa là với mọi bộ duyệt `i` thuộc `[first,last)` nếu `pred(*i) == true` thì sẽ in `*i` ra màn hình. Như vậy, đối số cho `print_if` ngoài các bộ duyệt `first` và `last` còn có một đối số là đối tượng hàm mệnh đề một đối. Sau khi dùng `binder1st` để tạo ra đối tượng hàm một đối `five_less_than` ta đã sử dụng đối tượng hàm này như trong chương trình:

```
print_if(v.begin(),v.end(),five_less_than);
```

Thực tế khi sử dụng, nhất là đối với các giải thuật của STL, người ta dùng một hàm tiện lợi hơn `binder1st` mà vẫn cho kết quả tương tự. Thay vì dùng `binder1st` người ta dùng hàm `bind1st` một cách trực tiếp như sau:

```
print_if(v.begin(),v.end(),bind1st(less<int> (),5));
```

Câu lệnh trên tương đương với:

```
binder1st<less<int> > five_less_than(less<int> (), 5);
print_if(v.begin(),v.end(),five_less_than);
```

Có thể thấy cách viết này chỉ cần một dòng lệnh trong khi nếu dùng `binder1st` phải thêm một dòng lệnh khởi tạo. Tuy nhiên, đối với người mới làm quen, cách viết này lại không rõ ràng bằng cách dùng `binder1st`. Khi khởi tạo đối tượng hàm bằng `binder1st`, qua cách đặt tên người ta có thể

hình dung ra mục đích hoặc ý nghĩa của đối tượng hàm đó. Khi nhìn vào câu lệnh `bind1st(less<int> (),5)` người mới làm quen có thể sẽ không hiểu được nó có ý nghĩa gì. Thực chất hàm `bind1st` cũng nhận đầu vào hai tham số gồm đối tượng hàm `F` và hằng `c` và trả về đối tượng hàm `f` sao cho `f(x)` trả về `F(c,x)`. Chương trình có thể viết lại như sau:

```
void main()
{
    vector<int> v;
    ostream_iterator<int> oi(cout," ");

    v.push_back(10); v.push_back(4); v.push_back(15);
    v.push_back(2); v.push_back(-1); v.push_back(8);
    cout << "v: "; copy(v.begin(),v.end(),oi); cout << endl;

    cout << "Cac so lon hon 5: ";
    print_if(v.begin(),v.end(),bind1st(less<int> (),5));
}
```

4.3.4.2. binder2nd

Bộ thích nghi `binder2nd` cũng tương tự như `binder1st`. Nếu `f` là một đối tượng của lớp `binder2nd` thì `f(x)` trả về `F(x,c)` trong đó *F là đối tượng hàm hai đối số thuộc KHÁI NIỆM Adaptable Binary Function*, `c` là hằng số. `F` và `c` được truyền cho cấu tử của `binder2nd` khi khởi tạo đối tượng mới. Có thể thấy `binder2nd` và `binder1st` đúng như tên gọi chỉ khác nhau ở thứ tự của `x` và `c`. `binder1st` cụ thể hóa tham số thứ nhất trong `F(x,y)` thành hằng số, còn `binder2nd` cụ thể hóa tham số thứ hai thành hằng số. Thông thường, nếu không sử dụng đối tượng của `binder2nd` nhiều lần thì người ta dùng hàm `bind2nd` vì nó tiện lợi hơn. Ví dụ dùng `bind2nd` trong chương trình trên (chương trình minh họa `bind1st`):

```
    cout << "Cac so nho hon 5: ";
    print_if(v.begin(),v.end(),bind2nd(less<int> (),5));
```

```
Cac so nho hon 5: 4 2 -1
```

Có thể thấy là chỉ cần thay `bind1st` bằng `bind2nd` là ta có kết quả hoàn toàn ngược lại. Điều này có thể giải thích dễ dàng vì `bind2nd` cụ thể hóa tham số thứ hai thành hằng số trong khi `bind1st` cụ thể hóa tham số thứ nhất. Do vậy, trong `print_if`, khi dùng `bind1st` ta sẽ gọi kiểm tra `less(5,x)` còn khi dùng `bind2nd` ta sẽ gọi `less(x,5)`.

Nếu không dùng `bind2nd` ta có thể dùng trực tiếp `binder2nd` để tạo ra một đối tượng và truyền vào cho `print_if`. (Cách làm tương tự như đối với `binder1st`)

```
cout << "Cac so nho hon 5: ";
binder2nd<less<int> > five_greater_than(less<int> (),5);
print_if(v.begin(),v.end(),five_greater_than);
```

Rõ ràng nếu không dùng lại `five_greater_than` nhiều lần thì dùng trực tiếp `bind2nd` sẽ đơn giản hơn. Điều quan trọng là *khi dùng* `binder2nd` *với đối tượng hàm nào thì phải cung cấp đối tượng hàm đó như tham số khuôn hình khi khởi tạo*. Ví dụ nếu viết như sau trình biên dịch sẽ báo lỗi:

```
binder2nd five_greater_than(less<int> (),5);
```

Điều này cũng đúng cho tất cả các bộ thích nghi khác về đối tượng hàm.

4.3.4.3. unary_negate

Đối tượng hàm `unary_negate` là một bộ thích nghi trên các đối tượng hàm thuộc KHÁI NIỆM Adaptable Predicate. Giả sử có `pred` là một đối tượng của đối tượng hàm Predicate thuộc KHÁI NIỆM Adaptable Predicate, `f` là một đối tượng của `unary_negate<Predicate>`. Khi đó, `f(x) == !pred(x)`. Có nghĩa là các đối tượng của `unary_negate` thực hiện phép phủ định logic các đối tượng hàm tương ứng. `pred` sẽ được đưa vào như tham số cho cấu tử của `f`, còn Predicate thì được cung cấp như tham số khuôn hình cho `unary_negate`.

Ví dụ, ta có đối tượng hàm `odd` thuộc KHÁI NIỆM Adaptable Predicate như sau:

```
class odd : public unary_function<int, bool>
{
public:
    odd () {}
    bool operator () (int n) const
    {
        return (n % 2) == 1;
    }
};
```

Đối tượng hàm trên kiểm tra tính chất lẻ của một số nguyên. odd được thừa kế từ lớp unary_function<int,bool> để thuộc KHÁI NIỆM Adaptable Predicate. Chú ý rằng dù một đối tượng hàm có toán tử gọi hàm một đối trả về kiểu bool (tức là thuộc KHÁI NIỆM Predicate) mà không thuộc KHÁI NIỆM Adaptable Predicate thì cũng không sử dụng được với unary_negate (xem mục 4.3.5). Khi đó, nếu muốn có một đối tượng hàm even kiểm tra tính chất chẵn của số nguyên thì ta không cần phải viết lại một lớp tương tự như lớp odd nữa mà chỉ cần dùng unary_negate như sau:

```
unary_negate<odd> even(odd());
```

Lúc này, lời gọi odd(3) sẽ cho kết quả true, trong khi even(3) sẽ cho kết quả false. Cần chú ý là trong khi cài đặt odd thì toán tử gọi hàm của odd được khai báo là *hàm thành phần hằng* nhờ từ khóa const phía sau tên hàm. Nếu không có từ khóa const, trình biên dịch sẽ báo lỗi. Xin xem thêm mục 4.3.5 để biết rõ hơn.

Ví dụ sau minh họa cho unary_negate. Trong chương trình có sử dụng đối tượng hàm odd ở trên và print_if trong ví dụ về binder1st và binder2nd.

```
void main()
{
     vector<int> v;
     ostream_iterator<int> oi(cout," ");

     v.push_back(10); v.push_back(34); v.push_back(15);
     v.push_back(21); v.push_back(11); v.push_back(8);
     cout << "v: "; copy(v.begin(),v.end(),oi); cout << endl;

     cout << "Cac so chan: ";
     print_if(v.begin(),v.end(),odd());
     cout << "\nCac so le: ";
     unary_negate<odd> even(odd());
      print_if(v.begin(),v.end(),even);
}
```

```
v: 10 34 15 21 11 8
Cac so chan: 15 21 11
Cac so le: 10 34 8
```

Tất cả các bộ thích nghi trên đối tượng hàm đều có một hàm tiện lợi tương ứng và unary_negate cũng thế. Nếu không sử dụng even nhiều lần

trong chương trình thì hai dòng lệnh

```
unary_negate<odd> even(odd());
print_if(v.begin(),v.end(),even);
```

có thể viết gọn lại nhờ hàm `not1` như sau:

```
print_if(v.begin(),v.end(),not1 (odd()));
```

Hàm `not1` chỉ nhận một tham số là đối tượng hàm thuộc KHÁI NIỆM Adaptable Predicate và trả về đối tượng hàm khác phủ định lại đối tượng hàm trên. Chú ý rằng đối tượng hàm do `not1` trả lại chỉ sinh ra khi chạy chương trình. Ta không thể viết tường minh một lớp, chẳng hạn AdaptablePred, rồi dùng trong chương trình như sau:

```
AdaptablePred even = not1(odd());
```

`not1` chỉ dùng với các hàm dạng `print_if` hoặc các khuôn hình giải thuật của STL sẽ nhắc đến trong chương sau. Đây là một điểm bất lợi của `not1` cũng như của `bind1st`, `bind2nd` hay các hàm tương ứng với các bộ thích nghi trên đối tượng hàm khác. Tuy nhiên, hiếm khi ta dùng đối tượng hàm một cách trực tiếp mà hay dùng với giải thuật và như vậy `not1`, `bind1st`,... vẫn được sử dụng thường xuyên.

4.3.4.4. binary_negate

Tương tự như `unary_negate`, `binary_negate` thực hiện phép phủ định logic đối với các đối tượng hàm thuộc KHÁI NIỆM Adaptable Binary Predicate. Cụ thể, nếu `f` là đối tượng của lớp `binary_negate<Predicate>` thì `f(x,y) == !pred(x,y)` trong đó `Predicate` là đối tượng hàm thuộc KHÁI NIỆM Adaptable Binary Predicate, `pred` là đối tượng của lớp `Predicate` và được truyền vào cho cấu tử của `binary_negate<Predicate>` khi khởi tạo `f`. Xét chương trình minh họa sau.

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

```
template <class T>
class abs_greater : public binary_function<T,T,bool>
{
public:
     abs_greater() {}
     bool operator() (const T& v1,const T& v2) const
     {
            T t1 = v1; T t2 = v2;
            if (v1 < 0) t1 = -v1;
            if (v2 < 0) t2 = -v2;
            return t1 > t2;
     }
};
void main()
{
     vector<int> v;
     ostream_iterator<int> oi(cout," ");
     v.push_back(10); v.push_back(-34); v.push_back(-15);
     v.push_back(21); v.push_back(11); v.push_back(8);
     cout << "v: "; copy(v.begin(),v.end(),oi); cout << endl;
     // Sap xep day so giam dan theo tri tuyet doi
     sort(v.begin(),v.end(),abs_greater<int> ());
     cout << "v: "; copy(v.begin(),v.end(),oi); cout << endl;
     // Sap xep day so tang dan theo tri tuyet doi
     binary_negate<abs_greater<int> > abs_less(abs_greater<int>
 ());
     sort(v.begin(),v.end(),abs_less);
     cout << "v: "; copy(v.begin(),v.end(),oi); cout << endl;
}
```

```
v: 10 -34 -15 21 11 8
v: -34 21 -15 11 10 8
v: 8 10 11 -15 21 -34
```

Đối tượng hàm `abs_greater` là đối tượng hàm cho phép so sánh lớn hơn về giá trị tuyệt đối. Đây là đối tượng hàm thuộc KHÁI NIỆM Adaptable Binary Predicate vì được thừa kế từ lớp `binary_function` (xem mục 4.3.5). Trong chương trình, ta dùng một khuôn hình giải thuật `sort` của STL để sắp xếp các số trong `vector` theo thứ tự quy định bởi một đối tượng hàm so sánh (ở đây là `abs_greater` và `abs_less`). Khi dùng `sort` với `abs_greater` ta được dãy sắp xếp theo thứ tự giảm dần về trị tuyệt đối. Khi dùng `sort` với `abs_less` ta lại được dãy có tính chất ngược lại. Như vậy, có thể thấy nhờ dùng `binary_negate` ta đã tiết kiệm được công sức không

phải viết thêm một đối tượng hàm `abs_less` tương tự như `abs_greater` nữa mà tạo ra ngay từ `abs_greater`.

Tương ứng với `binary_negate` ta có hàm `not2` mà trong nhiều trường hợp, sử dụng `not2` sẽ thuận tiện hơn. Khi dùng `not2` phải truyền tham số là một đối tượng hàm thuộc KHÁI NIỆM Adaptable Binary Predicate. Ví dụ dùng `not2` thay cho `binary_negate`:

```
sort(v.begin(),v.end(),not2(abs_greater<int> ()));
```

4.3.4.5. unary_compose

Cần chú ý là `unary_compose` và `binary_compose` không có trong STL chuẩn được Ansi C++ công nhận. Tuy nhiên trong bản STL phân phối bởi SGI (Silicon Graphic Inc) lại có hai bộ thích nghi này và xét thấy chúng cũng rất tiện lợi nên chúng tôi muốn đưa ra để giới thiệu. Chúng tôi đã biên dịch thử trên các trình biên dịch quen thuộc và mới thấy có g++ và gcc trên RedHat Linux 7.2 do sử dụng STL của SGI nên hỗ trợ hai bộ thích nghi này. Tuy nhiên, vẫn có thể sử dụng hai bộ thích nghi này với các trình biên dịch khác theo hai cách sau. Cách thứ nhất thay vì dùng bộ thư viện STL có sẵn trong trình biên dịch, hãy dùng trực tiếp bộ thư viện STL do chúng tôi cung cấp trong đĩa CD đi kèm quyển sách này. Đây là bộ thư viện của SGI nên có sẵn hai bộ thích nghi `unary_compose` và `binary_compose`. Cách thứ hai bạn có thể sao chép nguyên mẫu của hai bộ thích nghi chúng tôi cho dưới đây (hoặc tìm trong đĩa CD đi kèm) ghép vào tệp chứa các đối tượng hàm của trình biên dịch bạn đang dùng (Ví dụ trong VC6.0 hay Visual.NET là tệp functional ứng với dẫn hướng biên dịch #include <functional>). Chúng tôi đã làm theo cách thứ hai với bộ Visual.NET và kết quả vẫn đúng như khi biên dịch bằng g++ trên Linux. Sau đây là nguyên mẫu của hai bộ thích nghi và các hàm tiện lợi tương ứng.

```
// unary_compose and binary_compose (extensions, not part of the
standard).

template <class _Operation1, class _Operation2>
class unary_compose
  : public unary_function<typename _Operation2::argument_type,
                          typename _Operation1::result_type>
{
protected:
  _Operation1 __op1;
```

```
  _Operation2 __op2;
public:
  unary_compose(const _Operation1& __x, const _Operation2& __y)
    : __op1(__x), __op2(__y) {}
  typename _Operation1::result_type
  operator()(const typename _Operation2::argument_type& __x)
const {
    return __op1(__op2(__x));
  }
};

template <class _Operation1, class _Operation2>
inline unary_compose<_Operation1,_Operation2>
compose1(const _Operation1& __op1, const _Operation2& __op2)
{
  return unary_compose<_Operation1,_Operation2>(__op1, __op2);
}

template <class _Operation1, class _Operation2, class
_Operation3>
class binary_compose
  : public unary_function<typename _Operation2::argument_type,
                          typename _Operation1::result_type> {
protected:
  _Operation1 _M_op1;
  _Operation2 _M_op2;
  _Operation3 _M_op3;
public:
  binary_compose(const _Operation1& __x, const _Operation2& __y,
                 const _Operation3& __z)
    : _M_op1(__x), _M_op2(__y), _M_op3(__z) { }
  typename _Operation1::result_type
  operator()(const typename _Operation2::argument_type& __x)
const {
    return _M_op1(_M_op2(__x), _M_op3(__x));
  }
};

template <class _Operation1, class _Operation2, class
_Operation3>
inline binary_compose<_Operation1, _Operation2, _Operation3>
compose2(const _Operation1& __op1, const _Operation2& __op2,
         const _Operation3& __op3)
{
  return binary_compose<_Operation1,_Operation2,_Operation3>
    (__op1, __op2, __op3);
}
```

unary_compose là một bộ thích nghi cho phép thực hiện phép hợp các hàm. Trong đại số, bạn đã biết có những hàm hợp kiểu như g(f(x)f(x)) và unary_compose cho phép thực hiện tương tự nhưng trên các đối tượng hàm của C++. Cụ thể, nếu f và g đều là đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function, thì có thể dùng unary_compose tạo ra đối tượng hàm h sao cho h(x) tương đương với f(g(x)). Muốn sử dụng, f và g cần được truyền vào cho cấu tử khi khởi tạo h, đồng thời kiểu của f và g được dùng như tham số khuôn hình cho unary_compose. Ví dụ sau minh họa cách sử dụng unary_compose. Ta cài đặt hai đối tượng hàm plus_c và multiply_c đều là các đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function. plus_c(x) thực hiện phép cộng x + c trong khi multiply_c(x) thực hiện phép nhân x*c.

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

template <class T>
class plus_c : public unary_function<T,T>
{
private:
    T c;
public:
    plus_c(T constant) : c(constant) {}
     T operator()(T x) const
    {
          x += c;
          return x;
    }
};
template <class T>
class multiply_c : public unary_function<T,T>
{
private:
    T c;
public:
    multiply_c(T constant) : c(constant) {}
    T operator()(T x) const
    {
          x *= c;
          return x;
```

```
    }
};
void main()
{
    plus_c<int> p(5);  // Khoi tao doi tuong voi hang c gan bang
5
    multiply_c<int> m(3);// Khoi tao doi tuong voi hang c bang 3
    cout << m(p(2)) << endl; // Thuc hien (2+5)*3
    unary_compose<multiply_c<int>,plus_c<int> > h(m,p);
    cout << h(2) << endl; // Thuc hien (2+5)*3
}
```

```
21
21
```

Để thực hiện phép tính `(2+5)*3`, trong chương trình sử dụng hai cách: một là trực tiếp bằng `m(p(2))`, hai là khai báo đối tượng hàm hợp `h` của `m` và `p` rồi gọi `h(2)`. Để khai báo `h`, trước tiên phải cung cấp `plus_c<int>` và `multiply_c<int>` như tham số khuôn hình cho `unary_compose`, sau đó truyền `m` và `p` cho cấu tử khởi tạo `h`. Cả hai cách đều cho cùng kết quả là giá trị của phép tính `(2+5)*3 = 21`. Bạn đọc có thể hỏi rằng tại sao phải sử dụng một cách rắc rối như thế trong khi chỉ cần viết đơn giản `m(p(2))` cũng đã có kết quả tương tự. Tuy nhiên, `unary_compose` được thiết kế không chỉ để sử dụng trực tiếp như trên mà mục đích chính là sử dụng với các hàm, đặc biệt là các giải thuật trong STL. Ví dụ, ta có thể thực hiện lần lượt `plus_c` và `multiply_c` lên một dãy số bằng khuôn hình giải thuật `transform` như sau:

```
    plus_c<int> p(5);
    multiply_c<int> m(3);
    unary_compose<multiply_c<int>,plus_c<int> > h(m,p);

    int a[6] = {1,2,3,4,5,6};
    transform(a,a+6,ostream_iterator<int> (cout," "),h);
```

```
18 21 24 27 30 33
```

Rõ ràng, đối số cuối cùng cho `transform` là một đối tượng hàm một đối. Nếu không sử dụng `unary_compose` ta không còn cách nào khác để tạo một đối tượng hàm như yêu cầu. `unary_compose` cũng có một hàm `compose1` tương ứng để sử dụng thuận tiện hơn. Ví dụ trên có thể viết lại bằng `compose1` như sau:

```
    plus_c<int> p(5);
    multiply_c<int> m(3);

    int a[6] = {1,2,3,4,5,6};
    transform(a,a+6,ostream_iterator<int> (cout," 
"),compose1(m,p));
```

Rõ ràng, cách viết trên ngắn gọn hơn một chút khi không phải khai báo h. Nhưng thực tế, việc khai báo ấy đã ẩn trong hàm compose1 rồi. (Nếu muốn, bạn có thể xem nguyên mẫu của compose1 ngay sau nguyên mẫu của unary_compose đã đưa ra ở trên).

Có thể nói rằng dùng các bộ thích nghi một cách kết hợp ta có thể tạo ra vô số những đối tượng hàm mới. Ta sẽ chứng minh điều này qua việc tạo ra đối tượng hàm hợp có ý nghĩa như trên, mà không cần cài đặt thêm plus_c hay multiply_c nữa. Đoạn chương trình sau có kết quả hoàn toàn giống hai đoạn chương trình trước, nhưng không sử dụng plus_c hay multiply_c mà chỉ sử dụng các đối tượng hàm có sẵn của STL cùng với bộ thích nghi binder1st.

```
    int a[6] = {1,2,3,4,5,6};
    transform(a,a+6,ostream_iterator<int> (cout," "),
          compose1(bind1st(multiplies<int> (),3), // doi tuong
ham 1
                   bind1st(plus<int> (),5)));     // doi tuong
ham 2
```

Thực tế ta đã sử dụng ghép nối các hàm tương ứng với các bộ thích nghi để tạo ra một đối tượng hàm hợp tương tự như h trong ví dụ đầu tiên:

```
compose1(bind1st(multiplies<int> (),3),bind1st(plus<int> (),5)));
```

Trước tiên, bind1st(plus<int> (),5) tạo ra một đối tượng hàm tương tự như plus_c<int>p(5), bind1st (multiplies<int()>(), 3) tạo ra một đối tượng hàm kiểu như multiply_c<int> m(3). Ta đã biết binder1st thực hiện việc cụ thể hóa tham số thứ nhất của một đối tượng hàm hai tham số. Ở trên, bind1st cụ thể hóa tham số thứ nhất là 5 đối với plus<int> để có đối tượng hàm thực hiện phép cộng 5+x và cụ thể hóa tham số thứ nhất là 3 đối với multiplies<int> để có đối tượng hàm thực hiện phép nhân 3*x. Sau đó, compose1 làm nốt nhiệm vụ còn lại là hợp hai đối tượng hàm đó lại. Chú ý rằng vì phép cộng và phép nhân có tính chất giao

hoán nên nếu ở trên ta có dùng bind2nd thay cho bind1st thì vẫn cho kết quả tương tự vì x+5 cũng như 5+x. Với các phép không có tính giao hoán như so sánh lớn hơn, nhỏ hơn thì sử dụng bind1st hay bind2nd là điều cần phải xét đến.

4.3.4.6. binary_compose

Như đã nhắc đến khi giới thiệu về bộ thích nghi unary_compose, bộ thích nghi binary_compose cũng không có trong STL chuẩn của ANSI C++. Tuy nhiên, chúng tôi vẫn muốn giới thiệu ở đây vì tính tiện lợi của bộ thích nghi này.

Trong đại số, ta hay gặp các hàm hợp kiểu như f(g1(x),g2(x)) và binary_compose là một bộ thích nghi cho phép thực hiện phép hợp tương tự như vậy trên các đối tượng hàm. Cụ thể, nếu f là một đối tượng hàm thuộc KHÁI NIỆM Adaptable Binary Function, g1, g2 là hai đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function thì có thể dùng binary_compose để tạo ra một đối tượng hàm h sao cho h(x) tương đương f(g1(x),g2(x)). Để tạo ra h từ binary_compose ta cũng phải lần lượt khai báo các tham số khuôn hình và truyền f, g1, g2 cho cấu tử của h như đối với unary_compose. Nếu đã quen với unary_compose có lẽ không cần nói nhiều nữa và xem xét ví dụ sau minh họa compose2, một hàm tiện lợi để sinh các đối tượng thay cho binary_compose.

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

void main()
{
    int a[6] = {1,2,3,4,5,6};
    transform(a,a+6,ostream_iterator<int> (cout," "),
          compose2(multiplies<int> (),            // doi tuong
ham 1
                   bind1st(plus<int> (),5),    // doi tuong ham
2
                   bind2nd(minus<int> (),5)));// doi tuong ham
3

}
```

```
-24 -21 -16 -9 0 11
```

Hàm `transform` trong ví dụ trên thực hiện phép tính `(x+5)*(x-5)` lên dãy `1,2,3,4,5,6` và in kết quả ra màn hình. Trước tiên `bind1st` tạo ra một đối tượng hàm thực hiện phép cộng `x+5`, `bind2nd` tạo ra đối tượng hàm thực hiện phép trừ `x-5`. Sau đó `compose2` thực hiện phép hợp từ hai đối tượng hàm trên với đối tượng hàm `multiplies` có sẵn trong STL để được tích `(x+5)*(x-5)`. Lưu ý là phép trừ không có tính chất giao hoán nên để tạo đối tượng hàm thực hiện `x-5` ta phải dùng `bind2nd`, nếu dùng `bind1st` ta sẽ có kết quả ngược lại là `(x+5)*(5-x)`.

4.3.4.7. pointer_to_unary_function

Đây là bộ thích nghi dùng để tương thích ngược với những hàm có sẵn trong C++ truyền thống như `abs`, `fabs`, `sqrt`,...hoặc các hàm ta đã viết từ trước nhưng không muốn mất thời gian nâng cấp lên thành đối tượng hàm. Từ đầu chương ta đã nói các con trỏ hàm cũng có thể coi là các đối tượng hàm và do vậy, có thể được dùng ở những nơi cần đến đối tượng hàm như một số ví dụ ta đã chỉ ra. Ví dụ một con trỏ hàm khai báo như sau: `T (*f)(A)` sẽ tương ứng với một đối tượng hàm thuộc KHÁI NIỆM Unary Function. (Người đọc cần biết rõ về con trỏ hàm, chúng tôi không nhắc lại về khái niệm này. Khai báo như trên sẽ được đọc như sau: `f` là con trỏ tới hàm nhận một tham số kiểu `A` và có kiểu trả về `T`). Tuy nhiên, tại những nơi chỉ sử dụng các đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function, con trỏ trên lại không sử dụng được. Đó là lý do tại sao cần đến `pointer_to_unary_function`, bộ thích nghi này làm nhiệm vụ chuyển các con trỏ hàm thành các đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function.

Ví dụ, sử dụng hàm giá trị tuyệt đối `abs` với khuôn hình giải thuật `transform` trong STL như sau sẽ không cần đến `pointer_to_unary_function`, bởi lẽ yêu cầu đối số của `transform` chỉ là đối tượng hàm thuộc KHÁI NIỆM Unary Function:

```
int a[5] = {2,-14,5,15,-4};
transform(a,a+5,ostream_iterator<int> (cout," "),abs);
```

```
2 14 5 15 4
```

Trong đoạn mã trên, `abs` được truyền vào `transform` như một con trỏ hàm, tức là một đối tượng hàm. `transform` áp dụng `abs` lên mảng nguyên `a` và in kết quả ra màn hình. Tuy nhiên, nếu muốn in ra dãy có cùng trị tuyệt

đối nhưng mang dấu âm, ta cần phải *kết hợp* hàm abs với đối tượng hàm negate bằng hàm bộ thích nghi compose1. Tuy nhiên, compose1 chỉ nhận đối số là các đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function nên cần phải chuyển abs từ con trỏ hàm thành đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function.

```
int a[5] = {2,-14,5,15,-4};
pointer_to_unary_function<int,int> abs_functor(abs);
   transform(a,a+5,ostream_iterator<int> (cout," "),
         compose1(negate<int> (), abs_functor));
```

```
-2 -14 -5 -15 -4
```

Hàm abs nhận tham số đầu vào kiểu int và kiểu trả về cũng là kiểu int nên <int,int> được cung cấp cho pointer_to_unary_function như tham số khuôn hình còn con trỏ abs được truyền vào cho cấu tử của abs_functor khi khởi tạo. Như vậy, ta đã tạo ra abs_functor có ý nghĩa giống như abs, nhưng lại là đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function. Do đó, có thể truyền abs_functor vào cho compose1. Ta cũng có thể dùng hàm ptr_fun thay cho pointer_to_unary_function để chương trình viết ngắn gọn hơn như sau:

```
int a[5] = {2,-14,5,15,-4};
transform(first, last, first,
     compose1(negate<double>, ptr_fun(fabs)));
```

Hàm ptr_fun nhận đối số là một con trỏ tới hàm một đối số và trả về đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function tương ứng với con trỏ hàm đó.

4.3.4.8. pointer_to_binary_function

Đây là đối tượng hàm tương tự như pointer_to_unary_function, nhưng áp dụng cho các con trỏ tới hàm hai tham số. pointer_to_binary_function dùng để chuyển các hàm hai tham số bình thường thành các đối tượng hàm thuộc KHÁI NIỆM Adaptable Binary Function. Để sử dụng, ta phải cung cấp ba tham số khuôn hình cho pointer_to_binary_function, gồm kiểu của hai tham số và kiểu trả về của hàm muốn chuyển thành đối tượng hàm. Tất nhiên, con trỏ hàm vẫn phải được truyền vào cho cấu tử khởi tạo. Ví dụ sau sẽ minh họa cách dùng

pointer_to_binary_function và hàm nạp chồng ptr_fun tương ứng. Chú ý là hàm ptr_fun ứng với pointer_to_binary_function khác với hàm ptr_fun ứng với pointer_to_unary_function. Đây là hai hàm nạp chồng.

```
int a[5] = {2,-14,5,15,4};
ostream_iterator<int> oi(cout," ");
pointer_to_binary_function<double,double,double>
pow_functor(pow);
transform(a,a+5,oi,bind2nd(pow_functor,2));
cout << endl;
transform(a,a+5,oi,bind2nd(ptr_fun(pow),2));
```

```
4 196 25 225 16
4 196 25 225 16
```

Ta có hai hàm transform thực hiện cùng một nhiệm vụ là tính bình phương các số trong mảng a, rồi ghi kết quả ra màn hình, nhưng một hàm dùng pointer_to_binary_function, một hàm dùng ptr_fun. Chú ý là do tham số và giá trị trả về của pow (hàm lũy thừa) đều là kiểu double, nên ta phải cung cấp double như tham số khuôn hình cho pointer_to_binary_function. Ngoài ra, ta cũng dùng hàm bind2nd chứ không dùng bind1st, vì ta muốn tính x^2 chứ không phải 2^x.

4.3.5. Xây dựng các đối tượng hàm adaptable

Ở các mục trước, ta đã xét tới những đối tượng hàm dạng adaptable. Đó là các đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function, Adaptable Binary Function. Tuy nhiên, chưa có một định nghĩa chính thức về các KHÁI NIỆM này, mà mới chỉ giới thiệu là các đối tượng hàm thuộc KHÁI NIỆM adaptable (và cũng chỉ có các đối tượng hàm dạng này), được dùng cho các bộ thích nghi của đối tượng hàm.

Một cách tổng quát, các đối tượng hàm dạng adaptable là các đối tượng hàm trong cài đặt có từ khóa typedef để định nghĩa kiểu của giá trị trả về (result_type) và kiểu của các tham số (argument_type, first_argument_type,... nếu có). Tương ứng với ba KHÁI NIỆM chính của đối tượng hàm là Generator, Unary Function và Binary Function ta có các KHÁI NIỆM Adaptable Generator, Adaptable Unary Function và Adaptable Binary Function.

4.3.5.1. Adaptable Generator

Trên thực tế, ta không sử dụng đến các đối tượng hàm thuộc KHÁI NIỆM Adaptable Generator. Có thể thấy điều này qua mục 4.3.4. Các bộ thích nghi được giới thiệu trong mục 4.3.4 đều nhận đối số là các đối tượng hàm thuộc hai KHÁI NIỆM Adaptable Unary Function và Adaptable Binary Function. Hơn nữa, tất cả các đối tượng hàm trong STL cũng đều chỉ thuộc hai KHÁI NIỆM này. KHÁI NIỆM Adaptable Generator chỉ có ý nghĩa khi xây dựng thư viện. Tuy nhiên, nếu bạn đọc vẫn muốn quan tâm tới đối tượng hàm thuộc KHÁI NIỆM này thì xin tham khảo hai mục 4.3.5.2 và 4.3.5.3.

4.3.5.2. Adaptable Unary Function

Có hai cách để tạo ra một đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function. Cách thứ nhất, chính quy, định nghĩa kiểu trả về và kiểu tham số trong khai báo lớp bằng `typedef`. Cách thứ hai đơn giản hơn, cho phép đối tượng hàm thừa kế từ lớp `unary_function`. Đây là cách ta đã làm trong một số ví dụ về bộ thích nghi ở mục 4.3.4. `unary_function` là một lớp khuôn hình chỉ có một nhiệm vụ là định nghĩa kiểu trả về và kiểu tham số trong khai báo. Do vậy, khi một đối tượng hàm thừa kế từ `unary_function`, nó sẽ thuộc KHÁI NIỆM Adaptable Unary Function.

Ví dụ sau sẽ minh họa cách tạo đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function bằng `typedef`.

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

template <class Arg,class Res>
class inverse
{
public:
     typedef Arg argument_type;
     typedef Res result_type;
     inverse() {}
     Res operator()(Arg x) const
     {
            return 1/x;
     }
};
```

```
void main()
{
     float a[5] = {1.0,2.0,3.0,4.0,5.0};
     transform(a,a+5,ostream_iterator<float> (cout,"\n"),
            compose1(negate<float> (),inverse<float,float> ()));
}
```

```
-1
-0.5
-0.333333
-0.25
-0.2
```

Do có hai dòng khai báo `typedef` định nghĩa kiểu trả về và kiểu tham số nên `inverse` là đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function. Vì vậy, `inverse` có thể được dùng với `compose1` và `negate` để tạo ra đối tượng hàm `-1/x` như trong chương trình. Theo cách thứ hai, ta không cần hai dòng `typedef` mà chỉ cần cho `inverse` thừa kế từ lớp `unary_function<Arg,Res>` như sau:

```
template <class Arg,class Res>
class inverse : public unary_function<Arg,Res>
{
public:
     inverse() {}
     Res operator()(Arg x) const
     {
            return 1/x;
     }
};
```

Cần lưu ý tới thứ tự của hai tham số khuôn hình cho `unary_function:Arg` trước, `Res` sau. Vì đây là thứ tự trong nguyên mẫu của `unary_function`, nếu bị đảo ngược trình dịch có thể sẽ báo lỗi. Nếu cài đặt theo cách thứ nhất, thứ tự này không quan trọng, vì ta tự định nghĩa bằng `typedef`. Nhưng nếu thừa kế từ `unary_function`, thì thứ tự này lại quan trọng.

Chú ý thứ hai là toán tử gọi hàm của đối tượng hàm phải được khai báo và cài đặt là *hàm thành phần hằng* bằng từ khóa `const` đi sau tên hàm. Khai báo như vậy để các đối tượng hằng của inverse có thể gọi được toán tử gọi hàm của chúng. Nếu toán tử gọi hàm của `inverse` không được cài đặt là hàm

thành phần hằng thì với một đối tượng hằng như sau: `const inverse<float,float> inv` ta sẽ không thể gọi được `inv(x)`. Bạn đọc có thể thắc mắc rằng: vậy đối tượng hằng `inverse` ở đâu trong chương trình trên? Câu trả lời: nó chính là tham số cho hàm `compose1` vì `compose1` được khai báo như sau:

```
template <class _Operation1, class _Operation2>
inline unary_compose<_Operation1,_Operation2>
compose1(const _Operation1& __op1, const _Operation2& __op2)
{
  return unary_compose<_Operation1,_Operation2>(__op1, __op2);
}
```

Có thể thấy cả `__op1` và `__op2` đều là các đối tượng hằng và trong chương trình, `__op1` chính là một đối tượng của kiểu `inverse<float,float>`.

Ghi nhớ hai quy tắc sau khi cài đặt các toán tử gọi hàm (cũng như hàm thành phần) của một lớp:

- Nếu hàm thành phần không làm thay đổi nội dung của lớp thì nên khai báo là hàm thành phần hằng để có thể được gọi từ cả đối tượng hằng lẫn đối tượng không phải hằng.
- Nếu hàm thành phần không phải hàm thành phần hằng thì đối tượng có thể gọi được hàm thành phần chỉ là các đối tượng không hằng.

4.3.5.3. Adaptable Binary Function

Cách tạo ra một đối tượng hàm thuộc KHÁI NIỆM Adaptable Binary Function cũng tương tự như các cách tạo ra một đối tượng hàm thuộc KHÁI NIỆM Adaptable Unary Function. Có hai cách là dùng `typedef` khai báo các kiểu trả về, tham số một cách trực tiếp hoặc thừa kế từ lớp có sẵn là `binary_function`. Ví dụ sau đưa ra cách dùng `typedef`.

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

```
template <class Arg1,class Arg2,class Res>
class plus_inverse
{
public:
     typedef Arg1 first_argument_type;
     typedef Arg2 second_argument_type;
     typedef Res result_type;
     plus_inverse() {}
     Res operator() (const Arg1& x,const Arg2& y) const
     {
            return 1/(x+y);
     }
};

void main()
{
     float a[4] = {0.5, 1.5, 2.5, 3.5};
     float b[4] = {1.5, 3.5, 0.5, -0.5};
     ostream_iterator<float> oi(cout," ");
     cout << "a: " ; copy(a,a+4,oi); cout << endl;
     cout << "b: " ; copy(b,b+4,oi); cout << endl;

     cout << "a+b: "; transform(a,a+4,b,oi,plus<float> ());
      cout << endl;
     cout << "1/(a+b): ";
      transform(a,a+4,b,oi,plus_inverse<float,float,float> ());
}
```

```
a: 0.5 1.5 2.5 3.5
b: 1.5 3.5 0.5 -0.5
a+b: 2 5 3 3
1/(a+b): 0.5 0.2 0.333333 0.333333
```

Cài đặt lại `plus_inverse` bằng cách thừa kế từ `binary_function`:

```
template <class Arg1,class Arg2,class Res>
class plus_inverse : public binary_function<Arg1,Arg2,Res>
{
public:
     plus_inverse() {}
     Res operator() (const Arg1& x,const Arg2& y) const
     {
            return 1/(x+y);
     }
};
```

Giống như với `inverse`, `operator()` của `plus_inverse` phải được khai báo là hàm thành phần hằng và thứ tự của `Arg1`, `Arg2` và `Res` là quan trọng. Xem xét kỹ, người đọc có thể thấy là `plus` cũng là đối tượng hàm thuộc KHÁI NIỆM Adaptable Binary Function như `plus_inverse`, nhưng khi sử dụng chỉ cần cung cấp một tham số khuôn hình, trong khi đó, với `plus_inverse` ta phải cung cấp ba tham số khuôn hình. Đó là vì với `plus`, STL quy định cả hai tham số và kiểu trả về phải cùng một kiểu. Có thể làm điều này với `plus_inverse`:

```
template <class T>
class plus_inverse : public binary_function<T,T,T>
{
public:
     plus_inverse() {}
     T operator() (const T& x,const T& y) const
     {
            return 1/(x+y);
     }
};
```

4.3.6. Bộ thích nghi trên hàm thành phần của lớp

Mục 4.3.4.5 đã giới thiệu về hai bộ thích nghi giúp chuyển các hàm bình thường thành các đối tượng hàm adaptable. Bây giờ ta xét tới các bộ thích nghi giúp chuyển hàm thành phần của một lớp thành đối tượng hàm tương ứng để có thể sử dụng được tại những nơi yêu cầu đối tượng hàm.

4.3.6.1. mem_fun_t và mem_fun_ref_t

Giả sử `X` là một lớp có hàm thành phần `Result X::f()` (nghĩa là hàm thành phần không đối số có kết quả trả về là `Result`). Khi đó, nếu muốn sử dụng `X::f()` tại những nơi đòi hỏi đối tượng hàm, ta phải cần đến `mem_fun_t` hoặc `mem_fun_ref_t`.

Để sử dụng `mem_fun_t`, phải cung cấp hai tham số khuôn hình là lớp `X` và `Result` là kiểu trả về của `X::f()`. Cấu tử của `mem_fun` cần được truyền vào một tham số là con trỏ tới hàm thành phần `X::f`, trong khi đó `operator()` lại cần tham số là con trỏ `x` của kiểu `X*`. Nếu `F` là đối tượng của `mem_fun<Result,X>`, sau khi khởi tạo với con trỏ hàm `X::f()`, toán

tử gọi hàm `F(x)` sẽ tương đương với `x->f()`. Chương trình sau minh họa cách dùng `mem_fun`.

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

class Object
{
public:
    virtual void say_hi() = 0;
};
class ObjChild1 : public Object
{
public:
    void say_hi()
    {
        cout << "Xin chao! Toi la doi tuong con 1!" << endl;
    }
};
class ObjChild2 : public Object
{
public:
    void say_hi()
    {
        cout << "Xin chao! Toi la doi tuong con 2!" << endl;
    }
};
template <class InputIter,class UnaryFunc>
void my_for(InputIter first,InputIter last,UnaryFunc f)
{
    for (InputIter i = first; i != last; i++)
        f(*i);
}

void main()
{
    vector<Object*> v;
    v.push_back(new ObjChild1);
    v.push_back(new ObjChild2);
    v.push_back(new ObjChild1);

    mem_fun_t<void,Object> mfun(Object::say_hi);
    // Dung mfun truc tiep
```

```
    Object* o = *(v.begin());
    mfun(o);
    // Dung mfun voi my_for
    my_for(v.begin(),v.end(),mfun);
}
```

```
Xin chao! Toi la doi tuong con 1!
Xin chao! Toi la doi tuong con 1!
Xin chao! Toi la doi tuong con 2!
Xin chao! Toi la doi tuong con 1!
```

Dễ thấy, sử dụng `mem_fun_t` đối với hàm thành phần tương tự như sử dụng `pointer_to_unary_function` với hàm. Trước tiên, phải cung cấp hai tham số khuôn hình là `void`, `Object` cho `mem_fun`, sau đó cung cấp con trỏ hàm thành phần `Object::say_hi` cho cấu tử của `mfun`. Cuối cùng, sử dụng `mfun` một cách trực tiếp hoặc như tham số cho hàm `my_for` đã được chỉ rõ ở trong chương trình.

Ứng với `mem_fun_t`, STL có hàm `mem_fun`, nhận đầu vào là con trỏ hàm thành phần `X::f()` và trả về đối tượng hàm tương ứng. Ví dụ trên có thể viết lại với `mem_fun`:

```
    // Dung ham mem_fun
    my_for(v.begin(),v.end(),mem_fun(Object::say_hi));
```

```
Xin chao! Toi la doi tuong con 1!
Xin chao! Toi la doi tuong con 2!
Xin chao! Toi la doi tuong con 1!
```

Bộ thích nghi `mem_fun_ref_t` cũng tương tự như `mem_fun_t`, nhưng dùng cho các đối tượng của kiểu `X`, chứ không phải `X*`. Do vậy, nếu `F` là đối tượng của `mem_fun_ref_t<Result,X>`, `x` là đối tượng của X thì `F(x)` sẽ tương đương với `x.f()`. Chú ý rằng `x` không phải con trỏ, nên ta không thể dùng kỹ thuật đa hình của hướng đối tượng đối với `mem_fun_ref_t` như trong ví dụ trên. Nếu muốn áp dụng `mem_fun_ref_t` trên một dãy các đối tượng, tất cả các đối tượng đó phải thuộc cùng một kiểu. Ta không thể dùng lại `vector v` trong ví dụ trên với `mem_fun_ref_t` vì `v` chứa các đối tượng thuộc hai kiểu khác nhau, mà phải tạo ra `vector` mới `v2` chỉ chứa các đối tượng cùng kiểu `ObjChild2` (Hơn nữa, cũng không được là kiểu `ObjChild2*`). Xem xét đoạn chương trình sau:

```
vector<ObjChild2> v2;
v2.push_back( ObjChild2());
v2.push_back( ObjChild2());
v2.push_back( ObjChild2());

mem_fun_ref_t<void,ObjChild2> mfr(ObjChild2::say_hi);
my_for(v2.begin(),v2.end(),mfr);
```

```
Xin chao! Toi la doi tuong con 2!
Xin chao! Toi la doi tuong con 2!
Xin chao! Toi la doi tuong con 2!
```

Chú ý là x được truyền vào cho toán tử `operator()` của F dưới dạng tham chiếu, nên nếu hàm thành phần f có làm thay đổi nội dung của x, sự thay đổi đấy vẫn có hiệu lực sau lời gọi `F(x)`. (Điều này cũng đúng với `mem_fun_t` vì x được truyền vào dưới dạng con trỏ). Ví dụ trên không cho thấy rõ điều này. Nếu muốn, bạn đọc có thể viết một chương trình khác để kiểm nghiệm.

`mem_fun_ref_t` cũng có hàm tương ứng là `mem_fun_ref` với một tham số là con trỏ hàm thành phần và trả về một đối tượng hàm ứng với hàm thành phần đó. Hai dòng lệnh cuối trên có thể viết gọn lại thành:

```
my_for(v2.begin(),v2.end(),mem_fun_ref(ObjChild2::say_hi));
```

4.3.6.2. mem_fun1_t và mem_fun1_ref_t

Đây là hai bộ thích nghi tương tự như hai bộ thích nghi trên nhưng cho các hàm thành phần có một đối số. Giả sử lớp X có hàm thành phần `Result X::f(Arg)`, x là con trỏ kiểu `X*`. Lớp `mem_fun1_t<Result,Arg>` có cấu tử nhận con trỏ hàm `X::f` làm tham số. F là đối tượng của `mem_fun1_t<Result,X,Arg>`, a thuộc kiểu `Arg`, khi đó `F(x,a)` sẽ tương đương với `x->f(a)`. Như vậy, tại những nơi yêu cầu đối tượng hàm hai tham số ta có thể sử dụng `mem_fun1_t` để tạo ra đối tượng hàm cần thiết từ hàm thành phần một biến của lớp. Chú ý rằng tham số khuôn hình cho `mem_fun1_t` là ba tham số bao gồm kiểu trả về, lớp X và kiểu tham số. Ví dụ, dùng `mem_fun1_t` được cho dưới đây. Đây là chương trình in bảng cửu chương.

```
#include <iostream>
#include <functional>
```

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

template <class T>
class print_row
{
private:
    T c;
public:
    print_row(T x) : c(x) {}
    void print(T X)
    {
        cout << c << " x " << X      // vi du 2 x 5 = 10
             << " = " << c*X << endl;
    }
};
template <class T,class InputIter,class BinaryFunc>
void my_transform (T x, InputIter first, InputIter last,
BinaryFunc F)
{
    for (InputIter i = first; i != last; i++)
    {
        F(x,*i);
    }
}
void main()
{
    print_row<int>* pr = new print_row<int>(2);
    int a[9] = {1,2,3,4,5,6,7,8,9};
mem_fun1_t<void,print_row<int>,int> mf(print_row<int>::print);
    my_transform(pr,a,a+9,mf);
}
```

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
```

Thay vì khai báo mf một cách "rắc rối" như trên, ta có thể dùng hàm mem_fun để có được đối tượng hàm tương tự như mf làm tham số cho my_transform. Lưu ý rằng theo đúng như các quy tắc đặt tên, đáng lẽ tên phải là hàm mem_fun1 nhưng STL khuyến cáo nên dùng hàm nạp chồng n_fun vì mem_fun1 sẽ bị bỏ đi trong các phiên bản sau. Hai câu lệnh cuối

viết gọn lại như sau:

```
my_transform(pr,a,a+9,mem_fun(print_row<int>::print);
```

mem_fun1_ref_t cũng tương tự như mem_fun1_t. Giả sử lớp X có hàm thành phần Result X::f(Arg), x là con trỏ kiểu X, lớp mem_fun1_t<Result,Arg> có cấu tử nhận con trỏ hàm X::f làm tham số, F là đối tượng của mem_fun1_t<Result,X,Arg>, a thuộc kiểu Arg, khi đó F(x,a) sẽ tương đương với x->f(a). Sự khác nhau giữa mem_fun1_ref_t và mem_fun1_t chỉ là mem_fun1_t làm việc với các con trỏ đối tượng trong khi mem_fun1_ref_t làm việc với tham chiếu đối tượng. Như vậy, với mem_fun1_t ta có thể dùng trong kỹ thuật đa hình của hướng đối tượng nhưng với mem_fun1_ref_t thì không. Bạn đọc tự viết ví dụ với mem_fun1_ref_t.

4.3.7. Bảng các đối tượng hàm có trong STL

Bảng sau liệt kê tất cả các đối tượng hàm có trong STL cho người đọc tiện tra cứu.

Tên hàm	Khởi tạo	operator()
plus	plus<T> p;	p(T arg1,T arg2)
minus	minus<T> m;	m(T arg1,T arg2)
multiplies	multiplies<T> m;	m(T arg1,T arg2)
divides	divides<T> d;	d(T arg1,T arg2)
Modulus	modulus<T> m;	d(T arg1,T arg2)
negate	negate<T> n;	n(T arg)
equal_to	equal_to<T> e;	e(T arg1,T arg2)
Not_equal_to	not_equal_to<T> ne;	ne(T arg1,T arg2)
greater	greater<T> g;	g(T arg1,T arg2)
less	less<T> l;	l(T arg1,T arg2)
greater_equal	greater_equal<T> ge;	ge(T arg1,T arg2)
less_equal,	less_equal<T> le;	le(T arg1,T arg2)
logical_and	logical_and<T> la;	la(T arg1,T arg2)
logical_or	logical_or<T> lo;	lo(T arg1,T arg2)
logical_not	logical_not<T> ln;	ln(T arg)
unary_function	unary_function<Arg, Result> uf	uf(Arg a)
binary_function	binary_function<Arg1, Arg2, Result> bf	bf(Arg1 a1, Arg2 a2)

unary_compose	unary_compose< AdaptableUnaryFunction1, AdaptableUnaryFunction2>	
binary_compose	binary_compose< AdaptableBinaryFunction, AdaptableUnaryFunction1, AdaptableUnaryFunction2>	
unary_negate	unary_negate <AdaptablePredicate>	
binary_negate	binary_negate <AdaptableBinaryPredicate>	
binder1st	binder1st <AdaptableBinaryFunction>	
binder2nd	binder2nd <AdaptableBinaryFunction>	
pointer_to_unary_function	pointer_to_unary_function <Arg, Result>	
pointer_to_binary_function	pointer_to_binary_function <Arg1, Arg2, Result>	

4.4. Tóm tắt

4.4.1. Ghi nhớ

Đối tượng hàm là đối tượng có toán tử gọi hàm `operator()`. Khi sử dụng đối tượng hàm trong chương trình, ta gọi đến đối tượng nhưng hình thức giống như gọi đến hàm.

Đối tượng hàm trong lập trình hướng đối tượng bằng C++ thay thế cho con trỏ hàm trong lập trình C truyền thống. Điểm ưu việt của đối tượng hàm so với con trỏ hàm là tính khái lược. Với đặc tính này, đối tượng hàm trở thành một thành phần quan trọng trong lập trình khái lược.

Đối tượng hàm bản thân là đối tượng nên có khả năng đóng gói dữ liệu. Khi nói về sử dụng đối tượng hàm tức là sử dụng toán tử gọi hàm của nó. Toán tử gọi hàm là thành phần quan trọng nhất của một đối tượng hàm. Một đối tượng hàm có thể nạp chồng nhiều toán tử gọi hàm và được sử dụng một cách rất linh hoạt.

Đối tượng hàm có thể được sử dụng trực tiếp trong chương trình bằng

cách gọi toán tử gọi hàm, nhưng đây không phải là ứng dụng hay dùng của đối tượng hàm. Thay vào đó, đối tượng hàm thường được sử dụng như tham số trực tiếp hoặc tham số khuôn hình cho một hàm. Các giải thuật hoặc các bộ chứa thường sử dụng đối tượng hàm như tham số khuôn hình.

Ba KHÁI NIỆM cơ bản cho đối tượng hàm là Generator, Unary Function và Binary Function tương ứng với các đối tượng hàm có toán tử gọi hàm không đối, một đối và hai đối số.

Thư viện STL cài đặt sẵn một số đối tượng hàm cơ bản và hay dùng nhất gồm: các phép toán số học, các phép toán logic và các phép toán so sánh. Ngoài ra còn có các bộ thích nghi trên đối tượng hàm. Với các bộ thích nghi, ta có thể kết hợp để có được những đối tượng hàm phức tạp từ những đối tượng hàm đơn giản.

4.4.2. Các lỗi hay gặp khi lập trình

Nhầm lẫn giữa cấu tử và toán tử gọi hàm của một đối tượng hàm vì hình thức của chúng giống nhau.

Sử dụng toán tử gọi hàm của một đối tượng hàm trước khi khởi tạo. Lỗi này hay gặp khi sử dụng đối tượng hàm để truyền tham số trực tiếp cho hàm.

4.5. Bài tập

Bài tập 4.1 Xem lại ví dụ trong mục 4.3.4.6 minh họa hàm compose2. Viết lại ví dụ đó bằng sử dụng `unary_compose` trực tiếp.

Chương 5

GIẢI THUẬT

Mục đích chương này:

- Giới thiệu về giải thuật trong lập trình khái lược
- Tìm hiểu các giải thuật có trong STL
- Biết cách sử dụng các giải thuật hiệu quả với bộ chứa và đối tượng hàm

5.1. Giải thuật

5.1.1. Khái niệm về thuật toán

Thuật toán hay giải thuật không còn là khái niệm mới mẻ với người lập trình. Thuật toán là một thủ tục thực hiện từng bước một để giải quyết một lớp bài toán nào đó. Nhắc đến thuật toán, nói chung là phải nhắc đến cấu trúc dữ liệu như trong cuốn sách nổi tiếng của Niclaus Wirth, người sáng lập ra ngôn ngữ Pascal: "Chương trình = Cấu trúc dữ liệu + Giải thuật". Rất nhiều thuật toán và cấu trúc dữ liệu đã được đưa ra kể từ khi ra đời ngôn ngữ lập trình cấp cao đầu tiên.

Trong những quyển sách trình bày về thuật toán, một thuật toán thường được mô tả bằng lưu đồ hoặc bằng ngôn ngữ giả mã và do vậy, xuất hiện thuật ngữ cài đặt thuật toán. Cài đặt thuật toán là công việc của người lập trình để chuyển những mô tả thuật toán bằng lưu đồ hoặc bằng ngôn ngữ giả mã thành những đoạn mã trên một ngôn ngữ cụ thể và làm việc với những dữ liệu cụ thể.

Đó là thuật toán và những khái niệm liên quan trong lập trình truyền thống. Một người lập trình tối thiểu cần nắm được những thuật toán cũng như cấu trúc dữ liệu cơ bản, nhằm tránh việc phát minh lại những gì đã có đồng thời phát triển nhanh ứng dụng của mình.

5.1.2. Giải thuật trong lập trình khái lược

Giải thuật là một khái niệm không mới trong lập trình khái lược, vì thực chất nó là một hàm thực hiện một giải thuật nào đó theo hướng tiếp cận khái

lược. Điều đó có nghĩa là giải thuật được thể hiện bởi hàm có tính linh hoạt cao, làm việc được trên nhiều kiểu dữ liệu. Trong các thư viện, nó được thể hiện dưới dạng các khuôn hình hàm, do vậy ta gọi tắt là khuôn hình giải thuật. Khuôn hình giải thuật trong lập trình khái lược có những điểm giống và khác so với các hàm triển khai thuật toán trong lập trình truyền thống. Trước hết, khuôn hình giải thuật cũng được đề ra nhằm giải quyết một lớp bài toán chung hay gặp. Chúng cũng làm việc trên những dữ liệu trong lập trình khái lược là các bộ chứa. Điểm khác biệt lớn nhất là tính linh hoạt của các khuôn hình giải thuật mà các hàm triển khai các thuật toán thông thường không có. Nói cách khác, hàm giải thuật được tham số hóa một cách triệt để, để có thể làm việc trên những dữ liệu kiểu bất kì, ngoài ra còn có thể có tính mềm dẻo nhất định trong xử lý. Do đã mang tính khái lược nên việc mô tả hàm giải thuật dưới dạng lưu đồ hay giả mã là không cần thiết. Khuôn hình giải thuật trong lập trình khái lược được cài đặt thành một hàm cụ thể trong một ngôn ngữ xác định (ADA, C++) và người lập trình có thể sử dụng được luôn.

Sau đây ta nghiên cứu việc xây dựng một khuôn hình giải thuật khái lược đơn giản để hiểu rõ hơn về khuôn hình giải thuật trong lập trình khái lược và ưu điểm của nó so với thuật toán thông thường. Xét một thuật toán tìm kiếm duyệt từ đầu đến cuối được cài đặt bằng ngôn ngữ C++:

```
const int* my_search (int* first, int* last, const int& val) {
   while(first != last && *first != val)
       ++first;
   return first;
}
```

Hàm trên duyệt tìm `val` trong khoảng `[first,last)` nếu tìm thấy sẽ trả về con trỏ tới vị trí đầu tiên tìm được, nếu không sẽ trả về `last`.

```
int x[5] = {1,5,3,5,4};
cout << "Tim 5 trong day: "
     << ((my_search(&x[0],&x[5],5) != &x[5])? "Co" : "Khong")
     << endl;
cout << "Tim 7 trong day: "
     << ((my_search(&x[0],&x[5],7) != &x[5])? "Co" : "Khong")
     << endl;
```

```
Tim 5 trong day: Co
Tim 7 trong day: Khong
```

Rõ ràng, thuật toán trên chỉ được cài đặt trên mảng các số nguyên. Để thuật toán thực hiện được với dữ liệu kiểu bất kì, ta chuyển hàm trên thành

dạng hàm khuôn hình:

```
template <class T>
const T* my_search (T* first, T* last, const T& val) {
   while(first != last && *first != val)
       ++first;
   return first;
}
```

Thuật toán trên tuy đã được khuôn hình hóa nhưng vẫn còn điểm chưa được tối ưu. Việc sử dụng con trỏ tới kiểu giá trị T có thể thay bằng sử dụng kiểu bộ duyệt. Khi đó, my_search được coi là một khuôn hình giải thuật. Cách khuôn hình giải thuật duyệt trên phần tử của dãy không phụ thuộc trực tiếp lên kiểu dữ liệu mà ẩn trong các đối tượng bộ duyệt. Như vậy, khuôn hình giải thuật được tách khỏi các bộ chứa mà chỉ thao tác thông qua các bộ duyệt.

```
template <class InputIterator,class T>
const InputIterator my_search(InputIterator first,InputIterator
last, const T& val)
{
     while (first != last && *first != val)
            ++first;
     return first;
}
```

Trong cài đặt giải thuật, ta chỉ cần hai thao tác cơ bản lên bộ duyệt là phép lấy tham chiếu * để nhận đối tượng trỏ tới bởi bộ duyệt và phép tăng ++ để nhận được bộ duyệt tiếp theo trong dãy. Do vậy, tham số cho khuôn hình giải thuật là bộ duyệt thuộc KHÁI NIỆM Input Iterator. Bây giờ, khuôn hình giải thuật đã làm việc tốt trên mảng nguyên thông thường cũng như trên các bộ chứa của STL.

```
// Sử dụng my_search với mảng nguyên
int x[5] = {1,5,3,5,4};
cout << "Tim 5 trong day: "
     << ((my_search(&x[0],&x[5],5) != &x[5])? "Co" : "Khong")
     << endl;
cout << "Tim 7 trong day: "
     << ((my_search(&x[0],&x[5],7) != &x[5])? "Co" : "Khong")
     << endl;
// Sử dụng my_search với vector
vector<int> v;
```

```
v.push_back(1);v.push_back(2);v.push_back(4);
cout << "Tim 2 trong vector: "
     << ((my_search(v.begin(),v.end(),2) != v.end())? "Co" :
"Khong")
     << endl;
```

```
Tim 5 trong day: Co
Tim 7 trong day: Khong
Tim 2 trong vector: Co
```

Nhờ tính khái lược của khuôn hình giải thuật và với bộ thư viện STL cung cấp sẵn những khuôn hình giải thuật tiện lợi nhất, người lập trình không còn phải cài đặt lại những thuật toán "kinh điển" nữa. Lấy ví dụ cụ thể với khuôn hình giải thuật tìm kiếm ở trên, trong lập trình truyền thống, ta sẽ phải cài đặt lại `my_search` mỗi khi muốn dùng nó trên một kiểu dữ liệu khác. Cho dù việc cài đặt lại có thể hoàn toàn tương tự không gây khó khăn nhưng cũng gây lãng phí thời gian và công sức. Với khuôn hình giải thuật `my_search` được thiết kế khái lược ta có thể dùng nó trên một kiểu dữ liệu bất kì, miễn là thỏa mãn một số yêu cầu nhất định cho tham số đầu vào. Đây chính là điểm hơn hẳn của lập trình khái lược giúp ta bỏ qua được những chi tiết phụ để tập trung duy nhất vào vấn đề cần giải quyết.

5.1.3. Sử dụng khuôn hình giải thuật trong STL

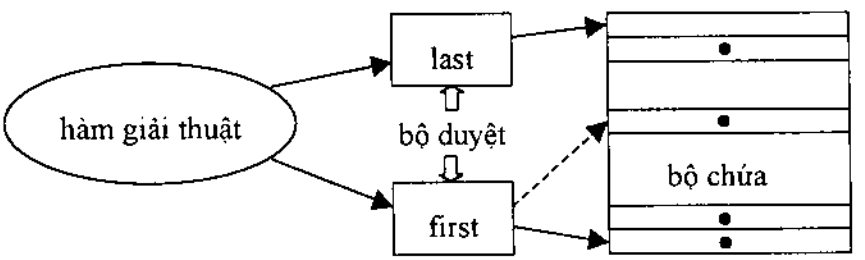
Nói đến khuôn hình giải thuật trong STL là nói đến các khuôn hình giải thuật khái lược. Chúng là các hàm khuôn hình sử dụng bộ duyệt để thao tác trên các bộ chứa nhằm thực hiện một mục đích nào đó. Bộ thư viện STL được xây dựng theo hướng tiếp cận *hướng giải thuật* với trung tâm là các khuôn hình giải thuật. Có thể thấy một nửa của STL là các khuôn hình giải thuật. Qua những chương trước, ta đã biết được một số khái niệm mới và quan trọng như bộ duyệt, bộ chứa và đối tượng hàm. Cả ba được đưa ra nhằm mục đích hỗ trợ để thiết kế nên các khuôn hình giải thuật khái lược nhất.

STL đưa ra một tập các khuôn hình giải thuật rất đa dạng và phong phú với mục đích giúp người lập trình nhanh chóng dễ dàng, sử dụng chúng ở mọi nơi cần đến trong chương trình. Do số lượng khuôn hình giải thuật nhiều, các khuôn hình giải thuật lại có tính khái lược có thể được áp dụng rất đa dạng, nên việc nhớ và sử dụng thành thạo chúng cũng đòi hỏi kinh nghiệm và một thời gian nhất định để làm quen. Tuy nhiên, khi đã trở nên quen thuộc với các khuôn hình giải thuật trong STL, ta sẽ có xu hướng sử dụng chúng ngày càng nhiều hơn. Đồng thời, ta cũng quen với xu hướng quan tâm tới nhiệm vụ đích

thực cần giải quyết hơn là quan tâm tới khuôn hình giải thuật của STL thực hiện nó như thế nào.

5.1.3.1. Sử dụng khuôn hình giải thuật với bộ duyệt và bộ chứa

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



Cơ chế làm việc của giải thuật được giải thích nhằm mục đích giúp người lập trình nếu muốn có thể tự viết các giải thuật khái lược của riêng mình. Tuy nhiên, nếu sử dụng các khuôn hình giải thuật có sẵn của STL ta chỉ cần đưa vào các bộ duyệt khuôn hình giải thuật đòi hỏi mà không cần quan tâm khuôn hình giải thuật sẽ thực hiện trên bộ chứa bằng cách nào.

Xét ví dụ với khuôn hình giải thuật `copy`:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator  first,
                    InputIterator  last,
                    OutputIterator result);
```

Hàm này thực hiện việc sao chép các phần tử từ khoảng `[first, last)` tới khoảng `[result, result + (last - first) )`. Khuôn hình giải thuật này ta đã sử dụng rất nhiều lần trong các ví dụ trước với mục đích hiển thị kết quả ra màn hình. Đây là một phong cách lập trình tốt. Thay vì dùng một vòng `for` và tại mỗi bước lặp lại viết kết quả ra màn hình, ta sao chép thẳng dữ liệu ra màn hình nhờ khuôn hình giải thuật `copy`. Theo khai

báo của copy, ta cần cung cấp tham số vào cho khuôn hình giải thuật là ba bộ duyệt trong đó first và last là hai bộ duyệt thuộc KHÁI NIỆM Input Iterator, còn result là bộ duyệt thuộc KHÁI NIỆM Output Iterator.

Có thể so sánh hai cách viết qua ví dụ dưới đây. Ví dụ này có sử dụng lớp person đã nhắc đến trong chương 4.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include "../person.h"
using namespace std;

void main ()
{
    person p1("Tran Vu","Truc");
    person p2("Dang Cong", "Kien");
    person p3("Tran Kim", "Chi");
    person p4("Cao Tran", "Kien");

    list<person> l;
    l.push_back(p1);l.push_back(p2);
    l.push_back(p3);l.push_back(p4);

    cout << "Su dung vong for" << endl;
    list<person>::iterator i;
    for (i = l.begin(); i != l.end(); i++)
          cout << *i << endl;

    cout << "Su dung thuat toan copy" << endl;
    copy(l.begin(),l.end(),ostream_iterator<person>(cout,
"\n"));

}
```

Có thể nhầm lẫn nếu cho rằng ostream_iterator<person>(cout, "\n") là một đối tượng hàm. Thực ra, đó là lời gọi tới cấu tử của một bộ duyệt đặc biệt là ostream_iterator. Chi tiết về ostream_iterator đã được nói đến trong chương 3. Ta có thể viết tường minh như sau nếu sử dụng copy nhiều lần:

```
ostream_iterator<person> oi(cout, "\n");
copy(l.begin(),l.end(),oi);
// ... su dung copy tren cac container khac van dung oi
```

Sau đây là kết quả chạy chương trình:

```
Su dung vong for
[Tran Vu Truc]
[Dang Cong Kien]
[Tran Kim Chi]
[Cao Tran Kien]
Su dung thuat toan copy
[Tran Vu Truc]
[Dang Cong Kien]
[Tran Kim Chi]
[Cao Tran Kien]
```

Có thể thấy cách dùng copy cũng cho cùng kết quả như cách dùng vòng for nhưng ngắn gọn và thuận tiện hơn. Tất nhiên, khuôn hình giải thuật copy không chỉ được sử dụng cho mục đích hiển thị kết quả nhưng ta sẽ nói đến chi tiết hơn về copy trong mục 5.2.2.1 của chương này. Bây, giờ ta xét một ví dụ nữa để minh họa thêm về cách sử dụng giải thuật với bộ chứa và bộ duyệt.

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

void main ()
{
    int a[6] = { 10, -30, 5, 40, 20 };
    int b[3] = { 5, 40, 20 };

    // Neu trinh dich ho tro member template function
    list<int> list_a(a, a+5);
    list<int> list_b(b, b+3);

    // Neu trinh dich khong ho tro, su dung cac dong sau
    // list<int> list_a, list_b;
    // list_a.insert(list_a.begin(), a, a+5);
    // list_b.insert(list_b.begin(), b, b+3);

    ostream_iterator<int> oi(cout, " ");

    cout << "List a: ";
    copy(list_a.begin(),list_a.end(),oi);

    cout << "\nList b: ";
```

```
    copy(list_b.begin(),list_b.end(),oi);

    // Thuat toan search tim vi tri giong nhau dau tien
    list<int>::iterator ia;
    ia = search(list_a.begin(), list_a.end(), list_b.begin(),
list_b.end());
    cout << "\nPhan tu giong nhau dau tien:" << *ia << endl
         << "Tai vi tri: "
         << ((int)distance(list_a.begin(),ia)+1) << endl;

    // Thuat toan equal kiem tra su giong nhau cua hai day
    cout << "Kiem tra lai:"
         << (equal(ia,list_a.end(),list_b.begin())? "Dung" :
"Sai")
         << endl;
}
```

Chương trình sử dụng hai khuôn hình giải thuật `search` và `equal` được STL cung cấp theo nguyên mẫu dưới đây:

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1,
                        ForwardIterator1 last1,
                        ForwardIterator2 first2,
                        ForwardIterator2 last2);
```

```
template <class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1,
           InputIterator1 last1,
           InputIterator2 first2);
```

Khuôn hình giải thuật `search` tìm dãy con trong dãy `[first1, last1)` giống với dãy `[first2, last2)` và trả về vị trí giống nhau đầu tiên. Ta phải truyền tham số vào là bốn bộ duyệt thuộc KHÁI NIỆM Forward Iterator khi sử dụng `search`.

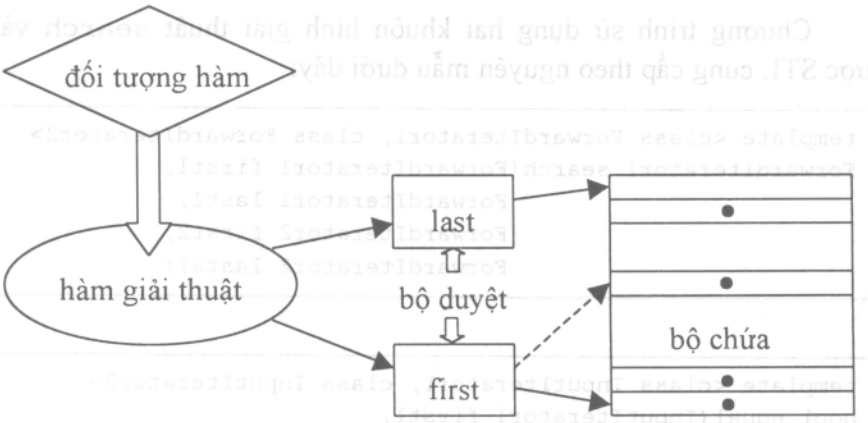
Khuôn hình giải thuật `equal` so sánh hai dãy `[first1, last1)` và `[first2, last2)` rồi trả về `true` nếu hai dãy giống nhau và `false` nếu ngược lại, trong đó `last2` được ngầm định bằng `first2 + (last1 - first1)`. Do vậy ta chỉ cần truyền tham số vào là ba bộ duyệt thuộc KHÁI NIỆM Forward Iterator khi sử dụng `equal`.

Kết quả chạy chương trình:

```
List a: 10 -30 5 40 20
List b: 5 40 20
Phan tu giong nhau dau tien:5
Tai vi tri: 3
Kiem tra lai:Dung
```

5.1.3.2. Sử dụng đối tượng hàm với khuôn hình giải thuật

Nhằm tăng thêm tính khái lược cho giải thuật, các đối tượng hàm được đưa vào các khuôn hình giải thuật dưới dạng tham số trực tiếp hoặc tham số khuôn hình. Như vậy, khuôn hình giải thuật sử dụng đối tượng hàm để thực hiện các thao tác bên trong và sử dụng bộ duyệt để giao tiếp bên ngoài với các bộ chứa.



Việc sử dụng đối tượng hàm như tham số trực tiếp hay tham số khuôn hình mỗi cách có những đặc điểm và ứng dụng riêng. Sau đây, ta sẽ xem xét cả hai cách sử dụng đó bằng những ví dụ nhỏ minh họa. Trước hết trở lại với khuôn hình giải thuật my_search trong mục 5.1.2.

```
template <class InputIterator,class T>
const InputIterator my_search(InputIterator first,InputIterator
last, const T& val)
{
    while (first != last && *first != val)
        ++first;
    return first;
}
```

Trong hàm này ta sử dụng phép so sánh mặc định là toán tử khác nhau (câu lệnh `*first != last`). Để cải thiện ta sẽ đưa vào một tham số nữa cho khuôn hình giải thuật là một đối tượng hàm thuộc KHÁI NIỆM Binary Predicate có nhiệm vụ so sánh giữa `*first` và `val`. Khi đó `my_search` có thể được dùng để tìm kiếm theo một tiêu chí bất kỳ.

```
template <class InputIterator,class BinaryPredicate,class T>
const InputIterator my_search(InputIterator first,
                              InputIterator last,
                              const T& val)
{
     BinaryPredicate comp;
     while (first != last && !comp(*first,val))
            ++first;
     return first;
}
```

Theo nguyên mẫu trên, đối tượng hàm được cung cấp cho `my_search` như một tham số khuôn hình. Việc khởi tạo hay sử dụng đối tượng hàm trong `my_search` như nào ta không cần quan tâm, chỉ cần cung cấp kiểu đối tượng hàm khi sử dụng `my_search` là đủ. Ví dụ, chương trình sau sử dụng một đối tượng hàm có toán tử gọi hàm là phép toán tìm đồng dư với modul 2 để tìm số chẵn đầu tiên trong dãy:

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

// Doi tuong ham dong du
class congruence
{
public:
     bool operator()(int n1, int n2)
     {
            return ((n1 % 2) == (n2 % 2));
     }
};
void main ()
{
     vector<int> v;
     v.push_back(1);v.push_back(3);v.push_back(2);v.push_back(5);

     typedef vector<int>::iterator IterInt;
     ostream_iterator<int> oi(cout, " ");
```

```
    cout << "Vector: " << endl;
    copy(v.begin(),v.end(),oi);

    cout << "\nTim so chan dau tien trong vector: "
         << *my_search<IterInt,congruence,int>
(v.begin(),v.end(),2)
         << endl;

}
```

```
Vector:
1 3 2 5
Tim so chan dau tien trong vector: 2
```

Tất nhiên, ta cũng có thể sử dụng các đối tượng hàm có sẵn trong STL như `less`, `greater`, `equal_to`,... hoặc một đối tượng hàm bất kì khác thuộc KHÁI NIỆM Binary Predicate. Cần lưu ý là đối tượng hàm đó phải có cấu tử không đối (mặc định hoặc không) để sử dụng trong `my_search`. Như trong ví dụ trên, đối tượng hàm `congruence` có cấu tử không đối mặc định nên có thể sử dụng được. Đây là điểm bất lợi của cách sử dụng đối tượng hàm như tham số khuôn hình vì chính sự che giấu việc khởi tạo đối tượng hàm bên trong khuôn hình giải thuật khiến việc sử dụng khuôn hình giải thuật không được linh hoạt. Người lập trình không biết được phải cung cấp đối tượng hàm nào phù hợp. Đặc biệt, khi đối tượng hàm có đóng gói dữ liệu ta không nên sử dụng cách này. Cũng vì lý do đó, tất cả các giải thuật của STL đều cài đặt theo cách thứ hai là sử dụng đối tượng hàm một cách trực tiếp như sau:

```
template <class InputIterator,class BinaryPredicate,class T>
const InputIterator my_search(InputIterator first,
                              InputIterator last,
                              BinaryPredicate comp,//truyen truc
tiep
                              const T& val)
{
    while (first != last && !comp(*first,val))
          ++first;
    return first;
}
```

Với cách sử dụng trực tiếp, tham số đối tượng hàm được truyền vào như đối tượng của một lớp, sau đó trong khuôn hình giải thuật, đối tượng này sẽ thực hiện một số thao tác trên các bộ duyệt truyền vào.Ta thấy đối tượng hàm

congruence chưa được khái lược trọn vẹn vì mới chỉ thực hiện phép toán đồng dư mặc định theo modul 2. Để congruence thực hiện được phép toán đồng dư modul bất kì, ta có hai cách giải quyết:

- Truyền thêm tham số thứ ba cho toán tử gọi hàm làm modul và thực hiện phép toán theo modul này:

```
// Doi tuong ham dong du
class congruence
{
public:
     bool operator()(int n1, int n2, int m)
     {
            return ((n1 % m) == (n2 % m));
     }
};
```

- Đóng gói modul vào dữ liệu và khi dùng phải khởi tạo modul:

```
// Doi tuong ham dong du
class congruence
{
public:
     // Cau tu co mot doi so
     congruence(int modul) : m(modul) {};

     bool operator()(int n1, int n2)
     {
            // Kiem tra tinh dong du theo modul m
            return ((n1 % m) == (n2 % m));
     }
     void setModul(int modul)
     {
            m = modul;
     }
private:
     int m; // Modul
};
```

Cách thứ nhất không khả thi vì theo nguyên mẫu của my_search, đối tượng hàm truyền vào phải thuộc KHÁI NIỆM Binary Predicate, trong khi đó congruence lại có toán tử gọi hàm ba đối. Nếu muốn sử dụng được my_search với congruence ta phải sửa lại my_search: truyền thêm tham số modul, thay đổi câu lệnh kiểm tra thành comp(*first,val,modul). Rõ ràng, cách làm này vừa rắc rối vừa không hiệu quả, vì bây giờ

my_search không thể làm việc được với các đối tượng hàm khác thuộc KHÁI NIỆM Binary Predicate của STL. Đến đây, ta hiểu rõ hơn tại sao các đối tượng hàm trong STL chỉ có các toán tử gọi hàm nhiều nhất hai tham số và không có khuôn hình giải thuật nào trong STL dùng đến đối tượng hàm có toán tử gọi hàm nhiều hơn hai tham số.

Cách làm thứ hai rõ ràng hiệu quả hơn và hoàn toàn tuân theo phong cách của STL. Khi sử dụng congruence ta cần khởi tạo trước và sau đó truyền vào my_search.

```
void main ()
{
    vector<int> v;
    v.push_back(11);v.push_back(13);v.push_back(12);v.push_back(
25);

    typedef vector<int>::iterator IterInt;
    ostream_iterator<int> oi(cout, " ");

    cout << "Vector: " << endl;
    copy(v.begin(),v.end(),oi);

    // Khoi tao doi tuong ham tinh dong du theo modul 2
    congruence congrc(2);

    cout << "\n\nTim so chan dau tien trong vector: "
            << *my_search (v.begin(),v.end(), congrc, 2)
            << endl;

    // Tinh dong du theo modul 5
    congrc.setModul(5);

    cout << "\nTim so dong du voi 3 theo modul 5: "
            << *my_search (v.begin(),v.end(), congrc, 3)
            << endl;
}
```

```
Vector:
11 13 12 25

Tim so chan dau tien trong vector: 12

Tim so dong du voi 3 theo modul 5: 13
```

Trong mục trước ta cũng đã nhắc đến khuôn hình giải thuật `equal` so sánh sự "giống nhau" giữa hai dãy. Nguyên mẫu của `equal` trong STL được cho như dưới đây:

```
template <class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1,
           InputIterator1 last1,
           InputIterator2 first2);
```

Nhưng đây chỉ là một hàm nạp chồng của `equal`. Trong STL còn có một hàm nạp chồng `equal` khác cho phép ta định nghĩa sự "giống nhau" theo một tiêu chí nào đó bằng một đối tượng hàm thuộc KHÁI NIỆM Binary Predicate:

```
template <class InputIterator1, class InputIterator2
          class BinaryPredicate>
bool equal(InputIterator1  first1,
           InputIterator1  last1,
           InputIterator2  first2,
           BinaryPredicate binary_pred);
```

Ví dụ, ta có thể so sánh sự giống nhau về tính chất chẵn lẻ của hai dãy nhờ sử dụng đối tượng hàm `congruence` ở trên:

```
void main ()
{
    int x[5] = {1,0,1,1,0};
    int y[5] = {3,4,5,7,8};
    int z[5] = {1,2,3,4,5};

    congruence congrc(2); // Doi tuong ham tinh dong du modul 2

    cout << "Tinh tuong dong chan le giua x va y: "
         << (equal(&x[0],&x[5],&y[0],congrc) ? "Tuong dong" :
"Khong tuong dong")
         << endl;

    cout << "Tinh tuong dong chan le giua x va z: "
         << (equal(&x[0],&x[5],&z[0],congrc) ? "Tuong dong" :
"Khong tuong dong")
         << endl;
}
```

```
Tinh tuong dong chan le giua x va y: Tuong dong
Tinh tuong dong chan le giua x va z: Khong tuong dong
```

Tóm lại, có thể thấy việc đưa thêm các toán tử gọi hàm vào làm tham số cho các khuôn hình giải thuật khiến các khuôn hình giải thuật càng mang tính khái lược hơn. Cùng một khuôn hình giải thuật ta có thể thao tác trên nhiều kiểu dữ liệu nhờ có bộ duyệt và bộ chứa, theo nhiều cách khác nhau thông qua đối tượng hàm. Mục 5.2 tiếp theo của chương sẽ trình bày tất cả các khuôn hình giải thuật sẵn có của STL cùng các ứng dụng của chúng trong lập trình khái lược. Người đọc sẽ dần làm quen với cách thức sử dụng khuôn hình giải thuật trong chương trình và sự tiện lợi của chúng.

5.2. Các giải thuật trong STL

Các giải thuật trong STL được chia thành ba lớp chính là các giải thuật làm đổi biến, các giải thuật không làm đổi biến và các giải thuật sắp xếp. Việc phân chia các giải thuật cũng có mức tương đối nhằm giúp người lập trình dễ nắm vững và tiện lợi trong tra cứu. Để sử dụng các giải thuật của STL cần thêm hướng dẫn biên dịch `#include <algorithm>` vào đầu mỗi chương trình.

Các giải thuật của STL sẽ lần lượt được giới thiệu trong các mục tiếp theo với một kiểu mẫu như sau:

1. Nguyên mẫu: đưa ra nguyên mẫu của khuôn hình giải thuật trong STL. Bạn đọc nên chú ý tới nguyên mẫu để nắm được các tham số khuôn hình và các tham số thực của khuôn hình giải thuật. Ví dụ, một nguyên mẫu như sau:

```
template <class ForwardIterator, class LessThanComparable>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const LessThanComparable& value);
```

   cho biết tham số khuôn hình cho khuôn hình giải thuật là hai kiểu thuộc các KHÁI NIỆM Forward Iterator và Less Than Comparable. Nếu bạn chưa quen thuộc với các KHÁI NIỆM cần cho khuôn hình giải thuật, hãy tra trong phụ lục. Phải nắm được KHÁI NIỆM ứng với mỗi khuôn hình giải thuật, ta mới sử dụng chúng một cách đúng đắn, nhất là khi làm việc với các kiểu do người dùng tự định nghĩa.

2. Giải thích: đưa ra giải thích ngắn gọn về giải thuật, cách sử dụng chúng trong chương trình.

3. Chương trình ví dụ: đây là các chương trình đơn giản được viết ra

nhằm mục đích giúp bạn đọc nắm rõ hơn cách sử dụng giải thuật. Tuy nhiên, khi bạn đọc đã quen có thể một số giải thuật, ở đây sẽ không đưa ra ví dụ nữa vì khuôn khổ hạn chế của quyển sách.

4. Các yêu cầu đặc biệt khi sử dụng: Các yêu cầu cần tuân theo khi sử dụng giải thuật, chủ yếu là các yêu cầu với các tham số của hàm. Ở đây chỉ đưa ra các yêu cầu đặc biệt. Một số yêu cầu mặc nhiên như *khi sao chép từ dãy thứ nhất sang dãy thứ hai, dãy thứ hai phải được cấp phát đủ chỗ chứa* chúng tôi sẽ không đưa ra, bạn đọc có thể tìm trong tham chiếu của STL.

5. Độ phức tạp của giải thuật.

5.2.1. Các giải thuật không làm đổi biến

Như ý nghĩa của tên gọi, các giải thuật không làm đổi biến khi thực hiện trên các bộ chứa sẽ không làm thay đổi các đối tượng được chứa trong bộ chứa. Các giải thuật dạng này chủ yếu là các hàm duyệt, tìm kiếm hay so sánh.

5.2.1.1. Khuôn hình giải thuật for_each

Nguyên mẫu hàm:

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator first, InputIterator last,
UnaryFunction f);
```

Khi giải quyết một bài toán, người lập trình rất hay gặp trường hợp phải thực hiện một thao tác nào đó lên một dãy đối tượng. Theo cách thông thường ta hay viết một vòng lặp `for`, `while`, ... và thực hiện thao tác đó. Trong STL, khuôn hình giải thuật `for_each` sẽ làm đơn giản hóa các công việc như vậy. Hàm này áp dụng đối tượng hàm f lên mỗi phần tử trong đoạn `[first,last)` . Nghĩa là tại mỗi lần lặp của `for_each`, f sẽ sử dụng toán tử gọi hàm với tham số là một phần tử của dãy. Giá trị trả về (nếu có) của f bị bỏ qua. Hàm `for_each` trả về đối tượng hàm sau khi nó đã được áp dụng lên tất cả các phần tử của dãy.

Như vậy, để giải quyết vấn đề trên cần đóng gói thao tác vào đối tượng hàm f, gọi hàm `for_each` với các tham số là f và các bộ duyệt trỏ đến vị trí đầu và vị trí sau cuối của dãy muốn áp dụng thao tác.

Ví dụ sau sẽ minh họa cách dùng `for_each` để in các số ra màn hình. Nếu bạn không quen với cách hiển thị bằng nạp chồng toán tử `operator>>` và dùng hàm `copy`, sử dụng cách sau với `for_each`.

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

template<class T> class print
{
public:
     print() : count(0) {}
     void operator() (T x)
     {
          cout << x << ' '; ++count;
     }
     int count;
};
int main()
{
     int A[] = {1, 4, 2, 8, 5, 7};
     const int N = sizeof(A) / sizeof(int);

     print<int> P = for_each(A, A + N, print<int>());
     cout << endl << "Da in " << P.count << " so." << endl;
```

```
}1 4 2 8 5 7
Da in 6 so.
```

Trước tiên cài đặt một đối tượng hàm `print` với toán tử gọi hàm thực hiện việc hiển thị. Sau đó, trong chương trình, mỗi lần cần in một dãy ta gọi hàm `for_each` với các tham số là vị trí đầu, vị trí sau cuối của dãy và một đối tượng của `print`. Chú ý tham số thứ hai là vị trí sau cuối của dãy chứ không phải vị trí cuối. Nếu dùng một trong các bộ chứa của STL ta có thể dùng hàm `end()` để có vị trí sau cuối. Nếu dùng mảng như trong ví dụ trên, ta có 6 phần tử. Như vậy, tham số thứ hai phải là `A[6]` tức là phần tử thứ 7 (không xác định) chứ không phải là `A[5]`.

Xét một ví dụ nữa tính tổng của các số chẵn trong dãy. Ta cài đặt một đối tượng hàm `evenSum` như sau:

```
class evenSum
{
private:
```

```
    int s;
public:
    evenSum() : s(0) {};
    ~evenSum() {};

    void operator() (int n);
    int getSum();
};
void evenSum::operator ()(int n)
{
    if ((n % 2) == 0)
        s += n;
}
int evenSum::getSum()
{
    return s;
}
```

Đối tượng hàm `evenSum` có thành phần riêng `s` để lưu tổng các số chẵn. Toán tử gọi hàm của `evenSum` có một đối số là `n`. Nếu `n` chẵn sẽ được cộng thêm vào `s`. Sử dụng `evenSum` trong chương trình với `for_each` như sau:

```
void main()
{
    evenSum mysum;
    int a[5] = { 1,2,4,5,6};

     cout << "Day so: " << endl;
    for_each(a, a + 5, print<int>());

     cout << "\nTong cac so chan trong day: " << endl;
     mysum = for_each(&a[0],&a[5],mysum);

     cout << mysum.getSum() << endl;
}
```

```
Day so:
1 2 4 5 6
Tong cac so chan trong day:
12
```

Ta sử dụng đối tượng hàm `print` trong ví dụ trước để hiển thị. Cần lưu ý tới cách truyền tham số chỉ vị trí trong hai lần sử dụng hàm `for_each`. Hai cách sử dụng đều tương đương nhau, nhưng cách thứ nhất "có vẻ" che dấu đi

tính chất "sau cuối" của tham số thứ hai. Tùy người lập trình quen sử dụng cách nào cũng được, nhưng trong những bài toán lớn, tốt nhất là nên sử dụng bộ chứa có sẵn của STL. Thứ hai, cách truyền đối tượng của evenSum nói riêng và các đối tượng hàm khác nói chung cho for_each là truyền theo kiểu tham biến. Do vậy, nếu không có câu lệnh:

```
mysum = for_each(&a[0],&a[5],mysum);
```

mà chỉ thực hiện:

```
for_each(&a[0],&a[5],mysum);
```

thì sau khi in ra ta sẽ có kết quả sai là 0. Lý do là mysum chỉ được truyền vào như tham biến, nên sau khi ra khỏi hàm for_each nó lấy lại giá trị trước khi truyền vào.

Độ phức tạp của khuôn hình giải thuật for_each là tuyến tính. Hàm thực hiện tất cả (last - firstfirst) lần hàm f.

Chú ý:

1. Theo như nguyên mẫu của for_each, hai tham số đầu là các biến thuộc kiểu có KHÁI NIỆM là Input Iterator. Tuy nhiên, trong chương trình ta dùng các địa chỉ mảng a, a+5. Điều này hoàn toàn được phép vì STL coi các địa chỉ mảng như các bộ duyệt để nhằm tương thích ngược.
2. Chúng ta luôn dùng đối tượng hàm *thật sự* làm tham số cho các khuôn hình giải thuật trong các chương trình minh họa. Tuy nhiên, như đã biết trong chương 4, các con trỏ hàm cũng được coi là các đối tượng hàm. Do vậy, chúng cũng có thể được sử dụng với các giải thuật. Điều này giúp bạn tận dụng được một số hàm có sẵn trong thư viện như abs, sin, cos,...
3. Bạn đọc sẽ thấy rằng các giải thuật được khuôn hình hóa rất cao. Mỗi giải thuật có ít nhất một tham số khuôn hình. Tuy nhiên, khi sử dụng người lập trình không nhất thiết phải cung cấp tham số khuôn hình mỗi lần gọi đến một khuôn hình giải thuật vì các hàm có cơ chế tự xác định kiểu. Dựa vào tham số cung cấp cho hàm, khuôn hình giải thuật sẽ tự xác định xem tham số có đúng với kiểu khai báo trong nguyên mẫu không. Do vậy, ta có thể viết:

```
for_each(&a[0],&a[5],mysum);
```

mà không phải viết là

```
for_each<int,evenSum> (&a[0],&a[5],mysum);
```

5.2.1.2. Khuôn hình giải thuật find

Nguyên mẫu hàm:

```
template<class InputIterator, class EqualityComparable>
InputIterator find(InputIterator first, InputIterator last,
                   const EqualityComparable& value);
```

Khuôn hình giải thuật `find` thực hiện việc tìm kiếm `value` trên một khoảng `[first,last)` và trả về bộ duyệt đầu tiên `i` thỏa mãn `*i == value`. Nếu không tìm thấy, `find` trả về bộ duyệt `last`. Sử dụng `find` rất dễ dàng. Chỉ cần lưu ý bộ duyệt `last` trỏ đến vị trí sau vị trí cuối của dãy muốn tìm. Ví dụ sau sẽ minh họa khuôn hình giải thuật `find`:

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

void main ()
{
    int a[10] = { 1,5,4,3,6,10,5,6,7,8};

    list<int> list_a;
    list_a.insert(list_a.begin(), a, a+8);

    ostream_iterator<int> oi(cout, " ");
    cout << "Day so: " << endl;
    copy(list_a.begin(),list_a.end(),oi);
    cout << endl;

    list<int>::iterator i1 =
find(list_a.begin(),list_a.end(),5);
    i1++;
    list<int>::iterator i2 = find(i1,list_a.end(),5);

    cout << "Cac so nam giua hai so 5 thu 1 va thu 2: "
         << endl;
    copy(i1,i2,oi);
    cout << endl;
```

```
    i2++;
    cout << "Tim 5 tiep: ";
    list<int>::iterator i3 = find(i2,list_a.end(),5);
    cout << ((i3 == list_a.end()) ? "Khong con" : "Van con")
         << endl;

}
```

```
Day so:
1 5 4 3 6 10 5 6
Cac so nam giua hai so 5 thu 1 va thu 2:
4 3 6 10
Tim 5 tiep: Khong con
```

Trong chương trình có tất cả ba lần sử dụng khuôn hình giải thuật `find`. Lần thứ nhất tìm được `i1` là bộ duyệt trỏ tới vị trí đầu tiên có giá trị `5`. Sau khi tăng `i1` tiếp tục tìm được `i2` trỏ tới vị trí thứ hai có giá trị `5`. Lần cuối cùng `find` không tìm thấy phần tử nào của `list` có giá trị `5` nữa nên trả về vị trí sau vị trí cuối. Như đã biết, vị trí này nhận được bằng cách gọi hàm thành phần `end()` của lớp `list`. Khi đem so sánh hai giá trị thấy bằng nhau, nên lệnh `cout` đưa thông báo ra màn hình là không tìm thấy.

Độ phức tạp của khuôn hình giải thuật `find` là tuyến tính, `find` thực hiện nhiều nhất `(last - first)` phép so sánh bằng.

5.2.1.3. Khuôn hình giải thuật find_if

Nguyên mẫu hàm:

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

Khuôn hình giải thuật `find_if` là mở rộng của khuôn hình giải thuật `find`. Khi tìm kiếm, khuôn hình giải thuật `find` sử dụng toán tử so sánh mặc định là `operator==`. Khuôn hình giải thuật `find_if` cho phép tìm kiếm theo một tiêu chuẩn bất kì, được quy định bởi đối tượng hàm `pred`. Đây là đối tượng hàm thuộc KHÁI NIỆM mệnh đề, tức là có giá trị trả về kiểu `bool`. Khuôn hình giải thuật `find_if` tìm kiếm trong khoảng `[first,last)` và trả về bộ duyệt đầu tiên `i` nếu `pred(*i)` bằng `true`. Nếu không tìm thấy, `find_if` trả về `last`.

Một cách tổng quát, nếu một giải thuật có sử dụng tham số là đối tượng

hàm thuộc KHÁI NIỆM Predicate, khuôn hình giải thuật đó có đuôi `_if` ở trong tên gọi. Một số giải thuật khác sử dụng đối tượng hàm thuộc KHÁI NIỆM Binary Predicate như hàm `adjacent_find` trình bày trong mục sau, không theo quy tắc như vậy, mà sử dụng nạp chồng hàm.

Sử dụng `find_if` cũng tương tự như sử dụng `find`, nhưng thay vì cung cấp giá trị để so sánh, ta phải cung cấp đối tượng hàm so sánh. Cần lưu ý đây là đối tượng hàm một tham số và phải tương thích kiểu với các phần tử trong dãy tìm kiếm. Ví dụ sau minh họa khuôn hình giải thuật `find_if`.

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

void main ()
{
    int a[10] = { 12,4,-4,3,8,13,51,-6,22,-38};

    list<int> list_a;
    list_a.insert(list_a.begin(), a, a+10);

    ostream_iterator<int> oi(cout, " ");
    cout << "Day so: " << endl;
    copy(list_a.begin(),list_a.end(),oi);
    cout << endl;

    // Tim so am thu nhat
    list<int>::iterator i1;
     i1=
find_if(list_a.begin(),list_a.end(),bind2nd(less<int>(),0));
    // Tim so am thu hai
     i1++;
    list<int>::iterator i2;
     i2 = find_if(i1,list_a.end(),bind2nd(less<int>(),0));

    cout << "Cac so duong giua hai so am thu 1 va thu 2: "
         << endl;
    copy(i1,i2,oi);
    cout << endl;

    i2++;
    cout << "Con so am nua? - ";
    list<int>::iterator i3;
```

```
    i3 = find_if(i2,list_a.end(),bind2nd(less<int>(),0));
    cout << ((i3 == list_a.end()) ? "Khong con" : "Van con: ")
         << (*i3)
         << endl;

}
```

```
Day so:
12 4 -4 3 8 13 51 -6 22 -38
Cac so duong giua hai so am thu 1 va thu 2:
3 8 13 51
Con so am nua? - Van con: -38
```

Chương trình trên sử dụng hàm `bind2nd`. `bind2nd` là một hàm cho phép chuyển một đối tượng hàm hai đối số thành một đối tượng hàm một đối số. Như sử dụng trong chương trình, `bind2nd` kết hợp từ đối tượng hàm hai đối số `less` (Xem chương Đối tượng hàm) và giá trị 0 thành một đối tượng hàm một đối số. Đối tượng hàm này trả về kết quả so sánh nhỏ hơn của đối số với 0.

Độ phức tạp của khuôn hình giải thuật `find_if` là tuyến tính. `find_if` thực hiện nhiều nhất là (`last - first`) lần hàm `pred`.

5.2.1.4. Khuôn hình giải thuật adjacent_find

Nguyên mẫu hàm:

```
template <class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first,
ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first,
ForwardIterator last, BinaryPredicate binary_pred);
```

Nguyên mẫu của `adjacent_find` có hai hàm nạp chồng. Tương tự như `find` và `find_if`, một hàm `adjacent_find` dùng toán tử so sánh mặc định là `operator==`, hàm nạp chồng còn lại sử dụng đối tượng hàm mệnh đề `binary_pred`.

Hàm `adjacent_find` được sử dụng khi muốn tìm hai số liên tiếp có cùng tính chất nào đó. Hàm thứ nhất tìm và trả về bộ duyệt đầu tiên `i`, sao cho `*i == *(i+1)`. Hàm thứ hai tìm và trả về bộ duyệt đầu tiên, sao cho `binary_pred(*i,*(i+1)) == true`. Nếu không tìm thấy, hàm trả về bộ duyệt trỏ tới vị trí sau vị trí cuối.

Ví dụ sau minh họa cả hai hàm nạp chồng `adjacent_find`: hàm thứ nhất tìm hai số bằng nhau liên tiếp đầu tiên, hàm thứ hai tìm hai số lẻ liên tiếp đầu tiên.

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

class prod_odd {
public:
   int operator() (const int& v1, const int& v2)
      { return v1%2 != 0 && v2%2 != 0; }
};
void main()
{
     int a[6] = { 2, 9, 6, 6, 13, 7};
     list<int> l;
     l.insert(l.begin(),a,a+6);
     // Hien thi
     ostream_iterator<int> oi(cout, " ");
     cout << "Day so: " << endl;
     copy(l.begin(),l.end(),oi);
     cout << endl;
     // Su dung ham nap chong thu nhat tim hai so bang nhau lien
tiep
     list<int>::iterator i = adjacent_find (l.begin(), l.end());

     if (i != l.end())
     {
            cout << "Hai so bang nhau lien tiep: "
                   << *i << " ";
            i++;
            cout << *i++ << endl;
     }
     else
            cout << "Khong co hai so bang nhau lien tiep" <<
endl;
     // Su dung ham nap chong thu hai tim hai so le lien tiep
     i = adjacent_find (l.begin(), l.end(),prod_odd());

     if (i != l.end())
     {
            cout << "Hai so le lien tiep: "
                   << *i << " ";
```

```
            i++;
            cout << *i++ << endl;
        }
        else
            cout << "Khong co hai so le lien tiep" << endl;
    }
```

```
Day so:
2 9 6 6 13 7
Hai so bang nhau lien tiep: 6 6
Hai so le lien tiep: 13 7
```

Khuôn hình giải thuật `adjacent_find` có độ phức tạp tuyến tính. Nếu `first == last`, không có phép so sánh nào thực hiện. Ngược lại, có nhiều nhất là `(last - first - 1)` phép so sánh.

5.2.1.5. Khuôn hình giải thuật find_first_of

Nguyên mẫu hàm:

```
template <class InputIterator, class ForwardIterator>
InputIterator find_first_of(InputIterator first1, InputIterator
last1,ForwardIterator first2, ForwardIterator last2);
```

```
template <class InputIterator, class ForwardIterator, class
BinaryPredicate>
InputIterator find_first_of(InputIterator first1, InputIterator
last1,ForwardIterator first2, ForwardIterator
last2,BinaryPredicate comp);
```

Tương tự khuôn hình giải thuật `find`, `find_first_of` cũng thực hiện tìm kiếm tuyến tính trên một dãy các đối tượng. Tuy nhiên, khuôn hình giải thuật `find` chỉ tìm kiếm một giá trị đặc biệt nào đó trong khi `find_first_of` tìm kiếm một giá trị bất kì trong khoảng định trước. Khuôn hình giải thuật này rất tiện lợi trong những bài toán tìm kiếm kiểu tồn tại như thế.

Nguyên mẫu cho ta thấy có hai hàm nạp chồng của `find_first_of`. Hàm thứ nhất tìm kiếm trong khoảng `[firstfirst1,last1)` và trả về bộ duyệt `i` đầu tiên sao cho tồn tại bộ duyệt `j` trong khoảng `[firstfirst2,last2)` mà `*i == *j`. Tương tự hàm nạp chồng thứ hai sử dụng đối tượng hàm comp để so sánh và trả về bộ duyệt `i` đầu tiên sao cho `comp(*i,*j) == true`.

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

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

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

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

    list<int>::iterator i = find_first_of(L1.begin(), L1.end(),
                                          L2.begin(), L2.end());
    cout << "Ket qua tim kiem: " << *i << endl;
}
```

```
Ket qua tim kiem: 3
```

Chương trình sau minh họa cách dùng khuôn hình giải thuật find_first_of với đối tượng hàm. Đối tượng hàm is_twice thuộc KHÁI NIỆM Binary Predicate. Toán tử gọi hàm của is_twice trả về giá trị true nếu tham số thứ hai gấp đôi tham số thứ nhất. Với hai dãy {34, 76, 45, 12, 8, 55} và {35,9,24}, khi áp dùng khuôn hình giải thuật find_first_of cùng với is_twice sẽ có kết quả trả về là phần tử thứ năm vì 24 == 12*2.

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

template <class T>
class is_twice
{
public:
    bool operator() (const T &x,const T &y)
    {
```

```
            if (x*2 == y) return true;
            else return false;
        }
};
void main()
{
    int a1[6] = {34, 76, 45, 8, 12, 55};
    int a2[3] = {35,9,24};

    list<int> L1; L1.insert(L1.begin(),a1,a1+6);
    list<int> L2; L2.insert(L2.begin(),a2,a2+3);

    list<int>::iterator i =
find_first_of(L1.begin(),L1.end(),L2.begin(),L2.end(),is_twice<in
t> ());
    cout << "Ket qua tim kiem: " << *i << endl;
}
```

```
Ket qua tim kiem: 12
```

Để thực hiện, `find_first_of` phải thực hiện tối đa là `(last1 - first1)*(last2 - first2)` phép so sánh.

5.2.1.6. Khuôn hình giải thuật count

Nguyên mẫu:

```
template <class InputIterator, class EqualityComparable>
iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last,
      const EqualityComparable& value);
```

Hàm giải thuật `count` được dùng khi cần đếm xem trong một khoảng `[firstfirst, last)` có bao nhiêu phần tử có giá trị bằng `value`. Theo nguyên mẫu, người mới quen với STL có thể e ngại khi thấy khai báo kiểu trả về của count là `iterator_traits<InputIterator>::difference_type` và không biết nó là kiểu gì nhưng thực tế, khi sử dụng ta chủ yếu dùng kiểu int hoặc các kiểu số nguyên thông thường khác. STL khai báo một cách rắc rối như thế là để xây dựng bộ thư viện. Theo quan điểm người dùng ta chỉ cần sử dụng một cách đơn giản với các kiểu số nguyên thông thường. Tuy nhiên, không phải trình biên dịch nào cũng hỗ trợ `iterator_traits` vì nó đòi hỏi phải hỗ trợ một tính năng của C++ gọi là *đặc tả một phần*. Chúng tôi đã thử

nghiệm và thấy rằng các trình biên dịch sau có thể sử dụng `iterator_traits` là VisualC++ 6.0 (trên Windows 98 trở đi), Visual.NET (trên Windows 2000 trở đi), và g++ 3.0, gcc 2.96 (trên RedHat Linux 7.2). Ví dụ, chạy chương trình sau với các trình biên dịch trên:

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

void main()
{
    char* p = "Chuong trinh minh hoa ham giai thuat count";
    cout << p << endl;
    int n = count(p,p + strlen(p),'o');
    cout << "Ki tu \'o\' xuat hien " << n << " lan trong cau
tren" << endl;
}
```

sẽ cho kết quả:

```
Chuong trinh minh hoa ham giai thuat count
Ki tu 'o' xuat hien 3 lan trong cau tren
```

Thực tế, STL còn một hàm nạp chồng khác của `count` không sử dụng `iterator_traits`. Hàm này là phiên bản cũ và có bốn đối số như nguyên mẫu dưới đây:

```
template <class InputIterator, class EqualityComparable, class
Size>
void count(InputIterator first, InputIterator last,
           const EqualityComparable& value,
           Size& n);
```

Hàm này cũng đếm xem trong dãy có bao nhiêu phần tử có giá trị bằng `value` rồi gán vào n. Hàm này vẫn được dùng vì vấn đề tương thích ngược và vì một số ít trình biên dịch không hỗ trợ đặc tả một phần ví dụ như Borland C 4.5. Tuy nhiên, người dùng nên sử dụng hàm thứ nhất, vì hàm sau sẽ bị loại bỏ khỏi STL. (VisualC++ đã không còn hỗ trợ hàm thứ hai).

Độ phức tạp: khuôn hình giải thuật `count` sử dụng đúng `first - last` phép so sánh khi thực hiện.

5.2.1.7. Khuôn hình giải thuật count_if

Nguyên mẫu:

```
template <class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);
```

Như đã nói, những khuôn hình giải thuật có đuôi `_if` là mở rộng của những khuôn hình giải thuật cùng tên. Khuôn hình giải thuật `count_if` cũng tương tự như khuôn hình giải thuật `count` nhưng sử dụng đối tượng hàm thuộc KHÁI NIỆM Predicate khi so sánh. Khuôn hình giải thuật `count_if` đếm tất cả các phần tử `i` trong khoảng `[first,last)` sao cho `pred(*i) == true`.

Ví dụ sau sử dụng đối tượng hàm và `count_if` để đếm các số chẵn trong một dãy số nguyên:

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

class modulus2
{
public:
     bool operator()(int n1)
     {
            return ((n1 % 2) == 0);
     }
};

void main()
{
     vector<int> v;
     v.push_back(1);v.push_back(3);v.push_back(2);
     v.push_back(5);v.push_back(14);v.push_back(6);

     cout << "Vector: " ;
     copy(v.begin(),v.end(),ostream_iterator<int> (cout," "));

     int n = count_if (v.begin(),v.end(),modulus2());
     cout << "\nSo cac so chan: " << n << endl;
}
```

```
Vector: 1 3 2 5 14 6
So cac so chan: 3
```

Tương tự count, count_if cũng có một hàm nạp chồng khác là phiên bản cũ của STL và cũng như count, không nên sử dụng hàm cũ này nếu như trình biên dịch có hỗ trợ hàm mới.

Độ phức tạp: count_if thực hiện tất cả (firstfirst - last) lần hàm pred để so sánh.

5.2.1.8. Khuôn hình giải thuật mismatch

Nguyên mẫu hàm:

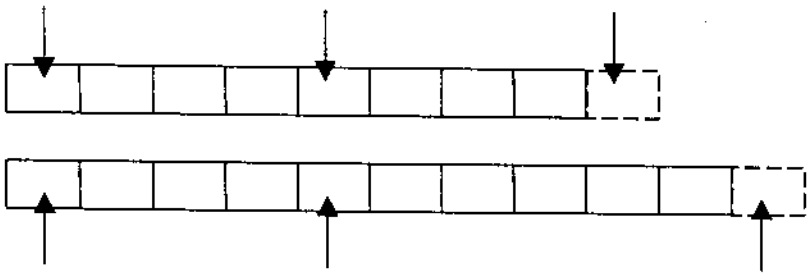
```
template <class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1
first1, InputIterator1 last1, InputIterator2 first2);
```

```
template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1
first1, InputIterator1 last1, InputIterator2
first2,BinaryPredicate binary_pred);
```

Không giống các khuôn hình giải thuật find đã giới thiệu, mismatch tìm kiếm sự khác nhau đầu tiên giữa hai dãy. Ví dụ có hai dãy A1 = { 3, 1, 4, 10, 1, 5, 9, 3 } và A2 = { 3, 1, 4, 10, 2, 3, 9, 3, 12, 7 }, khi áp dụng mismatch kết quả trả về sẽ là cặp bộ duyệt trỏ vào vị trí thứ năm trong hai dãy tương ứng với cặp giá trị (1, 2) được in nghiêng.

Hàm nạp chồng thứ nhất tìm bộ duyệt i đầu tiên trong [first,last) sao cho *i != *(first2 + (i - first1)). Kết quả trả về chính là cặp bộ duyệt i và (first2 + (i - first1). Nếu không tìm thấy i như vậy kết quả trả về là last1 và (first2 + (last1 - first1)). Như vậy, có thể thấy là dãy thứ hai phải có số phần tử nhiều hơn hoặc bằng dãy thứ nhất.

Hàm nạp chồng thứ hai cũng tìm kiếm tương tự nhưng không dùng operator!= mặc định, mà dùng đối tượng hàm binary_pred sao cho binary_pred(*i, *(first2 + (i - firstfalsefirst1)) == false. Chú ý là với mismatch ta tìm kiếm sao cho giá trị trả về của đối tượng hàm mệnh đề là false.



Chương trình sau sẽ minh họa cả hai hàm nạp chồng của mismatch. Trong chương trình cài đặt một đối tượng hàm mệnh đề eps_diff hai tham số so sánh độ lệch giữa hai biến với một giá trị định trước. Đối tượng hàm này sẽ được truyền vào khi sử dụng hàm mismatch thứ hai.

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

template <class T>
class eps_diff
{
private:
    T eps;
public:
    eps_diff(T epsilon) : eps(epsilon) {};
    bool operator() (T val1,T val2)
    {
        if (val1 > val2)
            return ((val1 - val2) < eps);
        else
            return ((val2 - val1) < eps);
    }
};
void main()
{
    float a[5] = {1.11, 3.01, 0.14, 0.22, 2.5};
    float b[5] = {1.11, 3.01, 0.14, 0.25, 2.51};
    float c[6] = {1.14, 3.02, 0.04, 0.19, 2.45, 4.12};
    //    Khai bao bien cho ket qua tra ve
    pair<float*,float*> difpos(0,0);

    //    Hien thi
```

```
    ostream_iterator<float> oi(cout, " ");
    copy(a,a+5,oi);      cout << endl;
    copy(b,b+5,oi);      cout << endl;
    copy(c,c+6,oi);      cout << endl;

    //    Su dung ham mismatch thu nhat
    difpos = mismatch (a,a+5,b);
    cout << "\nDay thu 1 va day thu 2 khac nhau tai vi tri:";
    cout << difpos.first - a + 1 << endl;

    //    Su dung ham mismatch thu hai
    cout <<"Day thu 1 va day thu 3 lech nhau qua 0.05 tai vi
tri:";
    difpos = mismatch(a,a+5,c,eps_diff<float>(0.05));
    cout << difpos.first - a + 1 << endl;
}
```

Ta xem cách sử dụng khuôn hình giải thuật `mismatch` thứ hai trong ví dụ trên. Toán tử gọi hàm của đối tượng hàm `eps_diff` trả về `true`, nếu độ lệch giữa hai biến truyền vào nhỏ hơn eps cho trước và trả về `false`, nếu ngược lại. Do vậy, khi khởi tạo `eps_diff` với giá trị 0.05 rồi đưa vào `mismatch`, `mismatch` sẽ tìm và trả về vị trí đầu tiên của cặp phần tử trong hai dãy lệch nhau quá 0.05.

Kết quả:

```
1.11 3.01 0.14 0.22 2.5
1.11 3.01 0.14 0.25 2.51
1.14 3.02 0.04 0.19 2.45 4.12

Day thu 1 va day thu 2 khac nhau tai vi tri: 4
Day thu 1 va day thu 3 lech nhau qua 0.05 tai vi tri: 3
```

Để thực hiện, `mismatch` cần tối đa `(last1 - first1)` phép so sánh, hoặc `(last1 - first1)` lần thực hiện đối tượng hàm `binary_predpred`.

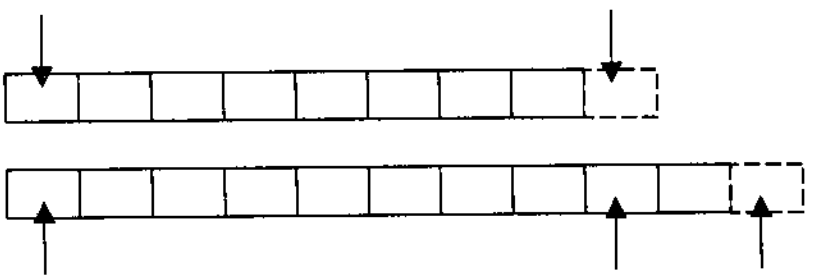
5.2.1.9. Khuôn hình giải thuật equal

Nguyên mẫu:

```
template <class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);
```

```
template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate binary_pred);
```

Khuôn hình giải thuật `mismatch` tìm kiếm sự khác nhau đầu tiên giữa hai dãy, còn `equal` kiểm tra xem hai dãy có giống nhau hay không khi so sánh từng phần tử một. Tuy nhiên, không nhất thiết hai dãy phải có số phần tử bằng nhau. `equal` chỉ kiểm tra đến phần tử cuối cùng của dãy thứ nhất. Nếu cho đến lúc đó, hai dãy vẫn giống nhau, `equal` sẽ trả về `true`, mà không quan tâm tới phần còn lại của dãy thứ hai.



Ví dụ với hai dãy trên, áp dụng `equal` ta sẽ nhận được kết quả trả về là `true`. Nhìn vào nguyên mẫu của `equal` ta thấy có hai hàm nạp chồng. Hàm thứ nhất trả về `true` nếu với mọi bộ duyệt `i` trong khoảng `[firstfirst1,last1)` có đẳng thức sau: `*i == *(first2 + (i - first1))`. Tương tự, hàm thứ hai trả về `true` nếu mọi bộ duyệt `i` trong khoảng `[firstfirst1,last1), binary_predpred(*i,*(first2 + (i - firsttruefirst1))) == true`.

Chương trình sau minh họa cả hai hàm `equal`. Chương trình có sử dụng đối tượng hàm `eps_diff` trong ví dụ về hàm `mismatch`.

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

template <class T>
class eps_diff
{
```

```
    // Xem vi du ve mismatch
};

void main()
{
    float a[5] = {1.11, 3.01, 0.14, 0.22, 2.51};
    float b[6] = {1.11, 3.01, 0.12, 0.25, 2.49, 1.98};

    //      Hien thi
    ostream_iterator<float> oi(cout, " ");
    //      In ra man hinh cac day
    copy(a,a+5,oi);     cout << endl;
    copy(b,b+5,oi);     cout << endl;

    //      Su dung ham equal thu nhat
    if (equal(a,a+5,b))
            cout << "Hai day giong nhau" << endl;
    else
            cout << "Hai day khac nhau" << endl;
    //      Su dung ham equal thu hai
    if (equal(a,a+5,b,eps_diff<float> (0.05)))
            cout << "Hai day xap xi nhau" << endl;
    else
            cout << "Hai day khong xap xi nhau" << endl;
}
```

```
1.11 3.01 0.14 0.22 2.51
1.11 3.01 0.12 0.25 2.49
Hai day khac nhau
Hai day xap xi nhau
```

Lần gọi `equal` thứ nhất cho kết quả `false`, vì tại vị trí thứ ba có `0.14 != 0.12` còn lần gọi `equal` thứ hai cho kết quả `true`, vì cho đến phần tử cuối cùng của dãy thứ nhất, các phần tử của hai dãy vẫn không lệch nhau quá `0.05`.

Khuôn hình giải thuật `equal` phải thực hiện tối đa là `(last1 - first1)` phép so sánh hoặc `(last1 - first1)` lần toán tử gọi hàm của đối tượng hàm `binary_pred`.

5.2.1.10. Khuôn hình giải thuật search

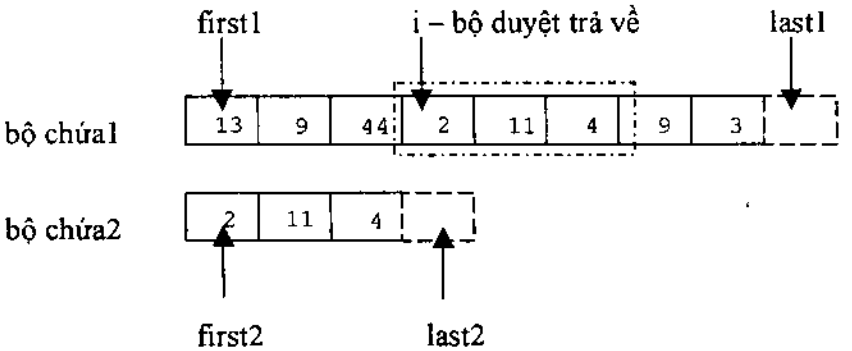
Nguyên mẫu:

```
template <class ForwardIterator1, class
ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1,
ForwardIterator1 last1,ForwardIterator2 first2,
```

```
ForwardIterator2 last2);
```

```
template <class ForwardIterator1, class ForwardIterator2, class
BinaryPredicate>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1
last1,ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate binary_pred);
```

Khuôn hình giải thuật `search` tìm trong dãy `[firstfirst1,last1)` dãy con giống với dãy `[firstfirst2,last2)`. Nếu tìm thấy, `search` trả về bộ duyệt `i` đầu tiên của dãy con. Nếu không tìm thấy dãy con nào thỏa mãn, `search` trả về `last1`.



Khuôn hình giải thuật `search` không quy định số phần tử của dãy thứ nhất phải nhiều hơn số phần tử của dãy thứ hai. Trong hai hàm nạp chồng, hàm thứ nhất so sánh các phần tử của hai dãy bằng toán tử `operator==`, còn hàm thứ hai sử dụng đối tượng hàm mệnh đề `binary_pred`.

Ví dụ sau minh họa cách dùng hai hàm search. Với hàm nạp chồng thứ hai sử dụng đối tượng hàm divided (không có trong STL). Toán tử gọi hàm của divided kiểm tra tính chia hết của hai tham biến truyền vào.

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

class divided
{
public:
    bool operator() (int var1,int var2)
```

```
    {
            return (var1 % var2) == 0;

    }
};
void main ()
{
    int v1[6] = { 1, 1, 2, 6, 8, 10};
    int v2[2] = { 2, 4 };

    //    Hien thi
    ostream_iterator<int> oi (cout, " ");
    cout << "v1: ";  copy(v1,v1+6,oi);  cout << endl;
    cout << "v2: ";  copy(v2,v2+2,oi);  cout << endl;

     //    Su dung ham search thu nhat
    int* location = search (v1, v1 + 6, v2, v2 + 2);
    if (location == v1 + 6)    // tro den vi tri sau vi tri cuoi
            cout << "v1 khong chua v2";
    else
            cout << "Tim thay v2 trong v1 tai vi tri: "
                << location - v1 +1;
    cout << "                    // Ham search thu nhat" << endl;

     //    Su dung ham search thu hai
    location = search (v1, v1 + 6,v2, v2 + 2, divided());
    if (location == v1 + 6) // tro den vi tri sau vi tri cuoi
            cout << "v1 khong chua v2";
    else
            cout << "Tim thay v2 trong v1 tai vi tri: "
                << location - v1 +1;
    cout << " // Ham search thu hai" << endl;
}
```

```
v1: 1 1 2 6 8 10
v2: 2 4
v1 khong chua v2                    // Ham search thu nhat
Tim thay v2 trong v1 tai vi tri: 4 // Ham search thu hai
```

Khi sử dụng, hàm search thứ hai có kết quả trả về như trên vì đã sử dụng đối tượng hàm divided để kiểm tra. Do 6 chia hết cho 2 và 8 chia hết cho 4 nên search trả về bộ duyệt trỏ vào vị trí của 6.

Khuôn hình giải thuật `search` có độ phức tạp bình phương trong trường hợp tồi nhất. Khi đó, `search` phải thực hiện `(last1 - first1) * (last2 - first2)` phép so sánh. Tuy nhiên, trường hợp tồi nhất ít khi xảy

ra. Độ phức tạp trung bình của search chỉ là tuyến tính.

5.2.1.11. Khuôn hình giải thuật search_n

Nguyên mẫu:

```
template <class ForwardIterator, class Integer, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator
last,
                         Integer count, const T& value);
```

```
template <class ForwardIterator, class Integer,
          class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator
last,
                         Integer count, const T& value,
                         BinaryPredicate binary_pred);
```

Khuôn hình giải thuật `search_n` tìm kiếm trong khoảng `[first,last)` dãy con gồm count phần tử trong đó tất cả các phần tử đều có giá trị bằng `value`. Nếu tìm thấy, `search_n` trả về bộ duyệt đầu tiên của dãy con. Nếu không `search_n` trả về `last`. Chính xác hơn, `search_n` chỉ tìm trong khoảng `[firstfirst, last - count)` và trả về bộ duyệt `i` đầu tiên sao cho mọi bộ duyệt `j` trong khoảng `[i, i + count), *j == value`.

Hàm nạp chồng thứ hai tìm dãy con gồm `count` phần tử sao cho mọi bộ duyệt `i` trong dãy, `binary_pred(*i, value) == true`.

Ví dụ sau minh họa hai hàm `search_n`.

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

class divided
{
public:
     bool operator() (int var1,int var2)
     {
            return (var1 % var2) == 0;
     }
```

```
};
void main()
{
    int a[8] = {1, 2, 3, 5, 10, 10, 10, 14};
    // Hien thi
    copy(a,a+8,ostream_iterator<int> (cout," ")); cout << endl;
    // Su dung ham search_n thu nhat
    int n = 3,val = 10;
    int* res = search_n(a,a+8,n,val);
    if (res == a + 8)
        cout << "Khong tim thay" << endl;
    else
        cout << "Tim thay "
             << n
             << " phan tu lien tiep bang "
             << val
             << " tai vi tri " << res - a + 1<< endl;
    // Su dung ham search_n thu hai
    val = 5;
    res = search_n(a,a+8,n,val,divided());
    if (res == a + 8)
        cout << "Khong tim thay " << endl;
    else
        cout << "Tim thay "
             << n
             << " phan tu lien tiep chia het cho "
             << val
             << " tai vi tri " << res - a + 1 << endl;
}
```

```
1 2 3 5 10 10 10 14
Tim thay 3 phan tu lien tiep bang 10 tai vi tri 4
Tim thay 3 phan tu lien tiep chia het cho 5 tai vi tri 3
```

Độ phức tạp của `search_n` là tuyến tính, `search_n` thực hiện tối đa `(last - first)` phép so sánh hoặc lời gọi tới toán tử hàm của `binary_pred`.

5.2.1.12. Khuôn hình giải thuật find_end

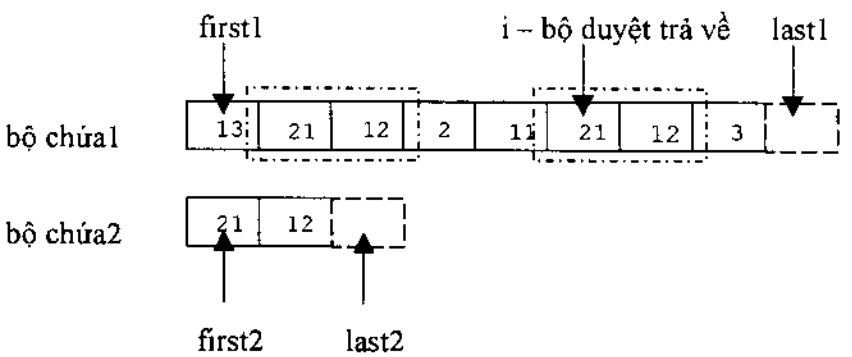
Nguyên mẫu:

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(ForwardIterator1 first1,
ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2
last2);
```

```
template <class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate>
ForwardIterator1 find_end(ForwardIterator1 first1,
ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2
last2, BinaryPredicate comp);
```

Đúng ra, khuôn hình giải thuật `find_end` phải có tên là `search_end` vì `find_end` giống các khuôn hình giải thuật `search` hơn là giống các khuôn hình giải thuật `find`. Cũng chính vì thế, trong tài liệu tham chiếu của STL cũng như trong cuốn sách này, `find_end` được xếp sau các khuôn hình giải thuật `search`, mà không được xếp sau các khuôn hình giải thuật `find`.

Tương tự search, `find_end` cũng tìm dãy con trong khoảng `[firstfirst1,last1)` sao cho dãy này giống với dãy `[firstfirst2,last2)`, nhưng khác với search, `find_end` tìm dãy con cuối cùng thỏa mãn tính chất đó. Nếu tìm thấy, `find_end` trả về bộ duyệt đầu tiên của dãy con, nếu không tìm thấy, `find_end` trả về `last1`.



Theo nguyên mẫu, có hai hàm nạp chồng của `find_end`. Hàm thứ nhất tìm và trả về bộ duyệt `i` cuối cùng sao cho với mọi bộ duyệt `j` trong `[firstfirst2,last2)`, `*j = *(i + (j - first2))`. Hàm thứ hai tìm sao cho `binary_pred(*(i+(j-first2)), *j) == truetrue`.

Chương trình sau minh họa hai hàm nạp chồng `find_end`. Đối tượng hàm `same_signed` có toán tử gọi hàm kiểm tra sự cùng dấu của hai biến, `same_signed` sẽ được sử dụng trong hàm `find_end` thứ hai.

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

template <class T>
class same_signed
{
public:
    bool operator() (T val1,T val2)
    {
            return (val1*val2 > 0);
    }
};
void main()
{
    int a[8] = {1, 1, 1, -1, -1, -1, -1, 1};
    int b[4] = {-1,-1,-1};
    int c[3] = {-1,-2,3};
    // Hien thi
    ostream_iterator<int> oi(cout," ");
    cout << "a: "; copy(a,a+8,oi); cout << endl;
    cout << "b: "; copy(b,b+3,oi); cout << endl;
    cout << "c: "; copy(c,c+3,oi); cout << endl;
    // Su dung ham find_end thu nhat
    int* res = find_end(a,a+8,b,b+3);
    if (res == a + 8)
            cout << "Khong tim thay" << endl;
    else
            cout << "Day con cuoi cung trong a bang b tai vi tri
"
            << res - a + 1 << endl;
    // Su dung ham find_end thu hai
    res = find_end(a,a+8,c,c+3,same_signed<int>());
    if (res == a + 8)
            cout << "Khong tim thay" << endl;
    else
            cout << "Day con cuoi cung trong a cung dau voi c tai
vi tri "
                << res - a + 1 << endl;
}
```

```
a: 1 1 1 -1 -1 -1 -1 1
b: -1 -1 -1
c: -1 -2 3
Day con cuoi cung trong a bang b tai vi tri 5
Day con cuoi cung trong a cung dau voi c tai vi tri 6
```

Ta thấy, tuy có 4 số -1 liên tiếp, nhưng kết quả trả về tại vị trí 5 vì find_end tìm dãy con cuối cùng. Nếu thay vì dùng find_end, ta dùng search sẽ có kết quả trả về tại vị trí thứ 4. Với hàm find_end thứ hai, kết quả trả về là vị trí 6 vì (-1,-2,3) cùng dấu với (-1,-1,1).

Số lượng các phép so sánh mà find_end thực hiện tỉ lệ với (last1 - first1) * (last2 - first2). Nếu cả Forward Iterator1 và Forward Iterator2 thuộc kiểu Bidirectional Iterator, độ phức tạp của find_end là tuyến tính. Trong trường hợp tồi nhất, find_end phải thực hiện (last1 - first1) * (last2 - first2) phép so sánh.

5.2.2. Các giải thuật làm đổi biến

Các giải thuật làm đổi biến khi thực hiện trên các bộ chứa sẽ gây biến đổi lên các đối tượng lưu trữ trong các bộ chứa đó. Chú ý là tuy giải thuật làm việc với các bộ duyệt nhưng việc biến đổi chỉ thực hiện trên các đối tượng được trỏ bởi bộ duyệt. Từ góc độ sử dụng, các giải thuật làm đổi biến được dùng nhiều hơn các giải thuật không làm đổi biến và cách sử dụng cũng rất linh hoạt. Các giải thuật làm đổi biến gồm các hàm phục vụ mục đích như sao chép, hoán vị, thay thế, khai triển,...

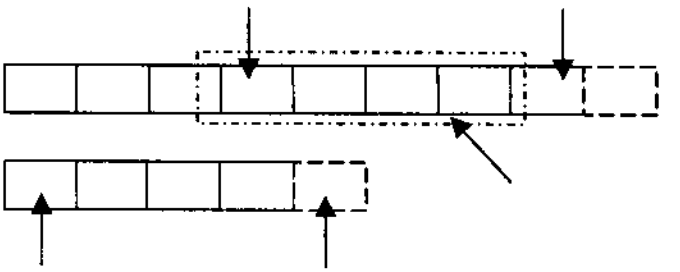
5.2.2.1. Khuôn hình giải thuật copy

Chúng ta đã quá quen với khuôn hình giải thuật copy trong suốt các chương trình từ chương 1 đến giờ. Đó là vì copy được sử dụng nhiều cho mục đích hiển thị. Tuy nhiên, ta mới sử dụng một cách mặc nhiên, mà chưa giải thích rõ ràng về giải thuật này. Nguyên mẫu của hàm copy cho như sau:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

Khuôn hình giải thuật copy thực hiện việc sao chép các phần tử trong khoảng [first,last) tới khoảng [result, result + (last - first)), nghĩa là copy thực hiện các phép gán *result = *firstfirst, *(result + 1) = *(first + 1), v.v... Việc sao chép có thể thực hiện từ bộ chứa này sang bộ chứa khác, nhưng cũng có thể thực hiện ngay trên chính một bộ chứa. Giá trị trả về của copy là result + (last - first). Chú ý rằng khoảng [result, result + (last - first)) phải là khoảng hợp lệ. Một cách đơn giản, dãy được sao chép phải

(last - first). Chú ý rằng khoảng [result, result + (last - first)) phải là khoảng hợp lệ. Một cách đơn giản, dãy được sao chép phải có đủ chỗ cho các phần tử sao chép. Ngoài ra, cần thêm một điều kiện là result không thuộc khoảng [first,last) khi sao chép trên các khoảng chồng nhau của cùng một dãy. Nếu điều kiện này không thỏa mãn, chương trình vẫn chạy nhưng không cho ra kết quả mong muốn.



Ví dụ sau minh họa cách sử dụng copy.

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

void main()
{
    ostream_iterator<int> oi(cout , " ");
    vector<int> a(8),b(8);

    a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 4;
    // Ap dung copy tren cung mot day
    copy(a.begin(),a.begin() + 4,a.begin() + 4);
    // Hien thi
    cout << "a: ";      copy(a.begin(),a.end(),oi); cout <<
endl;
    // Ap dung copy tren hai khoang chong nhau
    copy(a.begin() + 2, a.begin() + 6, a.begin() + 1);
    // Hien thi
    cout << "a: ";      copy(a.begin(),a.end(),oi); cout <<
endl;
    // Ap dung copy tren hai day khac nhau:
    copy(a.begin(),a.end(),b.begin());
    // Hien thi
```

```
    cout << "b: ";        copy(a.begin(),a.end(),oi); cout <<
endl;
}
```

```
a: 1 2 3 4 1 2 3 4
a: 1 3 4 1 2 2 3 4
b: 1 3 4 1 2 2 3 4
```

Dòng lệnh `vector<int> a(8), b(8);` tạo ra hai `vector` kích thước 8. Lần thứ nhất `copy` sao chép 4 phần tử đầu tiên vào 4 phần tử tiếp theo trong cùng một `vector a`. Lần thứ hai, `copy` thực hiện việc sao chép trên hai khoảng chồng lên nhau. (Sao chép các phần tử 3, 4, 1, 2 đè lên các phần tử 2, 3, 4, 1). Điều này hoàn toàn là được vì như đã nói, việc thay đổi giá trị chỉ thực hiện trên các đối tượng mà không liên quan tới các bộ duyệt. Lần thứ ba, `copy` thực hiện việc sao chép từ `vector a` sang `vector b`. Chú ý là ta khởi tạo `a` và `b` cùng kích thước nên có thể sao chép được. Nếu khởi tạo `b` với kích thước nhỏ hơn hoặc không khởi tạo kích thước, chương trình sẽ gây lỗi.

Nếu sao chép trên các khoảng chồng nhau mà result thuộc `[first,last)` thì sẽ gây kết quả sai. Ví dụ, trong chương trình trên thay câu lệnh:

```
copy(a.begin() + 2, a.begin() + 6, a.begin() + 1);
```

bằng:

```
copy(a.begin() + 2, a.begin() + 6, a.begin() + 3);
```

thì sẽ có kết quả:

```
a: 1 2 3 4 1 2 3 4
a: 1 2 3 3 3 3 3 4
b: 1 2 3 3 3 3 3 4
```

Rõ ràng, ta muốn sao chép các phần tử 3, 4, 1, 2 lên các phần tử 4, 1, 2, 3 nhưng kết quả chỉ toàn các số 3. Người đọc muốn tìm hiểu thêm, xin xem phần bài tập ở cuối chương. Người lập trình chỉ cần nhớ rằng result không được thuộc khoảng `[first,last)` khi sao chép các khoảng chồng nhau của cùng một dãy.

Ứng dụng hay dùng nhất của `copy` là hiển thị kết quả ta đã rất quen thuộc. Khi hiển thị, thay vì sao chép vào một bộ chứa nào đó, `copy` sẽ sao

chép các phần tử ra màn hình qua một bộ duyệt đặc biệt là `ostream_iterator`. Chi tiết về `ostream_iterator` có thể xem trong chương về bộ duyệt.

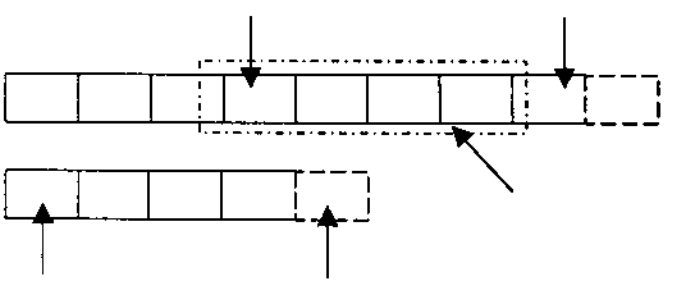
Khuôn hình giải thuật `copy` có độ phức tạp tuyến tính, `copy` phải thực hiện `last - first` phép gán.

5.2.2.2. Khuôn hình giải thuật copy_backward

Nguyên mẫu:

```
template <class BidirectionalIterator1, class
BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1
first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 result);
```

Như tên gọi, khuôn hình giải thuật `copy_backward` thực hiện việc sao chép ngược lại với khuôn hình giải thuật `copy` thông thường. Nghĩa là `copy_backward` thực hiện việc sao chép các phần tử trong khoảng `[first,last)` vào khoảng `[result - (last - first), result)` nhưng `copy_backward` tiến hành các phép gán từ cuối dãy ngược lại: `*(result - 1) = *(last - 1)`, `*(result - 2) = *(last - 2)`, v.v... Giá trị trả về của `copy_backward` là `result - (last - first)`. Điều kiện để `copy_backward` thực hiện được là `[result - (last - first), result)` phải là khoảng hợp lệ. Một cách đơn giản, dãy được sao chép phải có đủ chỗ cho các phần tử sao chép. Ngoài ra, cần thêm một điều kiện là result không thuộc khoảng `[first,last)` khi sao chép trên các khoảng chồng nhau thuộc cùng một dãy. Nếu điều kiện này không thỏa mãn, chương trình vẫn chạy nhưng không cho ra kết quả mong muốn.



Người đọc có thể thắc mắc tại sao cần đến hai hàm copy gần như giống hệt nhau cho cùng một mục đích. Nếu thực hiện việc sao chép trên các dãy không chồng nhau, hai hàm hoàn toàn tương đương. Nhưng khi thực hiện sao chép trên các khoảng chồng nhau của cùng một dãy sẽ có sự khác biệt. Lưu ý tới điều kiện *result không thuộc khoảng* [first,last) trong hai hàm. Đối với copy, điều kiện này có nghĩa là *bộ duyệt đầu của dãy được sao chép không được nằm trong dãy sao chép*, trong khi đó với hàm copy_backward điều kiện đó lại có nghĩa là *bộ duyệt cuối của dãy được sao chép không được nằm trong dãy sao chép*. Tóm lại, việc quyết định sử dụng copy hay copy_backward phụ thuộc vào việc có sao chép trên các khoảng chồng nhau hay không.

Ví dụ sau minh họa khuôn hình giải thuật copy_backward.

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

void main()
{
    ostream_iterator<int> oi(cout, " ");
    vector<int> v1(7),v2(5);

    v1[4] = 5; v1[5] = 6; v1[6] = 7;
    cout << "v1: "; copy(v1.begin(),v1.end(),oi); cout << endl;
    // Su dung copy_backward tren cung mot day
    copy_backward(v1.begin() + 4,v1.end(),v1.begin() + 4);
    // Hien thi
    cout << "v1: "; copy(v1.begin(),v1.end(),oi); cout << endl;
    // Su dung copy_backward tren hai day
    copy_backward(v1.begin() + 2,v1.end(),v2.end());
    // Hien thi
    cout << "v2: "; copy(v2.begin(),v2.end(),oi); cout << endl;

}
```

```
v1: 0 0 0 0 5 6 7
v1: 0 5 6 7 5 6 7
v2: 6 7 5 6 7
```

Khuôn hình giải thuật copy_backward có độ phức tạp tuyến tính. copy_backward phải thực hiện last - first phép gán.

5.2.2.3. Khuôn hình giải thuật swap

Nguyên mẫu:

```
template <class Assignable>
void swap(Assignable& a, Assignable& b);
```

Từ góc độ sử dụng, khuôn hình giải thuật swap là một trong những khuôn hình giải thuật đơn giản nhất của STL. Nó thực hiện việc hoán chuyển giá trị giữa hai tham số: giá trị của a được gán cho b và giá trị của b được gán cho a.

Ví dụ sử dụng swap:

```
int x = 1;
int y = 2;
swap(x, y);  // x == 2, y == 1
```

Tất nhiên, với đặc tính khái lược, swap không chỉ làm việc với kiểu nguyên, mà có thể làm việc với kiểu bất kì. Để thực hiện, swap cần gọi một hàm cấu tử sao chép và một toán tử gán. Do vậy, tham số cho swap phải là kiểu Assignable tức là có cấu tử sao chép và toán tử gán. Cơ chế làm việc của swap vì vậy chưa hẳn là tối ưu cho mọi kiểu. Ví dụ, nếu sử dụng swap với hai vector<double> có N phần tử sẽ mất khoảng 3*N phép gán trong khi nếu sử dụng hàm thành phần swap của chính vector chỉ mất 9 phép gán. Nếu muốn tối ưu, người dùng nên tự định nghĩa hàm thành phần swap cho kiểu riêng của mình khi cần thiết. Các bộ chứa của STL như vector, deque đều có cơ chế đặc biệt bên trong các bộ chứa để khi gọi khuôn hình giải thuật swap cho hai bộ chứa, nó sẽ gọi tới hàm thành phần swap của chính bộ chứa. Chi tiết về vấn đề này xin xem thêm trong chương về các bộ chứa.

5.2.2.4. Khuôn hình giải thuật swap_ranges

Nguyên mẫu:

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1,
                             ForwardIterator1 last1,
                             ForwardIterator2 first2);
```

Trong khi swap chỉ hoán chuyển giá trị của hai đối tượng, swap_ranges hoán chuyển giá trị của một dãy đối tượng. Khuôn hình giải thuật

swap_ranges hoán chuyển mỗi phần tử trong khoảng [first,last) tương ứng với một phần tử trong khoảng [first2,first2 + (last1 - first1)). Giá trị trả về của swap_ranges là first2 + (last1 - first1).

Ví dụ sử dụng swap_ranges:

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

void main()
{
    ostream_iterator<int> oi(cout," ");

    vector<int> v1(6),v2(6);
    fill(v1.begin(),v1.end(),0);
    fill(v2.begin(),v2.end(),1);

    cout << "v1: "; copy(v1.begin(),v1.end(),oi); cout << endl;
    cout << "v2: "; copy(v2.begin(),v2.end(),oi); cout << endl;

    swap_ranges(v1.begin(),v1.end(),v2.begin());

    cout << "v1: "; copy(v1.begin(),v1.end(),oi); cout << endl;
    cout << "v2: "; copy(v2.begin(),v2.end(),oi); cout << endl;
}
```

```
v1: 0 0 0 0 0 0
v2: 1 1 1 1 1 1
v1: 1 1 1 1 1 1
v2: 0 0 0 0 0 0
```

Chương trình sử dụng khuôn hình giải thuật fill để khởi tạo vector với một giá trị cho trước. Khuôn hình giải thuật fill sẽ được đề cập sau. Thực tế, nếu sử dụng swap_ranges như trên ta có thể dùng swap như dưới đây cũng được:

```
swap(v1,v2);
```

Tại sao lại cần swap_ranges khi swap cũng làm được điều tương tự? Tuy nhiên, trường hợp trên là ta hoán chuyển toàn bộ vector. Không nhất

thiết lúc nào ta cũng cần hoán chuyển toàn bộ vector mà chỉ cần hoán chuyển một phần như sau:

```
swap_ranges(v1.begin(),v1.begin() + 3, v2.begin());
```

khi đó sẽ có kết quả:

```
v1: 0 0 0 0 0 0
v2: 1 1 1 1 1 1
v1: 1 1 1 0 0 0
v2: 0 0 0 1 1 1
```

Rõ ràng với trường hợp hoán chuyển một phần của dãy ta buộc phải dùng swap_ranges. Ngoài ra, ta cũng có thể hoán chuyển các phần tử ngay trên chính một dãy. Điều này, swap không thể làm được. Chú ý là khi hoán chuyển các phần tử trong cùng một dãy, hai khoảng [firstfirst1, last1) và [first2, first2 + (last1 - first1)) không được giao nhau.

Độ phức tạp của swap_ranges là tuyến tính. swap_ranges thực hiện tất cả last1 - first1 phép hoán chuyển.

5.2.2.5. Khuôn hình giải thuật transform

Nguyên mẫu:

```
template <class InputIterator, class OutputIterator, class
UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryFunction
op);
```

```
template <class InputIterator1, class InputIterator2, class
OutputIterator, class BinaryFunction>
OutputIterator transform(InputIterator1 first1, InputIterator1
last1,
                         InputIterator2 first2, OutputIterator
result,                  BinaryFunction binary_op);
```

Có thể nói, transform là một giải thuật khá mạnh và có nhiều ứng dụng. Nếu bạn đọc đã quen với for_each sẽ thấy transform còn tiện lợi hơn và cách sử dụng cũng tương tự. Đối với hàm nạp chồng thứ nhất có bốn tham số, transform thực hiện op(*i) trên mỗi bộ duyệt i của khoảng

[first, last) và kết quả trả về cho bộ duyệt ra tương ứng bắt đầu từ result. Nghĩa là với mọi 0 ≤ n < last - first, transform thực hiện phép gán *(result + n) = op(*(first + n)). Để ý rằng đây chính là điểm hơn hẳn của transform so với for_each vì với for_each, kết quả trả về của toán tử gọi hàm op bị bỏ qua.

Hàm nạp chồng thứ hai của transform có năm tham số, cũng tương tự nhưng thực hiện trên hai dãy. Nghĩa là với mọi 0 ≤ n < last - first, transform thực hiện phép gán *(result + n) = binary_op(*(first1 + n),*(first2+n)).

Chương trình sau minh họa hai hàm nạp chồng transform.

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

void main()
{
    vector<int> v(5),y(8);
    // Khoi tao v voi gia tri 5
    fill(v.begin(),v.end(),5);
    // Khoi tao nua dau cua y voi gia tri 2
    fill(y.begin(),y.begin() + 4,2);
    // Khoi tao nua cuoi cua y voi gia tri -2
    transform(y.begin(),y.begin()+4, y.begin()+4,negate<int>());

    ostream_iterator<int> oi(cout," ");
    cout << "v     : "; copy(v.begin(),v.end(),oi); cout <<
endl;
    cout << "y     : "; copy(y.begin(),y.end(),oi); cout <<
endl;

    cout << "v + y : ";
    // Cong hai vector bang trasform va hien thi
    transform(v.begin(),v.end(),y.begin(),oi, plus<int>() );
}
```

```
v     : 5 5 5 5 5
y     : 2 2 2 2 -2 -2 -2 -2
v + y : 7 7 7 7 3
```

Sau khi dùng fill để khởi tạo nửa đầu `vector y`, ta dùng `transform` để khởi tạo nửa sau bằng cách lấy nghịch dấu nửa đầu nhờ đối tượng hàm `negate`. Bốn giá trị `2` đầu tiên lần lượt bị đảo dấu thành `-2` và ghi tiếp vào nửa sau của `y`. Đây là ví dụ cho thấy cách dùng khuôn hình giải thuật `transform` thứ nhất và trên hai khoảng thuộc cùng một dãy. Lần thứ hai, ta dùng khuôn hình giải thuật `transform` thứ hai thực hiện phép cộng trên hai `vector v` và `y` nhờ đối tượng hàm `plus` rồi ghi kết quả ra màn hình qua bộ duyệt đặc biệt `oi`.

Qua ví dụ trên có thể thấy sử dụng `transform` rất linh hoạt. Ta có thể thực hiện `transform` trên một dãy hoặc trên hai dãy. Các kết quả thực hiện có thể lưu lại trên các bộ chứa hoặc ghi thẳng ra thiết bị hiển thị hoặc ra file.

Khuôn hình giải thuật `transform` có độ phức tạp tuyến tính, có tất cả `last - first` lần thực hiện toán tử gọi hàm `op` hoặc `binary_op`.

5.2.2.6. Khuôn hình giải thuật replace và replace_if

Nguyên mẫu:

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last, const
T& old_value,const T& new_value)
```

```
template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
Predicate pred, const T& new_value)
```

Khuôn hình giải thuật replace thay thế mọi phần tử trong khoảng `[first,last)` có giá trị `old_value` bằng giá trị mới `new_value`. Khuôn hình giải thuật `replace_if` tương tự tìm mọi bộ duyệt `i` trong khoảng `[first,last)`, nếu `predpred(*i) == true` thì thực hiện phép gán `*i = new_value`.

Chương trình dưới đây minh họa hai khuôn hình giải thuật `replace` và `replace_if`.

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;
```

```
void main()
{
    int a[6] = {1, 2, 3, 4, 5, 6};

    ostream_iterator<int> oi(cout, " ");
    cout << "a: "; copy(a,a+6,oi); cout << endl;

    replace_if(a,a+6,bind2nd(less<int> (),5),5);
    cout << "a: "; copy(a,a+6,oi); cout << endl;

    replace(a,a+6,5,6);
    cout << "a: "; copy(a,a+6,oi); cout << endl;
}
```

```
a: 1 2 3 4 5 6
a: 5 5 5 5 5 6
a: 6 6 6 6 6 6
```

Hàm `bind2nd` được sử dụng để kết hợp một đối tượng hàm hai tham số với một giá trị thành một đối tượng hàm một tham số. Cụ thể trong chương trình ta kết hợp đối tượng hàm `less<int>` với giá trị 5 để được đối tượng hàm kiểm tra xem một số có nhỏ hơn 5 hay không. Khuôn hình giải thuật `replace_if` trong chương trình thay tất cả các số nhỏ hơn 5 bằng 5, sau đó khuôn hình giải thuật `replace` thay tất cả các giá trị 5 thành giá trị 6 và ta có kết quả như trên.

Có thể thấy sử dụng `replace` và `replace_if` khá dễ dàng. Độ phức tạp của cả hai giải thuật này đều là tuyến tính. Chúng thực hiện tất cả `last - first` phép so sánh và nhiều nhất là `last - first` phép gán.

5.2.2.7. Khuôn hình giải thuật replace_copy và replace_copy_if

Nguyên mẫu:

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator
last,
OutputIterator result, const T& old_value, const T& new_value);
```

```
template <class InputIterator, class OutputIterator, class
Predicate, class T>
OutputIterator replace_copy_if(InputIterator first, InputIterator
last,OutputIterator result, Predicate pred,const T& new_value)
```

Như tên gọi, khuôn hình giải thuật `replace_copy` vừa làm nhiệm vụ sao chép vừa làm nhiệm vụ thay thế. Khi cần phải sao chép một số đối tượng từ bộ chứa này sang bộ chứa khác, đồng thời một số đối tượng đặc biệt sẽ không được sao chép mà thay thế, ta dùng `replace_copy`. Khuôn hình giải thuật này sao chép các phần tử trong khoảng `[first,last)` sang `[result + (firstfirst - last))` nhưng với những bộ duyệt `i` mà `*i == old_value` thì không sao chép mà thay bằng `new_value`. Nghĩa là với mọi `n` sao cho `0 ≤ n < last - first`, `replace_copy` thực hiện phép gán `*(result + n) = new_value` nếu và chỉ nếu `*(first + n) == old_value`, nếu không `replace_copy` thực hiện phép gán `*(result + n) = *(first + n)`.

Khuôn hình giải thuật `replace_copy_if` làm việc tương tự `replace_copy` nhưng thay điều kiện kiểm tra `*(first + n) == old_value` bằng điều kiện kiểm tra `predpred(*(first + n)) == truetrue`.

Chương trình dưới đây minh họa hai giải thuật trên.

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

void main()
{
     int a[6] = {1, 2, 3, 4, 5, 6};
     ostream_iterator<int> oi(cout, " ");
     cout << "a: "; copy(a,a+6,oi); cout << endl;

     cout << "a: ";
     replace_copy_if(a,a+6,oi,bind2nd(less<int> (),5),5);
     cout << endl;

     cout << "a: ";
     replace_copy(a,a+6,oi,5,6);
     cout << endl;
```

```
}
```

```
a: 1 2 3 4 5 6
a: 5 5 5 5 5 6
a: 1 2 3 4 6 6
```

Nhận xét rằng kết quả ra ở hàng thứ hai giống với kết quả đưa ra trong ví dụ về `replace` và `replace_if`, nhưng kết quả ra ở hàng thứ 3 lại không giống. Lý do ở chỗ `replace_copy` và `replace_copy_if` thực hiện sao chép lên dãy khác (ở đây là ra màn hình), nên sau khi gọi `replace_copycopy_if`, dãy `a` vẫn là `1, 2, 3, 4, 5, 6` chứ không phải là `5, 5, 5, 5, 5, 6` như trên. Vậy để tránh nhầm lẫn, chương trình nên viết lại như sau:

```
    cout << "b: ";
    replace_copy_if(a,a+6,oi,bind2nd(less<int> (),5),5);
    cout << endl;

    cout << "c: ";
    replace_copy(a,a+6,oi,5,6);
    cout << endl;
```

```
a: 1 2 3 4 5 6
b: 5 5 5 5 5 6
c: 1 2 3 4 6 6
```

Độ phức tạp của `replace_copy` và `replace_copy_if` đều là tuyến tính, cả hai đều phải thực hiện `last - first` phép so sánh và `last - first` phép gán.

5.2.2.8. Khuôn hình giải thuật fill và fill_n

Nguyên mẫu

```
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T&
value);
```

```
template <class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T&
value);
```

Ta đã sử dụng `fill` khá nhiều lần trong các ví dụ minh họa cho các giải thuật khác. Khuôn hình giải thuật `fill` gán giá trị `value` cho mọi phần tử trong khoảng `[first,last)`. Ví dụ :

```
vector<int> v(10);
fill(v.begin(),v.end(),1);
```

sẽ tạo ra `vector v` với 10 phần tử có giá trị 1. Có thể thấy khi sử dụng `fill` với một bộ chứa, chẳng hạn `vector` trong ví dụ trên, bộ chứa đó phải khởi tạo kích thước trước. Trong ví dụ trên, nếu viết:

```
vector<int> v;
```

sẽ không sai về mặt cú pháp nhưng, khi chạy chương trình, `fill` sẽ không thực hiện phép gán nào hết.

Tương tự như `fill`, khuôn hình giải thuật `fill_n` cũng thực hiện phép gán giá trị `value` cho mọi phần tử trong dãy `[firstfirst, first + n)`. Ví dụ đoạn mã sau:

```
vector<int> v(10);
fill_n(v.begin(),5,2);
copy(v.begin(),v.end(),ostream_iterator<int> (cout," "));
```

cho kết quả:

```
2 2 2 2 2 0 0 0 0 0
```

Khi dùng `fill_n` chỉ cần chú ý `[firstfirst,first+n)` phải là khoảng hợp lệ. Nếu điều kiện này không thỏa mãn, khi chương trình chạy sẽ gây lỗi. Tương tự như `fill`, kích thước của bộ chứa cũng cần khởi tạo trước khi áp dụng `fill_n`.

Độ phức tạp của `fill` và `fill_n` cùng là tuyến tính, `fill_n` thực hiện tất cả n phép gán trong khi `fill` thực hiện `last - first` phép gán.

5.2.2.9. Khuôn hình giải thuật generate và generate_n

Nguyên mẫu:

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

```
template <class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator
gen);
```

Khuôn hình giải thuật generate cũng là một trong các giải thuật sinh số như fill nhưng tiện lợi hơn. Trong khi fill chỉ sinh các số giống hệt nhau, generate sử dụng đối tượng hàm gen có thể sinh các số ngẫu nhiên hoặc các số theo một quy luật nào đó do gen định ra. Với mọi bộ duyệt i thuộc [first,last), generate thực hiện phép gán *i = gen(*i). Khuôn hình giải thuật generate_n tương tự thực hiện phép gán *i = gen(*i) với mọi i thuộc [firstfirst,first+n).

Chương trình sau dùng generate để sinh một bộ số nguyên ngẫu nhiên và dùng generate_n để sinh bộ số Fibonacci.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <stdlib.h>
#include <time.h>
using namespace std;

class Fibonacci
{
public:
     Fibonacci () : v1 (0), v2 (1) {}
     int operator () ()
     {
            int r = v1 + v2;
            v1 = v2;
            v2 = r;
            return v1;
     };
private:
    int v1;
    int v2;
};

void main()
{
     srand( (unsigned)time( NULL ) );
     ostream_iterator<int> oi(cout," ");
```

```
    vector<int> v(5),y(10);

    generate(v.begin(),v.end(),rand);
    cout << "v: "; copy(v.begin(),v.end(),oi); cout << endl;

    Fibonacci f;
    generate_n(y.begin(),10,f);
    cout << "y: "; copy(y.begin(),y.end(),oi); cout << endl;
}
```

```
v: 10559 29595 19281 13698 8677
y: 1 1 2 3 5 8 13 21 34 55
```

Chú ý để dùng `rand` và `srand` khi sinh số ngẫu nhiên cần khai báo hai thư viện là `stdlib.h` và `time.h`. Tuy `rand` không phải là đối tượng hàm, nhưng có thể dùng như đối tượng hàm trong chương trình trên. Điều này đã được nhắc đến trong mục 2.2.2 trước khi giới thiệu các giải thuật.

Khuôn hình giải thuật `generate` và `generate_n` có độ phức tạp tuyến tính. `generate` cần gọi `last - first` lần toán tử gọi hàm của `gen` trong khi `generate_n` cần gọi n lần.

5.2.2.10. Khuôn hình giải thuật remove và remove_if

Nguyên mẫu:

```
template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator
last,
                        const T& value);
```

```
template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator
last,Predicate pred);
```

Xóa các phần tử trong một dãy là thao tác rất hay gặp trong khi giải quyết những bài toán thực tế. Hai khuôn hình giải thuật `remove` và `remove_if` sẽ giúp người lập trình trong thao tác này. Khuôn hình giải thuật thứ nhất, `remove` xóa khỏi `[first,last)` các phần tử có giá trị bằng `value`. Kết thúc, `remove` trả về bộ duyệt `new_last` sao cho trong `[firstfirst,new_last)` không tồn tại bộ duyệt `i` mà `*i == value`. Chú ý rằng các bộ duyệt trong khoảng `[new_last, last)` vẫn có thể lấy

tham chiếu nhưng giá trị chúng trỏ tới là không xác định. Khuôn hình giải thuật thứ hai, `remove_if` xóa khỏi `[first,last)` các phần tử có giá trị x nếu `predpred(x) == truetrue`.

Hai khuôn hình giải thuật `remove` và `remove_if` đều không làm thay đổi thứ tự của các phần tử. Chỉ những phần tử thỏa mãn điều kiện của hàm mới bị xóa khỏi dãy. Cần lưu ý, từ xóa dễ gây nhầm lẫn rằng kích thước của bộ chứa bị thay đổi. Thực tế bộ chứa vẫn có kích thước như thế, nhưng các phần tử tính từ `new_last` trở đi là không xác định. Chương trình dưới đây minh họa hai khuôn hình giải thuật này.

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

class gen
{
private:
     int s;
public:
     gen(int sum) : s(sum-1){};
     int operator() ()
     {
            s++;
            return s;
     }
};
void main()
{
     vector<int> a(10);
     generate_n(a.begin() + 5,5,gen(6));
     fill(a.begin(),a.begin()+5,5);
     ostream_iterator<int> oi(cout, " ");
     cout << "a: "; copy(a.begin(),a.end(),oi); cout << endl;

     vector<int>::iterator new_last =
remove(a.begin(),a.end(),5);
     cout << "a sau remove: "; copy(a.begin(),new_last,oi);
     cout << "\nsize(a): "; cout << a.size() << endl;

     new_last =
remove_if(a.begin(),new_last,bind2nd(equal_to<int> (),8));
     cout << "a sau remove_if: "; copy(a.begin(),new_last,oi);
```

```
    cout << "\nsize(a): "; cout << a.size() << endl;
}
```

```
a: 5 5 5 5 5 6 7 8 9 10
a sau remove: 6 7 8 9 10
size(a): 10
a sau remove_if: 6 7 9 10
size(a): 10
```

5.2.2.11. Khuôn hình giải thuật remove_copy và remove_copy_if

Nguyên mẫu hàm:

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator
last,
                           OutputIterator result, const T&
value);
```

```
template <class InputIterator, class OutputIterator, class
Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator
last,OutputIterator result, Predicate pred);
```

`remove_copy` sao chép các phần tử có giá trị khác `value` trong khoảng `[first,last)` sang khoảng khác bắt đầu từ result. Thứ tự các phần tử trong `[first,last)` khi sao chép sang không bị thay đổi.

`remove_copy_if` sao chép các phần tử trong khoảng `[first,last)` sang khoảng khác bắt đầu từ result trừ các phần tử ứng với bộ duyệt `i` sao cho `pred(*i) == true`. Thứ tự các phần tử trong `[first,last)` khi sao chép sang không bị thay đổi.

Cả hai `khuôn hình giải thuật` đều yêu cầu result không được thuộc khoảng `[first,last)`. Ví dụ sau minh họa hai hàm `remove_copy` và `remove_copy_if`.

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

void main ()
```

```
{
    int a[6] = { 1, 2, 3, 1, 2, 3 };
    int b[6] = { 12, 10, 8, 6, 4, 2 };
    ostream_iterator<int> oi(cout," ");

    copy(a,a+6,oi); cout << endl;
    // sao chep cac so khac 1 ra man hinh
    remove_copy(a,a+6,oi,1); cout << endl;

    copy(b,b+6,oi); cout << endl;
    // sao chep cac so nho hon 8 ra man hinh
    remove_copy_if(b,b+6,oi,bind2nd(less<int> (),8));

}
```

```
1 2 3 1 2 3
2 3 2 3
12 10 8 6 4 2
12 10 8
```

5.2.2.12. unique và unique_copy

Nguyên mẫu:

```
template <class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator
last);
```

```
template <class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator
last,
                       BinaryPredicate binary_pred);
```

Khuôn hình giải thuật `unique` thực hiện thao tác xóa khỏi dãy `[first,last)` các phần tử giống nhau liên tiếp, trừ phần tử đầu tiên, sau đó trả lại bộ duyệt `new_last` sao cho `[firstfirst,new_last)` không còn các phần tử giống nhau liên tiếp. Nghĩa là dãy `{1, 1, 1, 2, 2, 1, 1}` sau khi áp dụng `unique` sẽ trở thành `{1,2,1}`. Các phần tử trong khoảng `[new_last,last)` vẫn có thể lấy tham chiếu nhưng giá trị chúng trỏ tới là không xác định. `unique` không làm thay đổi thứ tự các phần tử trước và sau khi xóa. Hai hàm nạp chồng của `unique` chỉ khác nhau ở chỗ một hàm dùng toán tử `operator==` mặc định khi so sánh, một hàm dùng đối tượng hàm mệnh đề `binary_pred`.

```
    unique có độ phức tạp tuyến tính. Hàm thứ nhất thực hiện
(last - first) - 1 phép so sánh, hàm thứ hai thực hiện (last -
first) - 1 lần gọi đối tượng hàm binary_pred.template <class
InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator
last,
                           OutputIterator result);
```

```
template <class InputIterator, class OutputIterator, class
BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator
last,
                           OutputIterator result, BinaryPredicate
binay_pred);
```

Khuôn hình giải thuật `unique_copy` sao chép từ dãy `[first,last)` sang dãy bắt đầu từ `result` nhưng đối với các phần tử giống nhau liên tiếp, `unique_copy` chỉ sao chép phần tử đầu tiên. Giá trị trả về là `result_last` sao cho dãy `[result,result_last)` không có hai phần tử liên tiếp bằng nhau. Hai khuôn hình giải thuật `unique_copy` chỉ khác nhau ở chỗ một hàm dùng toán tử `operator==` mặc định khi so sánh, một hàm dùng đối tượng hàm mệnh đề `binary_pred`.

Độ phức tạp của `unique_copy` là tuyến tính. Có tất cả `last - first` phép so sánh và nhiều nhất `last - first` phép gán.

5.2.2.13. reverse và reverse_copy

Nguyên mẫu:

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator
last);
```

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
BidirectionalIterator last, OutputIterator result);
```

Khuôn hình giải thuật `reverse` đảo ngược thứ tự các phần tử trong dãy `[first,last)`. Nghĩa là dãy `{1,2,3}` sau khi áp dụng `reverse` sẽ trở thành `{3,2,1}`.

reverse có độ phức tạp tuyến tính, reverse gọi tới khuôn hình giải thuật swap chính xác (last - first) / 2 lần.

Khuôn hình giải thuật reverse_copy sao chép các phần tử từ dãy [first,last) sang dãy bắt đầu từ result sao cho thứ tự các phần tử trong dãy kết quả ngược lại với thứ tự các phần tử trong dãy ban đầu.

Khi dùng reverse_copy phải chú ý hai dãy [first,last) và [result,result+(last - first)) không được chồng nhau.

reverse_copy có độ phức tạp tuyến tính. Dễ thấy reverse_copy thực hiện tất cả last - first phép gán.

5.2.2.14. rotate và rotate_copy

```
Nguyên mẫu:template <class ForwardIterator>
inline ForwardIterator rotate(ForwardIterator first,
ForwardIterator middle,ForwardIterator last);
```

```
template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator
middle, ForwardIterator last, OutputIterator result);
```

Khuôn hình giải thuật rotate hoán chuyển vị trí các phần tử trong [first,last) sao cho [first,middle) thế chỗ [middle,last) và ngược lại. Xem xét ví dụ sau:

```
char X[] = "Standard-template-library";
cout << X << endl;
rotate(X,X+9,X+25);
cout << X << endl;
```

```
Standard-template-library
template-libraryStandard-
```

rotate có độ phức tạp tuyến tính, nó thực hiện nhiều nhất last - first phép hoán chuyển vị trí các phần tử.

rotate_copy cũng tương tự như rotate, nhưng thay vì thực hiện trên chính dãy [first,last), rotate_copy sao chép kết quả sang dãy khác bắt đầu từ result. Điều kiện để thực hiện rotate_copy là hai khoảng [first,last) và [result, result + (first - last)) không được chồng nhau.

rotate_copy có độ phức tạp tuyến tính, nó thực hiện tất cả last - first lần sao chép phần tử.

5.2.2.15. random_shuffle

Nguyên mẫu:

```
template <class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first,
RandomAccessIterator last);
```

```
template <class RandomAccessIterator, class
RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first,
RandomAccessIterator last, RandomNumberGenerator& rand)
```

Khuôn hình giải thuật random_shuffle tạo một bộ hoán vị ngẫu nhiên giữa các phần tử của dãy [first,last). Ví dụ, nếu có dãy {1,2,3,4} thì sau khi áp dụng random_shuffle ta có thể được dãy {2,3,4,1} hoặc một dãy bất kì khác trong 4! hoán vị của dãy ban đầu. Thông thường, ta không mấy khi cần đến hàm nạp chồng thứ hai của random_shuffle. Nếu sử dụng hàm này ta phải cài đặt một đối tượng hàm rand có nhiệm vụ sinh bộ số ngẫu nhiên, trong khi C++ đã có sẵn một hàm tương tự như vậy và được sử dụng trong hàm thứ nhất rồi. Ví dụ sau minh họa khuôn hình giải thuật random_shuffle thứ nhất.

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>
#include <time.h>
using namespace std;
```

```
class Object
{
private:
    int id;
public:
    Object(int identify) : id(identify) {}
    void print_id ()
    {
```

```
        cout << "My id is: " << id << endl;
    }
};
void main()
{
    srand( (unsigned)time( NULL ) );

    vector<Object*> v;
    v.push_back(new Object(1));
    v.push_back(new Object(2));
    v.push_back(new Object(3));
    v.push_back(new Object(4));

    // in id cua cac doi tuong
    for_each(v.begin(),v.end(),mem_fun(Object::print_id));
    // hoan vi cac doi tuong
    random_shuffle(v.begin(),v.end());
    cout << "Cac doi tuong da hoan vi!" << endl;
    // in id cua cac doi tuong
    for_each(v.begin(),v.end(),mem_fun(Object::print_id));
}
```

```
My id is: 1
My id is: 2
My id is: 3
My id is: 4
Cac doi tuong da hoan vi!
My id is: 2
My id is: 4
My id is: 3
My id is: 1
```

Chương trình trên thực hiện việc hoán vị các đối tượng kiểu Object* của `vector v` bằng `random_shuffle`. Chú ý rằng do `random_shuffle` sử dụng hàm rand của thư viện C++ nên trước khi dùng `random_shuffle` ta nên có dòng lệnh `srand( (unsigned)time( NULL ) )`; để mỗi lần chạy sẽ cho một bộ hoán vị khác nhau. Hàm `mem_fun` tạo đối tượng hàm từ hàm thành phần đã được giới thiệu trong chương về đối tượng hàm. Nếu bạn đọc đã quên, nên xem lại mục 3.6.1.

`random_shuffle` có độ phức tạp tuyến tính. Nếu `last != first`, `random_shuffle` thực hiện tất cả `(last - first) - 1` phép hoán chuyển phần tử.

5.2.2.16. partition và stable_patition

```
Nguyên mẫu:template <class ForwardIterator, class Predicate>
ForwardIterator partition(ForwardIterator first,
ForwardIterator last, Predicate pred)
```

```
template <class ForwardIterator, class Predicate>
ForwardIterator stable_partition(ForwardIterator first,
ForwardIterator last, Predicate pred);
```

Khuôn hình giải thuật `partition` thay đổi lại thứ tự các phần tử trong dãy `[first,last)` dựa trên đối tượng hàm mệnh đề `pred` sao cho các phần tử thỏa mãn `pred` sẽ đứng trước các phần tử không thỏa mãn `pred`. Kết quả trả về của `pred` là bộ duyệt `middle` sao cho mọi bộ duyệt `i` thuộc dãy `[first, middle ), pred(*i) == true` và mọi bộ duyệt `j` thuộc dãy `[middle,last), pred(*j) == falsefalse`. Ví dụ sau sắp xếp các số nguyên sao cho số chẵn đứng trước, số lẻ đứng sau:

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

void main()
{
    int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    const int N = sizeof(A)/sizeof(int);
    partition(A, A + N,
          compose1(bind2nd(equal_to<int>(), 0),
                   bind2nd(modulus<int>(), 2)));
    copy(A, A + N, ostream_iterator<int>(cout, " "));

}
```

```
10 2 8 4 6 5 7 3 9 1
```

Khuôn hình giải thuật `stable_partition` tương tự như `partition` nhưng thứ tự của các phần tử thỏa mãn `pred` đứng trước `middle` và thứ tự của các phần tử không thỏa mãn `pred` đứng sau `middle` không thay đổi so với dãy ban đầu. Nghĩa là nếu chạy ví dụ trên với `stable_partition` ta sẽ có kết quả:

```
2 4 6 8 10 1 3 5 7 9
```

Độ phức tạp của partition là tuyến tính, trong khi đó stable_partition lại phức tạp hơn. partition thực hiện last - first phép so sánh với pred và nhiều nhất (last - first) / 2 phép hoán chuyển. stable_partition cũng thực hiện last - first phép so sánh với pred, nhưng thực hiện N*log(N) phép hoán chuyển trong trường hợp tồi nhất và N phép hoán chuyển trong trường hợp tốt nhất, trong đó N = last - first).

5.2.3. Các giải thuật sắp xếp

Các giải thuật sắp xếp trong STL giúp người lập trình bớt được rất nhiều công sức trong các ứng dụng thực tiễn đòi hỏi các bài toán liên quan đến sắp xếp. Trong các giải thuật được giới thiệu dưới đây, mỗi hàm đều có hai hàm nạp chồng. Một hàm dùng toán tử so sánh operator<, hàm kia dùng đối tượng hàm so sánh comp. Các hàm nạp chồng dùng operator< cho kết quả là các dãy được sắp xếp theo thứ tự tăng dần. Nếu muốn so sánh với các tiêu chí khác có thể sử dụng các hàm nạp chồng dùng đối tượng hàm so sánh comp. Chú ý, khái niệm tăng dần sử dụng ở đây có ý nghĩa như sau: nếu bộ duyệt i đứng trước bộ duyệt j trong dãy thì *j < *i là sai. Điều này không có nghĩa là *i < *j hay *i ≤ *j. Đối với các hàm nạp chồng sử dụng đối tượng hàm so sánh, khái niệm tăng dần có nghĩa là comp(*j, *i) == false. Bạn đọc cần lưu ý điều này vì chúng ta thường có thói quen hình dung việc so sánh với các số nguyên trong khi các giải thuật sắp xếp của STL làm việc được với rất nhiều kiểu và đối tượng. Cần nhắc lại là, khái niệm tăng dần có nghĩa là nếu i đứng trước j thì *j không nhỏ hơn *i hoặc comp(*j,*i) == false.

5.2.3.1. sort và stable_sort

Nguyên mẫu:

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

```
template <class RandomAccessIterator, class StrictWeakOrdering>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          StrictWeakOrdering comp);
```

Khuôn hình giải thuật `sort` sắp xếp các phần tử trong dãy `[first,last)` sao cho phần tử đứng sau không nhỏ hơn phần tử đứng trước. Nói chung, nếu không có những yêu cầu đặc biệt, `sort` là một giải thuật sắp xếp đủ mạnh cho các ứng dụng. Ta sẽ không phải mất công cài đặt `quicksort` hay một thuật toán sắp xếp nào mà có thể dùng luôn `sort` của STL là đủ. Giải thuật này sử dụng `introsort`, một thuật toán được đánh giá là nhanh ngang với `quicksort`.

Hàm nạp chồng thứ nhất của `sort` dùng toán tử `operator<` mặc định khi so sánh. Hàm nạp chồng thứ hai dùng đối tượng hàm hai tham số comp. Ta dùng hàm nạp chồng thứ hai khi muốn sắp xếp với các toán tử so sánh khác như `operator>`, `operator<=`, `operator>=`,... Riêng bộ chứa `list` của STL có hàm hàm thành phần `sort` hoạt động hiệu quả hơn và do vậy không nên dùng khuôn hình giải thuật `sort` với `list`. (Xem chương về bộ chứa, mục về `list`).

```
int A[] = {1, 4, 2, 8, 5, 7};
const int N = sizeof(A) / sizeof(int);
sort(A, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
```

```
1 2 4 5 7 8
```

Độ phức tạp của `sort` là `O(N.log(N))` trong cả trường hợp xấu nhất và trường hợp trung bình.

```
template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator
last);
```

```
template <class RandomAccessIterator, class StrictWeakOrdering>
void stable_sort(RandomAccessIterator first,
RandomAccessIterator last, StrictWeakOrdering comp);
```

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

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

class lt_nocase
{
public:
     lt_nocase() {}
     bool operator()(char c1, char c2)
     {
            return tolower(c1) < tolower(c2);
     }
};
int main()
{
     char A[] = "fdBeACFDbEac";
     const int N = sizeof(A) - 1;
     stable_sort(A, A+N, lt_nocase());
     cout << A << endl;
}
```

```
AaBbCcdDeEfF
```

Đối tượng hàm so sánh `lt_nocase` so sánh các kí tự không phân biệt chữ hoa, chữ thường. Rõ ràng khi so sánh với `lt_nocase` thì `a` không nhỏ hơn `A` và `A` cũng không nhỏ hơn `a` (nhưng không thể nói rằng `a` bằng `A`). Ta thấy rằng thứ tự tương đối của các kí tự `A`, `a` hay `B`, `b` là không đổi sau `stable_sort`. Điều này không đúng nếu ta dùng `sort`. `stable_sort` hữu dụng khi sắp xếp các bản ghi có nhiều trường. Ví dụ như khi sắp xếp tên người trong một danh sách. Ta có thể dùng `sort` để sắp xếp danh sách theo tên sau đó dùng `stable_sort` để sắp xếp lại danh sách theo họ.

Độ phức tạp của `stable_sort` trong trường hợp tồi nhất là `Nlog2(N)` và trường hợp tốt nhất là `Nlog(N)` với `N = last - first`. `stable_sort` sử dụng thuật toán sắp xếp trộn `merge_sort`.

5.2.3.2. partial_sort và partial_sort_copy

Nguyên mẫu:

```
template <class RandomAccessIterator>
void partial_sort(RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last);
```

```
template <class RandomAccessIterator, class StrictWeakOrdering>
void partial_sort(RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last,
                  StrictWeakOrdering comp);
```

partial_sort chỉ sắp xếp một phần dãy [first,last) cho đến middle. Nghĩa là (middle - first) phần tử nhỏ nhất sẽ được sắp xếp trong khoảng [firstfirst,middle), còn các phần tử trong khoảng [middle,last) không được sắp xếp và thứ tự của chúng là không xác định. Lý do dùng partial_sort là trong nhiều trường hợp chỉ cần sắp xếp một phần của dãy. Rõ ràng là dùng partial_sort sẽ nhanh hơn dùng sort. partial_sort sử dụng thuật toán sắp xếp vun đống heapsort và có độ phức tạp là NlogM trong đó N = last - first, M = middle - first.

Hai khuôn hình giải thuật partial_sort chỉ khác nhau ở điểm là một hàm dùng toán tử so sánh operatorsort và một hàm dùng đối tượng hàm comp để so sánh.

Ví dụ, sử dụng partial_sort:

```
int A[] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
const int N = sizeof(A) / sizeof(int);

partial_sort(A, A + 5, A + N);
copy(A, A + N, ostream_iterator<int>(cout, " "));
```

```
1 2 3 4 5 11 12 10 9 8 7 6
```

Nguyên mẫu của partial_sort_copy:

```
template <class InputIterator, class RandomAccessIterator>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last);
```

```
template <class InputIterator, class RandomAccessIterator,
          class StrictWeakOrdering>
```

```
RandomAccessIterator partial_sort_copy(InputIterator first,
InputIterator last,RandomAccessIterator result_first,
RandomAccessIterator result_last, Compare comp);
```

Khuôn hình giải thuật partial_sort_copy cũng tương tự như partial_sort nhưng sao chép kết quả lên dãy bắt đầu từ result_first. Nghĩa là partial_sort_copy sao chép N phần tử từ dãy [first,last) sang dãy [result_first, result_first + N) (trong đó N là số nhỏ hơn trong hai số last - first và result_last - result_first). Các phần tử trong dãy [result_first, result_first + N) được sắp xếp theo thứ tự tăng dần.

Độ phức tạp của partial_sort_copy xấp xỉ (last - first) log(N) phép tính, trong đó N là số nhỏ hơn trong hai số (last - first) và (result_last - result_first).

5.2.3.3. nth_element

Nguyên mẫu:

```
template <class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator
nth, RandomAccessIterator last);
```

```
template <class RandomAccessIterator, class StrictWeakOrdering>
void nth_element(RandomAccessIterator first, RandomAccessIterator
nth, RandomAccessIterator last, StrictWeakOrdering comp);
```

Như tên gọi, nth_element sắp xếp lại dãy [first,last) sao cho phần tử trỏ bởi bộ duyệt nth sẽ có cùng giá trị với phần tử ở vị trí tương ứng nếu dãy được sắp xếp tăng dần. Ngoài ra không phần tử nào đứng sau phần tử nth nhỏ hơn các phần tử đứng trước nth. Hai dãy [first,nth) và [nth,last) tuy thế không được bảo đảm là đã được sắp xếp theo thứ tự. Chúng chỉ có tính chất đã kể ở trên. Xét ví dụ sau:

```
int A[] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4, 1, 5};
const int N = sizeof(A) / sizeof(int);
nth_element(A, A + 6, A + N);
```

```
copy(A, A + N, ostream_iterator<int>(cout, " "));5 2 6 1 4 3 7
8 9 10 11 12
```

Có thể thấy phần tử thứ 6 (đánh chỉ số từ 0) có giá trị 7 (Dãy trên khi sắp xếp sẽ là 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12) và các phần tử sau 7 không có phần tử nào nhỏ hơn các phần tử trước 7.

(Chương trình trên khi biên dịch với g++ trên hệ điều hành Linux mới cho kết quả như trên trong khi biên dịch với VC6.0 hoặc Visual.NET trên Windows đều cho kết quả là một dãy đã sắp xếp. Đây chỉ là sự trùng hợp ngẫu nhiên và kết quả như thế vẫn hoàn toàn đúng. Nếu bạn thực sự muốn thấy hai dãy sau và trước `nth` không được sắp xếp, hãy chạy thử chương trình với số lượng phần tử lớn hơn và nth xê dịch về một trong hai phía càng xa càng tốt).

Tính trung bình, độ phức tạp của `nth` là tuyến tính với `last - first`.

5.2.3.4. lower_bound

Nguyên mẫu:

```
template <class ForwardIterator, class LessThanComparable>
ForwardIterator lower_bound(ForwardIterator first,
ForwardIterator last,const LessThanComparable& value);
```

```
template <class ForwardIterator, class T, class
StrictWeakOrdering>
ForwardIterator lower_bound(ForwardIterator first,
ForwardIterator last,const T& value, StrictWeakOrdering comp);
```

Cho trước một dãy đã sắp xếp tăng dần `[first,last)`, `lower_bound` sẽ tìm ra vị trí đầu tiên trong dãy có thể điền được `value` vào mà không làm hỏng tính chất được sắp xếp của dãy.

Khuôn hình giải thuật `lower_bound` thứ nhất trả về bộ duyệt `i` xa nhất có tính chất mọi bộ duyệt `j` thuộc `[firstfirst,i)`, `*j < value`. Đối với hàm nạp chồng thứ hai, tính chất trên sẽ là: mọi bộ duyệt `j` thuộc `[firstfirst,i)`, `comp(*j, value) == true`. Chú ý rằng giá trị trả về là bộ duyệt `i`, nhưng vị trí có thể chèn được `value` vào là vị trí trước `i` Nếu `value` lớn hơn mọi số trong dãy, giá trị trả về sẽ là bộ duyệt `last`, ngược lại nếu `value` nhỏ hơn mọi số trong dãy, giá trị trả về sẽ là bộ duyệt `first`.

Độ phức tạp của lower_bound phụ thuộc vào KHÁI NIỆM của bộ duyệt. Nếu tham số khuôn hình cho bộ duyệt thuộc KHÁI NIỆM Random Access Iterator, lower_bound thực hiện log(last - first) + 1 phép so sánh. Còn nếu chỉ là Forward Iterator, số phép so sánh sẽ tỉ lệ với (last - first).

5.2.3.5. upper_bound

Nguyên mẫu:

```
template <class ForwardIterator, class LessThanComparable>
ForwardIterator upper_bound(ForwardIterator first,
ForwardIterator last, const LessThanComparable& value);
```

```
template <class ForwardIterator, class T, class
StrictWeakOrdering>
ForwardIterator upper_bound(ForwardIterator first,
ForwardIterator last, const T& value, StrictWeakOrdering comp);
```

Cho trước một dãy đã sắp xếp tăng dần [first,last), upper_bound sẽ tìm ra vị trí cuối cùng trong dãy có thể điền được value vào mà không làm hỏng tính chất sắp xếp của dãy.

Khuôn hình giải thuật upper_bound thứ nhất trả về bộ duyệt i xa nhất có tính chất mọi bộ duyệt j thuộc [firstfirst,i), value không nhỏ hơn *j. Đối với hàm nạp chồng thứ hai mọi bộ duyệt j thuộc [firstfirst,i), comp(value, *j) == false. Chú ý rằng giá trị trả về là bộ duyệt i nhưng vị trí có thể chèn được value vào là vị trí trước i. Nếu value lớn hơn mọi số trong dãy, giá trị trả về sẽ là bộ duyệt last.

Độ phức tạp của upper_bound phụ thuộc vào KHÁI NIỆM của bộ duyệt. Nếu tham số khuôn hình cho bộ duyệt thuộc KHÁI NIỆM Random Access Iterator, upper_bound thực hiện log(last - first) + 1 phép so sánh. Còn nếu chỉ là Forward Iterator, số phép so sánh sẽ tỉ lệ với (last - first).

5.2.3.6. equal_range

Nguyên mẫu:

```
template <class ForwardIterator, class LessThanComparable>
pair<ForwardIterator, ForwardIterator>
```

```
equal_range(ForwardIterator first, ForwardIterator last,
            const LessThanComparable& value);
```

```
template <class ForwardIterator, class T, class
StrictWeakOrdering>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T&
value,     StrictWeakOrdering comp);
```

Có thể cho rằng `equal_range` là hợp của `lower_bound` và `upper_bound`. Khuôn hình giải thuật này đồng thời tìm vị trí đầu tiên và cuối cùng trong dãy đã sắp xếp `[first,last)` sao cho có thể chèn thêm `value` vào mà không làm ảnh hưởng tới thứ tự đã sắp xếp của dãy. Cặp bộ duyệt trả về `(i,j)` có tính chất sau:

- Mọi bộ duyệt `k` thuộc `[firstfirst, i)`, `*k < value`. (Tính chất của `lower_bound`)
- Mọi bộ duyệt `k` thuộc `[firstfirst, j)`, `value` không nhỏ hơn `*k`. (Tính chất của `upper_bound`)

Mọi bộ duyệt `k` thuộc `[i,j)`, `value` không nhỏ hơn `*k` và `*k` cũng không nhỏ hơn `value` (chú ý rằng như thế chưa chắc đã là `value == *k !`). (Suy ra từ hai tính chất trên).Độ phức tạp của `equal_range` phụ thuộc vào KHÁI NIỆM của bộ duyệt. Nếu tham số khuôn hình cho bộ duyệt thuộc KHÁI NIỆM Random Access Iterator, `equal_range` thực hiện `2*log(last - first) + 1` phép so sánh còn nếu chỉ là Forward Iterator, sẽ số phép so sánh tỉ lệ với `(last - first)`.

5.2.3.7. binary_search

Nguyên mẫu:

```
template <class ForwardIterator, class LessThanComparable>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const LessThanComparable& value);
```

```
template <class ForwardIterator, class T, class
StrictWeakOrdering>
bool binary_search(ForwardIterator first, ForwardIterator last,
const T& value,          StrictWeakOrdering comp);
```

Như tên gọi, binary_search cài đặt thuật toán tìm kiếm nhị phân. Khuôn hình giải thuật binary_search tìm kiếm trong dãy [first,last) một phần tử *tương đương* với value và trả về true nếu tìm thấy và trả về false nếu ngược lại. Chú ý rằng, phần tử tìm thấy x tương đương với value có nghĩa là x không nhỏ hơn value và value cũng không nhỏ hơn x. Do vậy KHÁI NIỆM cho value chỉ là Less Than Comparable chứ không phải Equality Comparable. Hai hàm nạp chồng của binary_search chỉ khác nhau là một hàm dùng toán tử so sánh operator< một hàm dùng đối tượng hàm comp.

Nói chung, khi tìm kiếm một phần tử trong một dãy, ta thường muốn biết nhiều thông tin hơn là chỉ biết sự tồn tại của nó như với binary_search. Trong những trường hợp như vậy, dùng lower_bound, upper_bound hay equal_range sẽ hiệu quả hơn (Các giải thuật này cũng sử dụng thuật toán tìm kiếm nhị phân).

Độ phức tạp: nhiều nhất binary_search thực hiện log(last-first) + 2 phép so sánh. Nếu KHÁI NIỆM của bộ duyệt là Random Access Iterator, độ phức tạp của binary_search là hàm log như trên. Nếu chỉ là Forward Iterator, độ phức tạp của binary_search là tuyến tính và tỉ lệ với (last - first).

5.2.3.8. merge

Nguyên mẫu:

```
template <class InputIterator1, class InputIterator2, class
OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
```

```
template <class InputIterator1, class InputIterator2, class
OutputIterator, class StrictWeakOrdering>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, StrictWeakOrdering
comp);
```

Khuôn hình giải thuật merge trộn hai dãy đã sắp xếp [firstfirst1,last1) và [firstfirst2,last2) theo thứ tự tăng dần

thành một dãy bắt đầu từ `result` và dãy này cũng được sắp xếp theo thứ tự tăng dần. Thứ tự tương đối của các phần tử trong dãy kết quả không bị thay đổi so với thứ tự của chúng ở trong hai dãy ban đầu.

Hàm nạp chồng thứ nhất của `merge` sử dụng toán tử so sánh `operator<` mặc định trong khi hàm thứ hai sử dụng toán tử so sánh comp. Với hàm thứ hai ta có thể dùng khi muốn có dãy sắp xếp giảm dần chẳng hạn.

Độ phức tạp của `merge` là tuyến tính, nhiều nhất `(last1 - first1) + (last2 - first2) - 1` phép so sánh được thực hiện.

5.2.3.9. inplace_merge

Nguyên mẫu:

```
template <class BidirectionalIterator>
inline void inplace_merge(BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last);
```

```
template <class BidirectionalIterator, class StrictWeakOrdering>
inline void inplace_merge(BidirectionalIterator first,
                          BidirectionalIterator middle,
                          BidirectionalIterator last,
                          StrictWeakOrdering comp);
```

`inplace_merge` trộn hai dãy đã sắp xếp liền nhau `[firstfirst, middle)` và `[middle, last)` thành một dãy `[first, last)` được sắp xếp.

5.2.3.10. min và min_element

Nguyên mẫu:

```
template <class T> const T& min(const T& a, const T& b);
```

```
template <class T, class BinaryPredicate>
const T& min(const T& a, const T& b, BinaryPredicate comp);
```

Khuôn hình giải thuật `min` trả về đối tượng nhỏ nhất trong hai đối tượng `a` và `b`. Hàm thứ nhất sử dụng toán tử so sánh `operator<` và trả về `a` nếu `a < b`. Hàm thứ hai sử dụng đối tượng hàm so sánh `comp` và trả về

b nếu comp(b,a) == true và trả về a nếu ngược lại. Chú ý là khi ta gọi min(a,b), trong min sẽ gọi đến comp(b,a).

```
template <class ForwardIterator>
ForwardIterator min_element(ForwardIterator first,
ForwardIterator last);
```

```
template <class ForwardIterator, class BinaryPredicate>
ForwardIterator min_element(ForwardIterator first,
ForwardIterator last, BinaryPredicate comp);
```

Khuôn hình giải thuật min_element trả về đối tượng nhỏ nhất trong dãy [first,last). min_element trả về bộ duyệt i đầu tiên sao cho không tồn tại j thuộc [first,last) mà *j nhỏ hơn *i. Hàm thứ nhất sử dụng toán tử so sánh operator<, hàm thứ hai sử dụng đối tượng hàm so sánh comp.

5.2.3.11. max và max_element

Nguyên mẫu:

```
template <class T> const T& max(const T& a, const T& b);
```

```
template <class T, class BinaryPredicate>
const T& max(const T& a, const T& b, BinaryPredicate comp);
```

Khuôn hình giải thuật max trả về đối tượng lớn hơn trong hai đối tượng a và b. Hàm thứ nhất sử dụng toán tử so sánh operator>, hàm thứ hai sử dụng đối tượng hàm so sánh comp. Hàm thứ hai sử dụng đối tượng hàm so sánh comp và trả về b nếu comp(b,a) == true và trả về a nếu ngược lại. Chú ý là khi ta gọi max(a,b), trong max sẽ gọi đến comp(b,a).

```
template <class ForwardIterator>
ForwardIterator max_element(ForwardIterator first,
ForwardIterator last);
```

```
template <class ForwardIterator, class BinaryPredicate>
ForwardIterator max_element(ForwardIterator first,
ForwardIterator last, BinaryPredicate comp);
```

Khuôn hình giải thuật `max_element` trả về đối tượng lớn nhất trong dãy `[first,last)`. `max_element` trả về bộ duyệt `i` đầu tiên sao cho không tồn tại `j` thuộc `[first,last)` mà `*j` nhỏ hơn `*i`. Hàm thứ nhất sử dụng toán tử so sánh `operator>`, hàm thứ hai sử dụng đối tượng hàm so sánh `comp`.

5.2.4. Các giải thuật trên tập hợp

5.2.4.1. includes

Nguyên mẫu:

```
template <class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);
```

```
template <class InputIterator1, class InputIterator2, class
StrictWeakOrdering>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              StrictWeakOrdering comp);
```

Khuôn hình giải thuật `includes` kiểm tra xem dãy đã sắp xếp `[firstfirst1, last1)` có chứa dãy đã sắp xếp `[firstfirst2, last2)` hay không. Yêu cầu của `includes` nói riêng và các giải thuật trên tập hợp nói chung là các dãy đều phải là các dãy đã sắp xếp thứ tự tăng dần. Hai hàm nạp chồng trên khác nhau ở cách chúng xác định khi nào một phần tử nhỏ hơn một phần tử khác. Hàm thứ nhất sử dụng toán tử so sánh mặc định `operator<`, hàm thứ hai sử dụng đối tượng hàm so sánh `comp` thuộc KHÁI NIỆM Strict Weak Ordering.

5.2.4.2. set_union

Nguyên mẫu:

```
template <class InputIterator1, class InputIterator2, class
OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1
last1,
InputIterator2 first2, InputIterator2 last2,OutputIterator
result);
```

```
template <class InputIterator1, class InputIterator2, class
OutputIterator, class StrictWeakOrdering>
OutputIterator set_union(InputIterator1 first1, InputIterator1
last1,
                         InputIterator2 first2, InputIterator2
last2,
                         OutputIterator result,
                         StrictWeakOrdering comp);
```

Khuôn hình giải thuật `set_union` tạo ra một dãy sắp xếp là hợp của hai dãy sắp xếp `[firstfirst1,last1)` và `[firstfirst2,last2)`. Kết quả trả về là con trỏ đến vị trí sau cuối của dãy kết quả. Trong trường hợp đơn giản nhất, `set_union` hoạt động theo đúng như lý thuyết tập hợp: dãy kết quả sẽ chứa mọi phần tử có mặt ở một trong hai dãy `[first1,last1)`, `[first2,last2)` hoặc cả hai. Trong trường hợp tổng quát, khi một dãy có thể có nhiều phần tử *tương đương* nhau, kết quả ra sẽ không đơn giản như vậy. Nếu có một giá trị xuất hiện trong `[firstfirst1,last1)` `n` lần và trong `[firstfirst2,last2)` `m` lần, trong dãy kết quả nó sẽ xuất hiện `max(n,m)` lần.

Hãy xem xét ví dụ sau minh họa khuôn hình giải thuật `set_union`.

```
#include <iostream>
#include <algorithm>
using namespace std;
void main()
{
     int a[5] = {1,2,3,4,5};
     int b[5] = {3,4,5,6,7};
     ostream_iterator<int> oi(cout," ");

     cout << "a hop b: ";
     set_union(&a[0],&a[5],&b[0],&b[5],oi);
     cout << endl;

     int c[6] = {1,1,2,3,4,4};
     int d[6] = {1,1,1,2,3,4};

     cout << "c hop d: ";
     set_union(&c[0],&c[6],&d[0],&d[6],oi);
}
```

```
a hop b: 1 2 3 4 5 6 7
c hop d: 1 1 1 2 3 4 4
```

Ví dụ trên minh họa trường hợp thông thường (`a` hợp `b`) và trường hợp

tổng quát (c hợp d) khi sử dụng khuôn hình giải thuật set_union lên các tập hợp.

5.2.4.3. set_intersection

Nguyên mẫu:

```
template <class InputIterator1, class InputIterator2, class
OutputIterator>
OutputIterator set_intersection(InputIterator1 first1,
InputIterator1 last1,InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
```

```
template <class InputIterator1, class InputIterator2, class
OutputIterator,class StrictWeakOrdering>
OutputIterator set_intersection(InputIterator1 first1,
InputIterator1 last1,InputIterator2 first2, InputIterator2 last2,
OutputIterator result, StrictWeakOrdering comp);
```

Khuôn hình giải thuật set_intersection thực hiện phép giao tập hợp giữa hai khoảng [firstfirst1,last1) và [firstfirst2,last2).

5.2.4.4. set_difference

Nguyên mẫu:

```
template <class InputIterator1, class InputIterator2, class
OutputIterator>
OutputIterator set_difference(InputIterator1 first1,
InputIterator1 last1,InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
```

```
template <class InputIterator1, class InputIterator2, class
OutputIterator, class StrictWeakOrdering>
OutputIterator set_difference(InputIterator1 first1,
InputIterator1 last1,InputIterator2 first2, InputIterator2 last2,
OutputIterator result, StrictWeakOrdering comp);
```

5.2.4.5. set_symmetric_difference

Nguyên mẫu:

```
template <class InputIterator1, class InputIterator2, class
OutputIterator>
```

```
OutputIterator set_symmetric_difference(InputIterator1 first1,
                                        InputIterator1 last1,
                                        InputIterator2 first2,
                                        InputIterator2 last2,
                                        OutputIterator result);
```

```
template <class InputIterator1, class InputIterator2, class
OutputIterator,class StrictWeakOrdering>
OutputIterator set_symmetric_difference(InputIterator1 first1,
                                        InputIterator1 last1,
                                        InputIterator2 first2,
                                        InputIterator2 last2,
                                        OutputIterator result,
                                        StrictWeakOrdering comp);
```

5.3. Tóm tắt

5.3.1. Ghi nhớ

Khuôn hình giải thuật là các khuôn hình hàm cài đặt các thuật toán. Khuôn hình giải thuật có tính khái lược cao: khuôn hình giải thuật được tham số hóa, khuôn hình hóa một cách triệt để, làm việc không phụ thuộc vào dữ liệu cụ thể, có tính mềm dẻo nhất định trong xử lý. Người dùng không cần cài đặt lại các thuật toán quen thuộc, có thể dùng ngay khuôn hình giải thuật cho những mục đích của mình. Khuôn hình giải thuật là trung tâm của STL nói riêng và của một thư viện lập trình khái lược nói chung.

Giải thuật làm việc với bộ chứa thông qua bộ duyệt. Nhờ đó, giải thuật không bị phụ thuộc vào bộ chứa cụ thể. Với đối tượng hàm làm tác nhân bên trong, giải thuật có tính mềm dẻo trong xử lý.

Giải thuật trong STL có thể chia thành ba lớp chính. Lớp các giải thuật không làm đổi biến. Lớp này chủ yếu bao gồm các khuôn hình giải thuật duyệt hoặc tìm kiếm như: `for_each`, `find`, `count`, `search`,... Lớp thứ hai gồm các giải thuật làm đổi biến. Lớp này bao gồm các giải thuật sao chép, xóa, đổi chỗ như: `copy`, `transform`, `replace`, `remove`, `fill`,... Lớp thứ ba là các giải thuật sắp xếp: `sort`, `merge`, `equal_range`, `min`, `max`,... và các hàm giải thuật cho phép toán tập hợp. Các khuôn hình giải thuật trong tên có đuôi `_if` chỉ khác các khuôn hình giải thuật cùng tên không có đuôi `_if` ở cách so sánh. Thay cho toán tử so sánh mặc định (`operator==`, `operator<`), các hàm này dùng đối tượng hàm so sánh tương ứng. Nhờ vậy, tính mềm dẻo của giải thuật được tăng lên.

Khi sử dụng giải thuật, cần chú ý cung cấp đúng kiểu của tham số khuôn hình. Sử dụng thành thạo giải thuật trong STL sẽ tiết kiệm được rất nhiều công sức cho người lập trình, giúp người lập trình tập trung vào công việc chính cần giải quyết. Tự xây dựng các giải thuật là một phong cách lập trình tốt để tiết kiệm mã nguồn, tiết kiệm thời gian, công sức.

5.3.2. Các lỗi hay gặp khi lập trình

Lỗi sai kiểu: bộ duyệt trong nguyên mẫu là Bidirect iterator nhưng chỉ cung cấp Forward iterator. Đối tượng hàm trong nguyên mẫu yêu cầu Binary function nhưng lại cung cấp đối tượng hàm thuộc KHÁI NIỆM Unary function.

Lỗi thiếu các yêu cầu cho kiểu: Khi yêu cầu giá trị T là kiểu Equality Comparable, T phải nạp chồng toán tử so sánh `operator==`. Lỗi này hay xảy ra khi người dùng sử dụng kiểu tự định nghĩa và do đó, có thể định nghĩa thiếu một hoặc vài toán tử mà kiểu yêu cầu.

Lỗi vi phạm các điều kiện của biến: Các điều kiện của biến như khoảng hợp lệ, khoảng chồng nhau phải được tuân thủ. Ví dụ yêu cầu `result` không được thuộc trong khoảng `[first,last)` nhưng khi dùng lại không thỏa mãn, sẽ gây ra thông báo lỗi hoặc kết quả sai.

5.4. Bài tập

1/ Xem lại ví dụ trong mục 2.2.7. Nếu muốn việc sao chép bằng `replace_copy_if` thực hiện ngay trên chính `a`, ta có thể thay bộ duyệt `oi` bằng con trỏ `a` như sau:

```
replace_copy_if(a,a+6,a,bind2nd(less<int> (),5),5);

cout << "b: ";
replace_copy(a,a+6,oi,5,6);
cout << endl;
```

```
a: 1 2 3 4 5 6
b: 6 6 6 6 6 6
```

Hỏi làm như trên là đúng hay sai?

2/ Dùng `negate` và `sort` để sắp xếp một dãy theo thứ tự giảm dần.

Chương 6

LẬP TRÌNH KHÁI LƯỢC

Mục đích chương này

- Giới thiệu tiếp cận khái lược
- Ý tưởng lập trình khái lược trong STL

Trong quy trình phát triển phần mềm ứng dụng, người ta thường nhắc đến 2 tiếp cận căn bản: top-down và bottom-up. Với tiếp cận top-down, hệ thống được xây dựng từ tổng thể đến chi tiết. Từ một yêu cầu tổng thể, hệ thống được các nhà phát triển phân rã thành các hệ thống con, các hệ thống con này lại tiếp tục được phân rã thành các mô đun nhỏ hơn cho tới khi các mô đun này có thể dễ dàng mô hình hóa và phát triển. Có thể nói tiếp cận top-down khá phù hợp với thực tế phát triển phần mềm ứng dụng bởi nó xuất phát từ các vấn đề đặt ra, phân tích và thiết kế chúng theo cấu trúc nghiệp vụ, từ đó xây dựng giải pháp tương ứng có tính chặt chẽ cao. Tuy nhiên đứng về góc độ công nghệ phần mềm, cách tiếp cận như vậy khó đảm bảo tính sử dụng lại của các mô đun phần mềm. Do các mô đun này được xây dựng từ quá trình phân rã chức năng nghiệp vụ nên khả năng sử dụng lại chúng trong các ứng dụng khác (thậm chí trong các phiên bản kế tiếp của hệ thống hiện tại) rất thấp. Hạn chế này dĩ nhiên sẽ kéo theo chi phí phát triển tốn kém của các nhà phát triển phần mềm chuyên nghiệp.

Để nâng cao tính sử dụng lại, các mô đun phần mềm cần phải được xây dựng sao cho chúng phụ thuộc tối thiểu với nhau và với các bài toán nghiệp vụ đặt ra. Trong tiếp cận bottom-up, các mô đun phần mềm được xây dựng trước (thường dưới dạng các thư viện), sản phẩm cuối cùng sẽ được hình thành bằng cách lắp ghép các mô đun này. Có thể thấy rõ tư tưởng của tiếp cận này trong phương pháp phát triển phần mềm hướng thành phần (component-based) được đề cập nhiều trong thời gian gần đây. Việc đảo ngược thứ tự phát triển (từ những mô đun cụ thể lắp ghép thành hệ thống hoàn chỉnh) tạo nên sự phân chia trong các nhà phát triển phần mềm. Một nhóm đi theo con đường phát triển phần mềm ứng dụng bằng cách lắp ghép các mô đun có sẵn theo một kiến trúc phù hợp yêu cầu nghiệp vụ đặt ra, trong

khi nhóm còn lại tập trung nghiên cứu, phát triển các mô đun mang tính tổng quát, có thể áp dụng cho nhiều ứng dụng khác nhau.

Nếu như nhiệm vụ đặt ra cho nhóm thứ nhất là tương đối rõ ràng xuất phát từ các yêu cầu nghiệp vụ đặt ra thì với nhóm thứ hai công việc đòi hỏi người thiết kế và lập trình phải được trang bị các phương pháp luận để có thể mô hình hóa và phát triển các mô đun tổng quát, có tính sử dụng lại cao. Trong công nghệ phần mềm, thiết kế khái lược (generic design) và lập trình khái lược (generic programming) là thuật ngữ đề cập đến các phương pháp luận như vậy.

Thiết kế khái lược tập trung nghiên cứu, phát triển các mô hình thiết kế có thể được sử dụng phổ biến trong các ứng dụng. Tiêu biểu cho thiết kế khái lược là hai kỹ thuật: mẫu thiết kế (design patterns) và mẫu quy trình (framework). Mẫu thiết kế cung cấp những khuôn mẫu (patterns) theo tiếp cận hướng đối tượng giúp các nhà thiết kế dễ dàng tổ chức các lớp trong một ứng dụng một cách hiệu quả và dễ dàng sử dụng lại. Trong cuốn sách nổi tiếng "Design Patterns", các tác giả Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides đã thống kê 23 mẫu thiết kế cơ bản được chia làm 3 nhóm :

Nhóm	Mẫu	Mục đích
Creational	Abstract Factory	Xây dựng các đối tượng thuộc cùng nhóm nào đó
	Builder	Xây dựng các đối tượng cấu thành từ các đối tượng khác
	Factory Method	Giao diện tạo đối tượng
	Prototype	Tạo ra đối tượng bằng đối tượng mẫu
	Singleton	Đóng gói các thủ tục trong một đối tượng
Structural	Adaptor	Giao diện cho một đối tượng
	Bridge	Tách biệt giữa giao diện và thể hiện của đối tượng
	Composite	Đối tượng hợp thành
	Decorator	Bổ sung chức năng cho đối tượng mà không cần sử dụng kế thừa
	Façade	Giao diện với một hệ thống
	Flyweight	Lưu trữ hiệu quả tập đối tượng
	Proxy	Cầu nối để truy nhập tới một đối tượng

Behavioral	Chain of responsibility	Đối tượng lưu trữ các yêu cầu chức năng
	Command	Đối tượng thực hiện các yêu cầu chức năng
	Interpretor	Kiểm tra chính tả và dịch ngôn ngữ
	Iterator	Duyệt trên danh sách đối tượng
	Mediator	Cơ chế tương tác chung giữa các đối tượng
	Memento	Lưu trữ thông tin bên ngoài đối tượng
	Observer	Quan sát sự phụ thuộc giữa các đối tượng và thực hiện cập nhật các đối tượng cần thay đổi
	State	Trạng thái của một đối tượng
	Strategy	Giải thuật
	Template Method	Các bước của giải thuật
	Visitor	Bổ sung các chức năng cho đối tượng mà không cần thay đổi lớp tương ứng

Các mẫu thiết kế không giống với các thư viện. Có thể hình dung về mẫu thiết kế như một siêu mô hình (meta model). Trong từng ứng dụng cụ thể, các nhà phân tích thiết kế sẽ sử dụng siêu mô hình này để tạo ra những mô hình lớp cụ thể cho ứng dụng của mình. Tính khái lược trong mẫu thiết kế thể hiện ở chỗ chúng không phụ thuộc bất kỳ ứng dụng nào, bất kỳ công cụ phát triển nào. Các nhà thiết kế sử dụng chúng để rút ngắn quá trình thiết kế mà vẫn đạt được hiệu quả công việc.

Nếu như các mẫu thiết kế mang lại ích lợi không nhỏ trong việc thiết kế các lớp cụ thể của một ứng dụng thì mẫu quy trình (framework) lại tập trung vào việc vạch ra các quy trình tổng quát cho việc thiết kế những khối chức năng của ứng dụng. Ví dụ, để xây dựng cơ chế quản lý người sử dụng cho một ứng dụng, các nhà thiết kế mẫu quy trình đề xuất ra các mô hình lưu trữ thông tin người sử dụng, cơ chế phân quyền cho người sử dụng, mô hình thực hiện các yêu cầu bổ sung, loại bỏ hoặc cập nhật, phân quyền cho từng người sử dụng. Cũng giống như mục tiêu của mẫu thiết kế, mẫu quy trình quan tâm tới tính khái lược trong thiết kế, chỉ có điều chúng không áp dụng cho việc thiết kế những lớp cụ thể mà áp dụng cho các khối chức năng cụ thể.

Thiết kế khái lược chỉ dừng lại ở mức độ thiết kế. Cả mẫu thiết kế và mẫu quy trình chỉ tạo ra các khuôn mẫu tổng quát trợ giúp các nhà thiết kế mô hình hóa ứng dụng một cách nhanh chóng và hiệu quả chứ chưa đề cập tới việc xây dựng mã nguồn như thế nào. Lập trình khái lược cũng tiếp thu tư tưởng khái lược nhưng không phải áp dụng cho giai đoạn thiết kế mà cho giai đoạn lập trình. Trên cùng một bản thiết kế, các lập trình viên luôn mong muốn viết được những đoạn mã nguồn vừa ngắn gọn lại vừa đảm bảo hiệu quả sử

dụng và dễ dàng sử dụng lại. Việc tận dụng các thư viện phần mềm có sẵn là con đường ngắn nhất để đạt được kết quả như vậy. Một số nhà phát triển lựa chọn việc phát triển các thư viện phần mềm mang tính chuyên dụng, áp dụng vào những lĩnh vực cụ thể, ví dụ như thư viện đại số tuyến tính, thư viện đồ họa,... Ngược lại một số nhà nhà phát triển khác theo phong cách lập trình khái lược thì lại xây dựng những thư viện mang tính tổng quát, áp dụng cho mọi ứng dụng, mọi lĩnh vực. Không thể nói dạng thư viện nào cần thiết hơn, tuy nhiên do thư viện chuyên dụng thường bao hàm các cấu trúc dữ liệu và các giải thuật khá cụ thể do vậy ứng dụng sử dụng các thư viện này đòi hỏi phải tuân thủ một cách khá ngặt nghèo và cứng nhắc khi sử dụng các cấu trúc dữ liệu và thư viện này. Trong khi đó, các thư viện phát triển với phong cách lập trình khái lược lại quan tâm tới việc xây dựng các cấu trúc dữ liệu và giải thuật sao cho chúng có thể sử dụng một cách linh hoạt nhất trong ứng dụng.

Có thể tóm tắt một số nguyên tắc chính của lập trình khái lược như sau:

- Xây dựng thư viện các cấu trúc dữ liệu và giải thuật tổng quát mang tính sử dụng lại cao trong các ứng dụng.
- Đảm bảo tính thích nghi của chúng trong mọi ứng dụng bằng cách tách rời chúng khỏi các yếu tố ngôn ngữ cụ thể như kiểu dữ liệu.
- Đảm bảo tính hiệu quả của chúng khi được cụ thể hóa trong từng ứng dụng.
- Đảm bảo tính độc lập của chúng với các công cụ phát triển.

Trong phần tiếp theo chúng tôi xin minh họa ý tưởng của lập trình khái lược được thể hiện trong bộ thư viện khuôn hình chuẩn STL.

6.1. Lập trình khái lược trong STL

Có thể nói, STL là một thể nghiệm khá hoàn hảo của lập trình khái lược. Tính khái lược của STL được thể hiện ở các khía cạnh :

6.1.1. Thư viện của các cấu trúc dữ liệu và giải thuật tổng quát

Như đã đề cập tới trong các chương trước, STL bao gồm 5 thành phần chính : Bộ chứa, Bộ duyệt, Giải thuật, Đối tượng hàm và Bộ thích nghi.

Có thể nói các cấu trúc hợp thành chiếm đa số trong các cấu trúc dữ liệu cơ bản. Với bộ chứa, STL cung cấp cho người sử dụng những cấu trúc dữ liệu hợp thành rất phổ biến như tập hợp, mảng, danh sách móc nối, ngăn xếp, hàng đợi, bảng tra (map),... Người sử dụng có thể dễ dàng sử dụng các cấu trúc dữ liệu này thông qua những khuôn hình STL cung cấp. Điều đáng nói ở đây là các cấu trúc dữ liệu này được thiết kế thống nhất. Người sử dụng có thể sử

dụng cùng một tên hàm thành phần trong nhiều cấu trúc dữ liệu khác nhau, miễn là cấu trúc đó có hỗ trợ.

Ví dụ : hàm thành phần `push_back()` có thể được sử dụng trong đa số các bộ chứa của STL.

Một cấu trúc khá đặc sắc khác của STL là bộ duyệt. Nếu như trong C, C++ con trỏ là một khái niệm quan trọng trong mọi thao tác dữ liệu thì với bộ duyệt, STL đã thay thế được phần nào khái niệm con trỏ này. Bộ duyệt khắc phục được một số hạn chế của con trỏ như :

- Tránh cho người sử dụng can thiệp trực tiếp tới bộ nhớ vật lý.
- Bổ sung các phương thức thao tác (đóng gói con trỏ thành đối tượng).

Ngoài ra như đề cập tới ở chương III, bộ duyệt là bộ phận trung gian để tách rời giữa tổ chức lưu trữ dữ liệu và cơ chế xử lý dữ liệu. Hầu hết các giải thuật trong STL đều thao tác với bộ duyệt thay vì thao tác trực tiếp với cấu trúc dữ liệu (bộ chứa). Nhờ khái niệm bộ chứa này mà lập trình viên có thể dễ dàng sử dụng (hoặc tự xây dựng) các giải thuật trên cấu trúc dữ liệu mà không phải thay đổi nội dung của chúng.

Đối tượng hàm cũng là một trong những thế mạnh của STL. Đối tượng hàm đóng gói các cơ chế xử lý dữ liệu thành các đối tượng, nhờ vậy những thao tác được linh động hơn. Trong một giải thuật, thay vì sử dụng lời gọi tới một hàm cụ thể, người sử dụng có thể sử dụng đối tượng hàm đại diện. Nhờ vậy giải thuật có thể dễ dàng thay đổi để áp dụng cho những tình huống xử lý khác nhau.

Ví dụ : `Hàm sort()` có thể được sử dụng cho việc sắp xếp một danh sách với nhiều tiêu chí sắp xếp khác nhau bằng cách áp dụng với các đối tượng hàm tương ứng.

Một thành phần khác không kém phần quan trọng trong STL là các bộ thích nghi. Một mặt, người sử dụng quan tâm tới tính đa dạng của thư viện, mặt khác, họ lại quan tâm tới tính thống nhất của chúng. Chẳng hạn với một tập đối tượng, lập trình viên có thể sử dụng cấu trúc mảng hay cấu trúc danh sách móc nối để quản lý (tính đa dạng), tuy nhiên khi thao tác với chúng, họ lại muốn coi chúng như những ngăn xếp hay hàng đợi (tính thống nhất). Bộ thích nghi có thể xem như một giao diện thống nhất giữa những yêu cầu mà người sử dụng cần với những cấu trúc mà thư viện hỗ trợ.

Cuối cùng, không thể không nhắc đến các giải thuật mang tính khái lược cao của STL. STL bao hàm những giải thuật cực kỳ đơn giản nhưng lại được sử dụng thường xuyên trong mọi ứng dụng. Số lượng các giải thuật

trong STL cũng không nhiều, tuy nhiên lại có độ linh hoạt cao. Điều này giúp cho người sử dụng không cần phải ghi nhớ quá nhiều cú pháp mà vẫn thực hiện được công việc mong muốn.

6.1.2. Độc lập với kiểu dữ liệu

STL không phải là một thư viện đối tượng mà là một thư viện khuôn hình. Thư viện này không phục vụ một lớp kiểu dữ liệu đặc thù như thường thấy ở các bộ thư viện khác mà nó hoàn toàn xây dựng trên những kiểu dữ liệu trừu tượng. Chính vì vậy cùng một đoạn mã nguồn, STL có thể áp dụng cho một lớp kiểu dữ liệu khác nhau. Một bộ chứa `vector` có thể được sử dụng trong ứng dụng cho các `vector` số nguyên, số thực hay thậm chí `vector` các đối tượng người sử dụng tự định nghĩa. Tương tự như vậy, cùng một giải thuật `sort` có thể sử dụng để sắp xếp trên các danh sách số nguyên, số thực,... chỉ cần chúng được sắp xếp tuần tự.

Có thể thấy tất cả các thành phần của STL đều áp dụng nguyên tắc này. Các bộ chứa sử dụng những kiểu dữ liệu trừu tượng cho phép lập trình việc xây dựng các cấu trúc dữ liệu khác nhau với cùng một khuôn hình. Các bộ duyệt cho phép thống nhất khái niệm con trỏ duyệt cho nhiều bộ chứa khác nhau, Khuôn hình giải thuật hoạt động với các bộ duyệt mà không cần quan tâm tới việc chúng tham chiếu tới đối tượng kiểu gì. Lớp đối tượng hàm cũng được gắn với các tham số khuôn hình do đó có thể tạo ra lớp những đối tượng hàm cụ thể. Bộ thích nghi là tầng giao diện giữa yêu cầu người sử dụng với các khuôn hình trong STL do vậy chúng cũng được tham số hóa bởi những kiểu dữ liệu là tham số khuôn hình.

Tính độc lập với kiểu dữ liệu giúp cho STL có một khả năng linh hoạt cao trong ứng dụng. Với cùng giao diện thống nhất, lập trình viên có thể tạo ra những cấu trúc dữ liệu mới, áp dụng những giải thuật trên đó mà không cần viết thêm (hoặc viết thêm rất ít) mã nguồn.

6.1.3. Tính hiệu quả

Trên một ngôn ngữ lập trình, việc tổng quát hóa thường kéo theo tính kém hiệu quả trong thực tế áp dụng. STL cũng không phải một ngoại lệ, tuy nhiên tác giả STL đã cố gắng hạn chế tối đa của sự kém hiệu quả này bằng cách phân loại các khuôn hình và chỉ sử dụng những xử lý hiệu quả trên các khuôn hình thích hợp. Ví dụ như thành phần hàm `push_front()` chỉ có mặt trong khuôn hình `list` (danh sách móc nối) chứ không có mặt trong khuôn hình `vector` (mảng). So với các thư viện khác, chẳng hạn như MFC, một vài

cấu trúc của STL tỏ ra không thực sự hiệu quả , tuy nhiên trên các thư viện đó thường không có được sự gọn gàng và thống nhất như với STL.

Mặt khác, với sự phát triển của C++, hạn chế này hoàn toàn có thể giải quyết được bằng những phương pháp cụ thể hóa khuôn hình trên các dữ liệu đặc thù để tạo nên những cấu trúc dữ liệu và giải thuật hiệu quả.

6.1.4. Tính độc lập với môi trường phát triển

Có lẽ không chỉ có STL mới có tính chất này, nhiều thư viện khác cũng được xây dựng với mục đích áp dụng cho các môi trường phát triển khác nhau. Tuy nhiên lợi thế của STL so với các thư viện khác ở chỗ nó đã trở thành một chuẩn thư viện của C++ do vậy hầu hết các công cụ phát triển C++ đều có hỗ trợ STL. Đây là một đặc điểm quan trọng cho các nhà phát triển bởi với STL họ sẽ không phải quan tâm tới việc xây dựng lại mã nguồn khi thay đổi môi trường phát triển.

6.2. Mở rộng lập trình khái lược với nền tảng STL

Vận dụng ý tưởng lập trình khái lược của STL, đồng thời cũng sử dụng chính bộ thư viện này, một số trung tâm nghiên cứu đã đưa ra các bộ thư viện khuôn hình khác phục vụ các lớp ứng dụng đặc thù hơn.

Trong phụ lục của cuốn sách này, chúng tôi xin giới thiệu tới bạn đọc hai bộ thư viện như vậy :

- Thư viện khuôn hình cho ma trận MTL (trường đại học Notre Dame – Pháp).
- Thư viện khuôn hình cho đồ thị GTL (trường đại học Passau – Đức).

Phụ lục A

CÁC KHÁI NIỆM TRONG STL

Có hai tiêu chí để phân loại các thành phần của thư viện STL. Theo cách thứ nhất, các thành phần được xếp vào 7 loại:

1. Bộ chứa
2. Bộ duyệt
3. Các thuật toán
4. Đối tượng hàm
5. Tiện ích
6. Bộ thích nghi
7. Bộ cấp phát

Theo cách thứ hai, các thành phần được phân vào 3 loại

1. Cấu trúc dữ liệu (các cấu trúc và các lớp)
2. Hàm
3. KHÁI NIỆM

Hai cách phân loại trên hoàn toàn độc lập. Mỗi cách đều có thể áp dụng cho mọi thành phần trong STL. Ví dụ, nếu xét theo cách phân loại thứ nhất, vector thuộc phạm trù bộ chứa. Nếu xét theo cách phân loại thứ hai, vector thuộc lớp các cấu trúc dữ liệu.

Trong cuốn sách này, chúng tôi trình bày về thư viện STL dựa theo cách phân loại thứ hai. Chúng tôi chọn hướng tiếp cận nhằm giới thiệu tới đông đảo bạn đọc một thư viện lập trình hữu ích mà không đòi hỏi ở người đọc những kiến thức quá sâu về lập trình.

Trong các chương 2,3 và 4, chúng tôi đã trình bày các cấu trúc dữ liệu cơ bản của STL. Các cấu trúc này bao gồm các lớp bộ chứa, các bộ duyệt, các đối tượng hàm và các bộ thích nghi. Các hàm trong thư viện STL là các giải thuật đã được trình bày trong chương 5. Tuy nhiên, với mong muốn trình bày thư viện STL dễ hiểu nhất, chúng ta chỉ xem xét các KHÁI NIỆM trong phần Phụ lục.

A.1. Sơ lược về KHÁI NIỆM

Hầu hết các bộ chứa, các giải thuật và các đối tượng hàm của STL được cung cấp dưới dạng các khuôn hình. Do vậy, khi sử dụng các lớp này cần xem xét tới kiểu dữ liệu nào có thể thay thế cho tham số khuôn hình. Một kiểu dữ liệu có thể thay thế cho tham số khuôn hình phải thoả mãn một số điều kiện nhất định. Các điều kiện được gọi là ràng buộc và phụ thuộc vào từng lớp cụ thể. Tập các ràng buộc có thể có được chia vào các nhóm. Mỗi nhóm các ràng buộc được gọi là KHÁI NIỆM.

Các KHÁI NIỆM được xây dựng theo nguyên tắc tinh chỉnh dần. Nghĩa là, các KHÁI NIỆM phức tạp được xây dựng bằng cách kết hợp các KHÁI NIỆM đơn giản hơn với nhau. Việc tinh chỉnh cũng có thể là bổ sung thêm hoặc sửa đổi các ràng buộc từ các KHÁI NIỆM khác. Để tạo ra một KHÁI NIỆM mới, người ta có thể kết hợp cả hai cơ chế trên. Với cách thức này, một KHÁI NIỆM có thể là kết quả từ việc kết hợp có sửa đổi và bổ sung từ một hoặc nhiều KHÁI NIỆM tiền đề. Khi đó, ta nói ngắn gọn KHÁI NIỆM được tinh chỉnh từ các KHÁI NIỆM tiền đề. Lưu ý rằng, việc tinh chỉnh không phải lúc nào cũng tạo ra một KHÁI NIỆM mới phức tạp hơn.

KHÁI NIỆM không được định nghĩa như một phần của ngôn ngữ lập trình nói chung và ngôn ngữ C++ nói riêng. Không có cách nào để định nghĩa một KHÁI NIỆM trong chương trình hay thể hiện một cấu trúc cụ thể nào đó là mô hình của một KHÁI NIỆM. Tuy vậy, KHÁI NIỆM là một phần đặc biệt quan trọng của thư viện STL. Sử dụng các KHÁI NIỆM tạo ra sự thuận tiện trong việc xây dựng cũng như sử dụng thư viện. Đối với người xây dựng thư viện, sử dụng KHÁI NIỆM tạo ra sự độc lập tương đối giữa đề xuất giao diện và xây dựng phần thực thi của các hàm.

A.2. Các KHÁI NIỆM trong STL

Để tiện trình bày chúng tôi tạm chia các KHÁI NIỆM trong thư viện STL vào 4 lớp sau:

1. Các KHÁI NIỆM cơ sở, bao gồm: Assignable, Default Constructible, Equality Comparable và Less Than Comparable.
2. Các KHÁI NIỆM liên quan tới bộ chứa, bao gồm: Container, Forward Container, Sequence, Associative Container, Reversible Container, Back Insert Sequence, Front Insert Sequence, Simple Associative Container, Pair Associative Container, Unique Associative Container, Multiple Associative Container, Sorted

Associative Container, Hash Associative Container, Unique Sorted Associative Container, Random Access Container.

3. Các KHÁI NIỆM liên quan tới bộ duyệt, bao gồm: Output Iterator, Trivial Iterator, Input Iterator, Forward Iterator, Bidirection Iterator và Random Access Iterator.
4. Các KHÁI NIỆM liên quan tới đối tượng hàm, bao gồm Generator, Unary Function, BinaryFuntion, Adapable Generator, Adapable Unary Function, Adapable Binary Function, Predicate, Hash Function, Binary Predicate, Adapable Predicate, AdapbleBinary Predicate, Strict Weak Order, Monoid Operation và Random Number Generator

A.2.1. Các KHÁI NIỆM cơ sở

KHÁI NIỆM Assignable

Một kiểu dữ liệu được gọi Assignable nếu có thể sao chép các đối tượng và gán giá trị cho biến thuộc kiểu này.

KHÁI NIỆM Default Constructible.

Một kiểu dữ liệu được gọi là Default Constructible nếu nó có cấu tử mặc định. Điều đó có nghĩa là có thể tạo ra một đối tượng thuộc kiểu này mà không cần khởi tạo đối tượng với bất cứ một giá trị nào.

KHÁI NIỆM Equality Comparable

Một kiểu dữ liệu được gọi là Equality Comparable nếu hai đối tượng thuộc kiểu này có thể so sánh sự ngang bằng thông qua toán tử == đồng thời quan hệ ngang bằng được định nghĩa bởi toán tử == là một quan hệ tương đương.

KHÁI NIỆM LessThan Comparable

Một kiểu dữ liệu được gọi là Less Than Comparable nếu hai đối tượng thuộc kiểu này có thể xác định trật tự bằng toán tử < đồng thời quan hệ được định nghĩa qua toán tử < là quan hệ bất đối xứng.

A.2.2. Các KHÁI NIỆM liên quan tới bộ chứa

KHÁI NIỆM Container

Một kiểu dữ liệu được gọi là Container nếu nó cho phép lưu trữ một tập các đối tượng thuộc kiểu khác. Các đối tượng này được gọi là phần tử của Container

KHÁI NIỆM Container được tinh chỉnh từ KHÁI NIỆM Assignable

KHÁI NIỆM Forward Container

Một kiểu dữ liệu thoả mãn KHÁI NIỆM Forward Container nếu nó là một Container đồng thời các phần tử của nó được sắp xếp theo một trật tự nhất định.

KHÁI NIỆM Forward Container được tinh chỉnh từ các KHÁI NIỆM Container, Equality Comparable và LessThanComparble.

KHÁI NIỆM Reversible Container

Một kiểu dữ liệu được gọi là Reversible Container nếu nó thoả mãn các điều kiện của KHÁI NIỆM Forward Container đồng thời bộ duyệt của nó là một Bidirection Iterator. Các bộ chứa dạng này cho phép duyệt bộ chứa theo thứ tự từ cuối lên đầu.

KHÁI NIỆM Reversible được tinh chỉnh từ KHÁI NIỆM Forward Container.

KHÁI NIỆM Random Access Container

Một kiểu dữ liệu được gọi là Random Access Container nếu nó là Reversible Container đồng thời bộ duyệt của nó là một Random Access Iterator. Các bộ chứa dạng này cho phép thời gian truy nhập một phần tử luôn là một hằng số.

KHÁI NIỆM Random Access Container được tinh chỉnh từ KHÁI NIỆM Reversible Container.

KHÁI NIỆM Sequence

Một Sequence là một Container với kích thước biến đổi. Các phần tử trong một Sequence được tổ chức theo thứ tự tuyến tính. Sequence hỗ trợ các thao tác bổ sung và loại bỏ các phần tử.

KHÁI NIỆM Sequence được tinh chỉnh từ các các KHÁI NIỆM Container và Default Constructible.

KHÁI NIỆM Front Insert Sequence

Một kiểu dữ liệu là Front Insert Sequence nếu nó là một Sequence đồng thời nó hỗ trợ các cơ chế cho phép thời gian để bổ sung hay truy xuất một phần tử ở đầu dãy luôn là hằng số.

KHÁI NIỆM Front Insert Sequence được tinh chỉnh từ KHÁI NIỆM Sequence.

KHÁI NIỆM Back Insert Sequence

Back Insert Sequence là một Sequence cho phép bổ sung hay truy xuất phần tử ở cuối với thời gian hằng số.

KHÁI NIỆM Back Insert Sequence được tinh chỉnh từ KHÁI NIỆM Sequence.

KHÁI NIỆM Associative Container

Associative Container là một Container có kích thước khả biến cho phép truy xuất các phần tử dựa trên giá trị khoá. Việc lưu trữ các phần tử được tổ chức dựa trên giá trị khoá. Đây là một đặc trưng quan trọng của Associative Container. Đặc trưng này dẫn tới một số tính chất của Associative Container.

- Thứ nhất, việc bổ sung các phần tử không cần chỉ định vị trí bổ sung.
- Thứ hai, mỗi phần tử trong một Associative Container liên quan tới hai giá trị: giá trị phần dữ liệu và giá trị khoá. Giá trị khoá của một phần tử không được phép thay đổi.

KHÁI NIỆM Associative Container được tinh chỉnh từ các KHÁI NIỆM Container và Default Constructible.

KHÁI NIỆM SimpleAssociativeContainer

Simple Associative Container là một Associative Container có giá trị phần dữ liệu và giá trị khoá là một. Do vậy giá trị của khoá cũng là giá trị phần dữ liệu và cũng là giá trị của phần tử.

KHÁI NIỆM Simple Associative Container được tinh chỉnh từ KHÁI NIỆM Associative Container.

KHÁI NIỆM PairAssociativeContainer

Pair Associative Container là một Associative Container với mỗi phần tử một cặp bao gồm giá trị khoá và giá trị phần tử. Hai giá trị này hoàn toàn độc lập với nhau. Như vậy, giá trị của một phần tử là một cặp giá trị.

KHÁI NIỆM Pair Associative Container được tinh chỉnh từ KHÁI NIỆM Associative Container.

KHÁI NIỆM Sorted Associative Container

Sorted Associative Container là một Associative Container. Các phần tử trong Sorted Associative Container được sắp xếp theo một trật tự nhất định. Trật tự này được xác định theo một quan hệ giữa các giá trị khoá.

Việc sắp xếp các phần tử trong Sorted Associative Container giúp cho độ phức tạp của hầu hết của thao tác trên bộ chứa không vượt quá $\log_2 n$.

KHÁI NIỆM Sorted Associative Container được tinh chỉnh dựa trên hai KHÁI NIỆM Reversible Container và Associative Container.

KHÁI NIỆM Hash Associative Container

Hash Associative Container là một Associative Container được tổ chức lưu trữ giống một bảng băm. Các phần tử trong Hash Associative Container không có một trật tự nhất định nào và thực tế là chúng không được sắp xếp. Độ phức tạp của các thao tác trên Hash Associative Container trong trường hợp xấu nhất là O(n) còn trung bình là O(C) với C là một hằng số. Điều này có nghĩa là đối với những ứng dụng chỉ đơn thuần lưu trữ và truy xuất các giá trị, thứ tự là không quan trọng, sử dụng Hash Associative Container hiệu quả hơn Sorted Associative Container.

KHÁI NIỆM Hash Associative Container được tinh chỉnh từ KHÁI NIỆM Associative Container.

KHÁI NIỆM Unique Associative Container

Một Unique Associative Container là một Associative Container thoả mãn yêu cầu không chứa hai phần tử có cùng giá trị khoá.

KHÁI NIỆM Unique Associative Container được tinh chỉnh từ KHÁI NIỆM Associative Container.

KHÁI NIỆM Multiple Associative Container

Một Multiple Associative Container là một Associative Container cho phép nhiều hơn một phần tử có cùng giá trị khoá.

KHÁI NIỆM Multiple Associative Container được tinh chỉnh từ KHÁI NIỆM Associative Container.

KHÁI NIỆM Unique Sorted Associative Container

Unique Sorted Associative Container là một Sorted Associative Container thoả mãn ràng buộc về tính duy nhất của giá trị khoá trong tập các phần tử do Unique Associative Container quy định.

KHÁI NIỆM Unique Sorted Associative Container được tinh chỉnh từ các KHÁI NIỆM Sorted Associative Container và Unique Associative Container.

A.2.3. Các KHÁI NIỆM liên quan tới bộ duyệt

KHÁI NIỆM TrivialIterator

Một đối tượng được gọi là Trivial Iterator nếu nó được sử dụng để trỏ tới một đối tượng khác. Việc giải tham chiếu trên đối tượng Trivial Iterator cho phép tham chiếu tới đối tượng được nó trỏ tới.

KHÁI NIỆM Trivial Iterator được tinh chỉnh từ các KHÁI NIỆM Assignable, Equality Comparable và Default Constructible.

KHÁI NIỆM Input Iterator

Input Iterator là một Trivial Iterator trỏ tới một phần tử trong một tập các phần tử. Ngoài việc hỗ trợ phép giải tham chiếu để tham chiếu tới đối tượng đang trỏ tới, Input Iterator còn có liên hệ với phần tử kế tiếp của phần tử đang được trỏ tới trong tập các phần tử. Liên hệ này cho phép Input Iterator có thể trỏ tới phần tử kế tiếp này bằng cách tự tăng lên một đơn vị.

KHÁI NIỆM Input Iterator được tinh chỉnh từ KHÁI NIỆM TrivialIterator.Trivial Iterator.

KHÁI NIỆM Output Iterator

Một đối tượng Output Iterator sẽ phải hỗ trợ các cơ chế để lưu lại một giá trị vào một tập phần tử tai vị trí nó đang trỏ tới. Output Iterator không cho phép tham chiếu tới một đối tượng nào.

KHÁI NIỆM Output Iterator được tinh chỉnh từ các KHÁI NIỆM Assignable và Default Constructible.

KHÁI NIỆM Forward Iterator

Một đối tượng Forward Iterator cho phép truy xuất các phần tử trong một tập các phần tử đồng thời cho phép lưu lại một giá trị vào sau vị trí nó đang trỏ tới. Forward Iterator chỉ cho phép duyệt dãy các giá trị theo một chiều nhất định.

KHÁI NIỆM Forward Iterator được tinh chỉnh từ các KHÁI NIỆM Input Iterator và OutputIterator.Output Iterator

KHÁI NIỆM Bidirection Iterator

Một đối tượng Bidirection Iterator cho phép truy xuất các phần tử trong một tập các phần tử đồng thời cho phép lưu lại một giá trị vào sau vị trí nó đang trỏ tới. Bidirection Iterator cho phép duyệt dãy các giá trị theo cả hai chiều.

KHÁI NIỆM Bidirection Iterator được tinh chỉnh từ KHÁI NIỆM ForwardIterator.Forward Iterator.

KHÁI NIỆM Random Access Iterator

Một Random Access Iterator là một Bidirection Iterator cho phép dịch chuyển tới các phần tử đứng trước hoặc sau phần tử đang trỏ tới (không nhất thiết là liền kề) với thời gian là hằng số.

KHÁI NIỆM Random Access Iterator được tinh chỉnh từ các KHÁI NIỆM Bidirection Iterator và LessThanComparable.Less Than Comparable.

A.2.4. Các KHÁI NIỆM liên quan tới đối tượng hàm

KHÁI NIỆM Generator

Một đối tượng được gọi là một Generator nếu nó có toán tử gọi hàm không tham số.

KHÁI NIỆM Generator được tinh chỉnh từ KHÁI NIỆM Assignable.

KHÁI NIỆM Unary Function

Một đối tượng được gọi là một Unary Function nếu nó có toán tử gọi hàm một đối số.

KHÁI NIỆM Unary Function được tinh chỉnh từ KHÁI NIỆM Assignable.

KHÁI NIỆM Binary Function

Một đối tượng được gọi là một Binary Function nếu nó có toán tử gọi hàm hai đối số.

KHÁI NIỆM Binary Function được tinh chỉnh từ KHÁI NIỆM Assignable.

KHÁI NIỆM Adapable Generator

Một Adapable Generator là một Generator có định nghĩa `typedef` cho kiểu trả về trong phần định nghĩa.

KHÁI NIỆM Adapable Generator được tinh chỉnh từ KHÁI NIỆM Generator

KHÁI NIỆM Adapable Unary Function

Một Adapable Unary Function là một Unary Function có các định nghĩa `typedef` cho kiểu trả về và kiểu của đối số.

KHÁI NIỆM Adapable Unary Function được tinh chỉnh từ KHÁI NIỆM UnaryFunction.Unary Function

KHÁI NIỆM AdapableBinaryFunction

Một Adapable Binary Function là một Binary Function có các định nghĩa `typedef` cho kiểu trả về và kiểu của các đối số.

KHÁI NIỆM Adapable Binary Function được tinh chỉnh từ KHÁI NIỆM BinaryFunction.Binary Function

KHÁI NIỆM Predicate

Một Predicate là một Unary Function có kiểu của giá trị trả về là `bool`.

KHÁI NIỆM Predicate được tinh chỉnh từ KHÁI NIỆM UnaryFunction.Unary Function

KHÁI NIỆM BinaryPredicate

Một Binary Predicate là một BinaryFunction có kiểu của giá trị trả về là `bool`.

KHÁI NIỆM Binary Predicate được tinh chỉnh từ KHÁI NIỆM BinaryFunction.Binary Function

KHÁI NIỆM AdapablePredicate

Một Adapable Predicate là một Predicate có định nghĩa `typedef` cho kiểu đối số và giá trị trả về (`bool`)

KHÁI NIỆM Adapable Predicate được tinh chỉnh từ các KHÁI NIỆM Predicate và AdapableUnaryFunction.Adapable Unary Function

KHÁI NIỆM AdapableBinaryPredicate

Một Adapable Binary Predicate là một Binary Predicate có định nghĩa `typedef` cho kiểu đối số và giá trị trả về (`bool`).

KHÁI NIỆM Adapable Binary Predicate được tinh chỉnh từ các KHÁI NIỆM Binary Predicate và AdapableBinaryFunction.Adapable BinaryFunction.

KHÁI NIỆM StrictWeakOrdering

Strict Weak Ordering là một Binary Predicate so sánh hai đối tượng. StreakWeakOrdering sẽ trả về giá trị true nếu đối tượng tương ứng với tham số đầu đứng trước đối tượng tương ứng với tham số thứ hai .Vị từ này phải thoả mãn KHÁI NIỆM tương ứng trong toán học.

KHÁI NIỆM Strict Weak Ordering được tinh chỉnh từ KHÁI NIỆM BinaryPredicate.Binary Predicate

KHÁI NIỆM MonoidOperation

Monoid Operation là một BinaryFunction thoả mãn 3 điều kiện sau:

- Kiểu của các đối số và kết quả trả về phải là một.
- Tồn tại một phần tử trung hoà (identity element). Phần tử trung hoà của hàm hai đối số `f` trên tập giá trị `T` là một phần tử `id` thuộc `T` sao cho với mọi `x` thuộc `T` ta luôn có `f(x, id) = f(id, x) = x`.
- Có tính kết hợp, nghĩa là nếu `f` là một hàm hai đối số, `x`, `y` và `z` là ba đối tượng cùng kiểu thì `f(f(x, y), z) = f(x, f(y, z))`.

KHÁI NIỆM RandomNumberGenerator

Random Number Generator là một hàm một đối số dùng để sinh ra một số nguyên ngẫu nhiên. Số ngẫu nhiên được sinh ra nằm trong khoảng `[0, N]` với `N` là giá trị của đối số.

KHÁI NIỆM Random Number Generator được tinh chỉnh từ KHÁI NIỆM UnaryFunction.Unary Function

KHÁI NIỆM HashFunction

Hash Function là một Unary Function đóng vai trò ánh xạ giữa giá trị của đối số với giá trị trả về. Hash Function đảm bảo rằng giá trị trả về chỉ phụ thuộc duy nhất vào đối số. Đồng thời, nếu giá trị của đối số bằng nhau, giá trị trả về của hàm cũng bằng nhau.

KHÁI NIỆM Hash Function được tinh chỉnh từ KHÁI NIỆM UnaryFunction.Unary Function

Phụ lục B

MỘT SỐ THƯ VIỆN KHUÔN HÌNH KHÁC

STL đã trở thành một xu hướng, một phong cách lập trình mới. Hiện nay trên thế giới đã có rất nhiều bộ thư viện được xây dựng theo phong cách STL. Có thể kể ra ở đây một vài thư viện tiêu biểu:

- MTL (Matrix Template Library): bộ thư viện khuôn hình cho ma trận. Đây là bộ thư viện do trường Đại học Notre Dames phát triển. MTL cung cấp các dạng khác nhau của ma trận (thưa, đặc, đường chéo, tam giác,...) dưới dạng các lớp bộ chứa. MTL đồng thời cung cấp các lớp bộ duyệt, đối tượng hàm, giải thuật để làm việc trên ma trận.
- GTL (Graph Template Library): bộ thư viện khuôn hình cho đồ thị. Đây là bộ thư viện do trường Đại học Passau phát triển. GTL tập trung chủ yếu lên các giải thuật phục vụ cho đồ thị như các giải thuật BFS, DFS, các giải thuật sắp xếp, tìm đường truyền thống trên đồ thị.
- BTL (Boost Template Libraries): có thể nói đây là một tập hợp các thư viện khuôn hình do nhiều người phát triển được quản lý bởi tổ chức Boost. Mỗi thành viên của Boost có thể thêm vào các thư viện khuôn hình của chính mình hoặc sửa đổi các thư viện sẵn có một cách tùy ý. Hiện tại, BTL có khoảng 50 thư viện khuôn hình bao gồm rất nhiều thể loại, trong đó có cả thư viện cho đồ thị (BGL - Boost Graph Library).
- XTL (XML Template Library): bộ thư viện khuôn hình cho XML.
- MySQL++: bộ thư viện khuôn hình cho hệ quản trị cơ sở dữ liệu MySQL. MySQL là một trong những hệ quản trị cơ sở dữ liệu thường gặp trên nền Unix/Linux. MySQL++ là bộ thư viện cung cấp một giao diện lập trình cho MySQL. Phiên bản mới nhất của MySQL++ đã được viết lại theo phong cách STL.
- CGI++: bộ thư viện khuôn hình cho lập trình CGI (Common Gateway Interface). Cũng như My SQL++, CGI++ mới được viết lại theo phong cách STL. Điều này có thể minh chứng cho sự phát triển của xu hướng lập trình STL.

- DTL (Database Template Library): bộ thư viện khuôn hình cho lập trình với cơ sở dữ liệu. DTL có thể làm việc được với nhiều hệ quản trị cơ sở dữ liệu, trên hệ điều hành Windows cũng như trên hệ điều hành Unix/Linux.

Các bộ thư viện trên và còn nhiều bộ thư viện khuôn hình khác vẫn còn đang được phát triển, mở rộng. Chúng tôi xin giới thiệu dưới đây một cách chi tiết hơn về hai bộ thư viện tiêu biểu là MTL và GTL.

B.1. Thư viện MTL

Trong các bộ thư viện liệt kê ở trên, MTL là bộ thư viện được xây dựng giống với STL nhất. Từ cách xây dựng thư viện, các thành phần của thư viện cho đến tài liệu tham khảo đều giống với STL. Tuy phần tài liệu tham khảo của MTL chưa được đầy đủ như của STL và còn nhiều thiếu sót, nhưng nếu bạn đọc đã quen với STL thì việc làm quen với MTL sẽ không quá khó khăn.

B.1.1. Cài đặt MTL

Cũng như các bộ thư viện kiểu STL khác, MTL có thể được biên dịch trên nhiều hệ điều hành. Chúng tôi đã thử biên dịch MTL trên Linux, Unix với gcc, trên Windows với Visual C++ 6.0 và Visual C.NET là các hệ điều hành và trình biên dịch C++ phổ biến ở Việt Nam. Người dùng muốn sử dụng MTL trên hệ điều hành nào, lên trang web http://lsc.nd.edu/research/mtl để lấy về bộ thư viện tương ứng. Tại thời điểm chúng tôi hoàn thành cuốn sách (11/2002), bạn đọc có thể tham khảo như sau:

- Tệp mtl-2.1.2-21.tar.gz dành cho hệ điều hành Linux/Unix.
- Tệp mtl-2.1.2-20.zip dành cho hệ điều hành Windows, trình biên dịch VC6.
- Tệp mtl-2.1.2-21.zip dành cho hệ điều hành Windows, trình biên dịch VC7 (hay VC.NET).

Bạn đọc có thể lấy ba tệp này trong đĩa CD đi kèm. Sau khi giải nén, tìm tệp INSTALL hướng dẫn cách cài đặt. Việc cài đặt MTL trên Windows rất dễ dàng: chỉ cần đặt biến đường dẫn cho các tệp tiêu đề chỉ tới thư mục giải nén mtl-2.1.2-xx là xong. Sau đó, khi sử dụng ta thêm các dẫn hướng biên dịch như `#include <<mtl/matrix.h>`.

Chương trình sau tạo hai ma trận, thực hiện phép nhân ma trận trên chúng rồi in ra.

```
#include <mtl/matrix.h>
#include <mtl/utils.h>
```

```
#include <mtl/mtl.h>
using namespace mtl;
void main()
{
    typedef mtl::matrix <int>::type MATRIX;
    MATRIX m(4,4), n(4,4), p(4,4);
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
        {
            m(i,j) = j;
            n(i,j) = i;
        }
    print_all_matrix(m);
    print_all_matrix(n);
    mtl::mult(m,n,p);
    print_all_matrix(p);
}
```

```
4x4
[
[0,1,2,3],
[0,1,2,3],
[0,1,2,3],
[0,1,2,3]
]
4x4
[
[0,0,0,0],
[1,1,1,1],
[2,2,2,2],
[3,3,3,3]
]
4x4
[
[14,14,14,14],
[14,14,14,14],
[14,14,14,14],
[14,14,14,14]
]
```

B.1.2. Các thành phần của MTL

Các thành phần chính của MTL bao gồm bộ chứa, bộ duyệt, bộ sinh, bộ chọn, đối tượng hàm, giải thuật. Do khuôn khổ hạn hẹp của quyển sách, chúng tôi không thể trình bày chi tiết các thành phần đó mà chỉ giới thiệu sơ bộ. Bạn đọc nên tham khảo thêm trong tài liệu của MTL đi kèm trong đĩa CD.

Định dạng ma trận

Trước khi nói về các thành phần của MTL, ta cần nói đến định dạng của ma trận trong MTL. Định dạng của ma trận bao gồm bốn định dạng: kiểu phần tử, kiểu hình dạng, kiểu lưu trữ và kiểu hướng. Kiểu phần tử bao gồm `int`, `double`, `float`, `bool`,... Kiểu hình dạng bao gồm: `rectangle` (hình chữ nhật), `band` (hình dải), `diagonal` (đường chéo). Kiểu hình dạng liên quan tới việc phân bố vị trí các phần tử khác 0 trong ma trận. Ví dụ, ma trận tam giác (trường hợp đặc biệt của `band`):

```
[ 1  2  3  4  5 ]
[ 0  6  7  8  9 ]
[ 0  0  10 11 12 ]
[ 0  0  0  13 14 ]
[ 0  0  0  0  15 ]
```

Kiểu lưu trữ liên quan tới phân bố mật độ các số khác 0 trong ma trận như đặc, thưa, nén... Chúng bao gồm: `packed`, `banded`, `dense`, `array`, `compressed`. Kiểu hướng gồm `column__major` và `row__major`. Khi ma trận có kiểu hướng là `column__major`, bộ duyệt hai chiều trên ma trận sẽ duyệt trên các cột. Khi ma trận có kiểu hướng là `row__major`, bộ duyệt hai chiều trên ma trận sẽ duyệt trên các hàng. Bộ chứa, bộ duyệt, bộ chọn, bộ sinh,... các thành phần khác của ma trận đều liên quan chặt chẽ tới một trong các định dạng trên của ma trận. Một ma trận khi không thể thiếu được một trong bốn định dạng đó. Ví dụ sau định nghĩa một ma trận nguyên hình chữ nhật lưu trữ kiểu đặc và có hướng là cột:

```
typedef matrix<int,
               rectangle<>,
               dense<>,
               column_major>::type ColMatrix;
```

Bộ chứa và bộ duyệt

MTL xây dựng các bộ chứa ứng với các KHÁI NIỆM: Matrix, ColumnMatrix (ma trận cột), DiagonalMatrix (ma trận đường chéo), RowMatrix (ma trận hàng), TwoDStorage (ma trận lưu trữ hai chiều) và Vector. KHÁI NIỆM trung tâm trong MTL tất nhiên là KHÁI NIỆM Matrix. Matrix có thể coi như một bộ chứa của bộ chứa hay một bộ chứa hai chiều (2D container). Vì là bộ chứa, MTL có các hàm thành phần `begin()`, `end()` trả về các bộ duyệt hai chiều (2D iterator - duyệt từng hàng hoặc từng cột). Giải tham chiếu bộ duyệt hai chiều ta được bộ chứa một chiều (1D

container). Bộ chứa một chiều lại có các hàm `begin()`, `end()` trả về bộ duyệt một chiều (1D iterator - duyệt từng phần tử của hàng hay cột). Đoạn chương trình sau in ra ma trận `a` dùng bộ duyệt một chiều và bộ duyệt hai chiều:

```
typedef Matrix::const_iterator i; // bo duyet hai chieu
typedef Matrix::OneD::const_iterator j; // bo duyet mot chieu
for (i = A.begin(); i != A.end(); i++)
{
    for(j = (*i).begin(); j != (*i).end(); j++)
       cout << (*j) << " ";
    cout << endl;
}
```

Từ các KHÁI NIỆM trên, MTL cung cấp rất nhiều bộ chứa hữu dụng: `column_matrix`, `row_matrix`, `diagonal_matrix`, `triangle_matrix`, `symmetric_matrix`,... Liên quan tới bộ chứa còn có bộ sinh (generator), bộ chọn (selector), bộ thích nghi (adaptor),...

Bộ sinh quan trọng nhất là `matrix`. Khi sinh một ma trận, phải cung cấp đủ bốn định dạng như đã nói. Các bộ sinh còn lại gồm `band_view`, `block_view`, `symmetric_view`, `tri_view` và `triangle_view`. Các bộ sinh này tạo ra ma trận với định dạng mới từ ma trận đầy đủ. Ví dụ, có thể dùng `triangle_view` để tạo ra ma trận tam giác từ ma trận đầy đủ. Chương trình dưới đây minh họa cách dùng `band_view`:

```
#include <iostream>
#include "mtl/matrix.h"
#include "mtl/mtl.h"

template <class Matrix>
void print_banded_views(Matrix& A)
{
  using namespace mtl;
  const int M = A.nrows();
  const int N = A.nrows();

  std::cout << "Ma tran day du" << std::endl;
  print_all_matrix(A);
  std::cout << std::endl;

  typedef rows_type<Matrix>::type RowMatrix;

  std::cout << "Ma tran dai:" << std::endl;

  band_view<RowMatrix>::type B(2, 1, A);
```

```
  print_all_banded(B, 2, 1);
  std::cout << std::endl;

}
void main()
{
  using namespace mtl;
  int M =5, N = 5;
  typedef matrix<double>::type Matrix;

  Matrix A(M, N);

  for (int j = 0; j < N; ++j)
    for (int i = 0; i < M; ++i)
      A(i,j) = double(i * N + j);

  print_banded_views(A);

}
```

```
Ma tran day du
5x5
[
[0,1,2,3,4],
[5,6,7,8,9],
[10,11,12,13,14],
[15,16,17,18,19],
[20,21,22,23,24]
]

Ma tran dai:
5x5
[
[0,1,0,0,0],
[5,6,7,0,0],
[10,11,12,13,0],
[0,16,17,18,19],
[0,0,22,23,24]
]
```

Bộ chọn gồm 15 lớp. Các bộ chọn dùng để định dạng ma trận trên hai định dạng lưu trữ và hình dạng. Ví dụ bộ chọn `diagonal` dùng để định ra ma trận đường chéo. Bộ chọn `dense` dùng để định ra ma trận có kiểu lưu trữ đặc.

Giải thuật

MTL có tất cả 40 giải thuật. (So sánh: STL có 60 giải thuật). Có thể nói, các thuật toán trên ma trận đã được cung cấp đầy đủ. Có thể liệt kê ra một số

thuật toán quen thuộc: cộng hai ma trận, nhân hai ma trận, ma trận chuyển vị, ma trận nghịch đảo,... Ví dụ sau minh họa giải thuật chuyển vị ma trận `transpose`:

```
#include <mtl/matrix.h>
#include <mtl/mtl.h>
void main()
{
    using namespace mtl;
    typedef matrix< double,
                    rectangle<>,
                    dense<external>, // external cho phep sinh ma
tran tu du lieu ben nghoai
                    column_major>::type Matrix;
    const Matrix::size_type N = 3;
    Matrix::size_type large;
    double dA[] = { 1, 3, 2, 1.5, 2.5, 3.5, 4.5, 9.5, 5.5 };
    Matrix A(dA, N, N); // sinh ma tran tu mang dA
    cout << "Ma tran ban dau: " << endl;
    print_all_matrix(A);
    transpose(A); // ham giai thuat chuyen vi
    cout << "Ma tran chuyen vi: " << endl;
    print_all_matrix(A);
}
```

```
Ma tran ban dau:
3x3
[
[1,1.5,4.5],
[3,2.5,9.5],
[2,3.5,5.5]
]
Ma tran chuyen vi:
3x3
[
[1,3,2],
[1.5,2.5,3.5],
[4.5,9.5,5.5]
]
```

Ví dụ sau minh họa cách dùng khuôn hình giải thuật `tri__solve` để giải phương trình `X*T = B`.

```
#include "mtl/matrix.h"
#include "mtl/mtl.h"
#include "mtl/utils.h"
#include "mtl/linalg_vec.h"
using namespace mtl;
```

```
void main()
{
    typedef matrix< double,
                    triangle<lower>,
                    packed<>,
                    row_major >::type Matrix;
    typedef external_vec<double> Vector;
    const int N = 3;

    Matrix A(N, N);
    set_diagonal(A, 1);

    A(1,0) = 2;  A(1,1) = 4;
    A(2,0) = 3;  A(2,1) = 5; A(2,2) = 6;

    double db[] = { 7, 46, 115};
    Vector b(db, N);

    std::cout << "Ma tran tam giac A" << std::endl;
    print_row(A);

    std::cout << "Vector b:" << std::endl;
    print_vector(b);

    tri_solve(A, b);

    std::cout << "Nghiem phuong trinh A*x = b:" << std::endl;

    print_vector(b);
}
```

```
Ma tran tam giac A
[
[1,],
[2,4,],
[3,5,6,],
]
Vector b:
[7,46,115,]
Nghiem phuong trinh A*x = b:
[7,8,9,]
```

Nói chung, một bộ thư viện lớn như MTL không thể trình bày ngắn gọn bằng một phụ lục mà không có thiếu sót. Chúng tôi chỉ hy vọng qua phần phụ lục này, các bạn có được một cái nhìn khái quát về MTL và có thể sử dụng được MTL cho một số cài đặt đơn giản về ma trận. Mong rằng sẽ có dịp trình bày chi tiết hơn về bộ thư viện này trong một quyển sách khác.

B.2. Thư viện GTL

Hiện nay, GTL mới chỉ ra phiên bản 1.0.0 nhưng phiên bản này cũng cung cấp khá đủ các dạng đồ thị và các thuật toán quen thuộc trên đồ thị. GTL rất tiện dụng cho những người phát triển cần đến các cài đặt về đồ thị, cây và các thuật toán liên quan.

B.2.1. Cài đặt GTL

GTL có thể biên dịch trên Windows với Visual C++ 5.0 trở lên. GTL cũng có thể biên dịch được trên các hệ điều hành khác, chi tiết xem trong tài liệu đi kèm. Chúng tôi chỉ giới thiệu cách sử dụng GTL với Visual C++. Người đọc có thể download GTL về từ http://www.uni-passau.de/ hoặc tìm trong đĩa CD đi kèm quyển sách. Sau khi giải nén, vào thư mục src và chạy tệp GTL.dsw để mở môi trường làm việc của Visual C++ với hai đề án: GTLStatic (tạo thư viện GTL tĩnh) và GTLDynamic (tạo thư viện GTL động). Trong môi trường làm việc của Visual C++, mở tệp GTL.h và tìm đến hai dòng 81 và 83 với nội dung như sau:

```
81: define GTL_EXTERN __declspec(dllexport)
83: define GTL_EXTERN __declspec(dllimport)
```

Hãy sửa lại hai dòng trên thành:

```
81: define GTL_EXTERN //__declspec(dllexport)
83: define GTL_EXTERN //__declspec(dllimport)
```

hoặc xóa hẳn cụm __declspec(dllexport). Vì nếu để lại hai dòng trên, khi biên dịch các ứng dụng sau này sử dụng GTL sẽ bị thông báo lỗi. Sau khi đã sửa, lần lượt biên dịch hai thư viện GTLStatic và GTLDynamic cho hai cách sử dụng tĩnh hoặc động. Vào Tools > Options sau đó chọn Directories và thêm vào các đường dẫn cho các tệp tiêu đề và tệp thư viện của GTL. Ví dụ, nếu thư mục GTL được giải nén nằm ở ổ C:\, thư mục cho các tệp tiêu đề sẽ là C:\GTL\include, thư mục cho tệp thư viện tĩnh là C:\GTL\include\lib-debug, thư mục cho tệp thư viện động là C:\GTL\bin-debug. Như vậy, bạn đã hoàn thành xong phần cài đặt thư viện GTL và có thể sử dụng chúng cho các bài toán lập trình của mình. Mỗi khi sử dụng, bạn cần thiết lập đề án của mình sử dụng thư viện GTL tĩnh hay động. Ví dụ, muốn sử dụng thư viện tĩnh, vào Projects > Settings > Link và thêm GTLStatic.lib vào ô object/library modules. Một chú ý nữa, khi sử dụng GTL cùng với thư viện MFC của Visual C++ sẽ bị thông báo các lỗi liên kết rất *khó hiểu*. Để loại bỏ lỗi này, vào Projects > Settings, chọn tab Link và thêm tùy chọn /FORCE:MULTIPLE vào Projects Options.

Chương trình sau minh họa cách tạo một đồ thị với hai đỉnh và một cung.

```
#include <GTL/graph.h>

void main()
{
   graph G;
   G.make_undirected();
   node n1 = G.new_node();
   node n2 = G.new_node();
   G.new_edge(n1, n2);

   cout << G << endl;
}
```

```
[0]:: -->[1]
[1]:: -->[0]
```

Chương trình chạy cho kết quả như trên chỉ sau khi bạn đã cài đặt xong GTL và đề án của bạn được thiết lập sử dụng thư viện GTL tĩnh hay động.

Chương trình trên tạo một đồ thị `G` với hai đỉnh `n1`, `n2` và một cung giữa hai đỉnh rồi in ra. Đồ thị `G` được khởi tạo với *mặc định* là đồ thị có hướng. Hàm thành phần `make_undirected` làm cho `G` trở thành vô hướng. Nếu không có lệnh `G.make_undirected()`, kết quả sẽ là:

```
[0]:: -->[1]
[1]::
```

Bạn đọc cần chú ý tới điểm này khi sử dụng GTL.

B.2.2. Các thành phần của đồ thị

Đỉnh và cung

Một đồ thị bao gồm các đỉnh và các cung nối các đỉnh. Các lớp `node` và `edge` tương ứng dùng để biểu diễn các đỉnh và các cung. Lớp `node` có cấu tử để khởi tạo một đỉnh nhưng nói chung không nên sử dụng cách này để tạo đỉnh. Lớp `graph` có hàm thành phần `new_node` để tạo một đỉnh mới đồng thời gán luôn cho đỉnh đó một chỉ số. Trong ví dụ trước, 0 và 1 chính là các chỉ số được tạo ra và có thể làm đại diện cho các đỉnh.

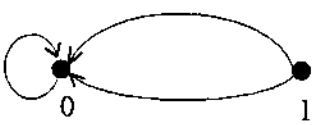
Để tạo một cung của đồ thị, nên dùng hàm thành phần `new_edge` của lớp `graph`. Một cung có thể tạo ra từ chỉ một đỉnh, cung như thế được gọi là cung tự nối vòng. Ngược lại, từ hai đỉnh có thể tạo ra được nhiều cung, các cung như thế được gọi là đa-cung. Ví dụ sau minh họa hai trường hợp trên.

```
#include <GTL/graph.h>
void main()
{
     graph G;
     node n0 = G.new_node();
     node n1 = G.new_node();
     // cung tu noi vong
     G.new_edge(n0,n0);
// da cung
     G.new_edge(n1, n0);
     G.new_edge(n1, n0);

     cout << G << endl;
}
```

```
[0]:: -->[0]
[1]:: -->[0]-->[0]
```

Đồ thị tạo ra trong ví dụ trên được minh họa bằng hình vẽ sau:



Có thể coi đồ thị như một bộ chứa. Khác với các bộ chứa của STL, đồ thị trong GTL có thể chứa hơn một kiểu dữ liệu, đó là đỉnh và cung. Để duyệt các đỉnh trên đồ thị, dùng bộ duyệt đỉnh `node_iterator`. Để duyệt các cung trên đồ thị, dùng bộ duyệt cung `edge_iterator`. Ví dụ sau minh họa hai cách duyệt trên đồ thị ứng với đỉnh và cung.

```
#include <GTL/graph.h>
void main()
{
     graph g;
     node n0 = g.new_node();
     node n1 = g.new_node();
     node n2 = g.new_node();

     edge e1 = g.new_edge(n0,n2);
     edge e2 = g.new_edge(n1,n2);

     graph::node_iterator ni = g.nodes_begin();
     for(;ni != g.nodes_end();ni++)
           cout << "Dinh thu: " << (*ni).id() << endl;
     graph::edge_iterator ei = g.edges_begin();
     for(;ei != g.edges_end();ei++)
```

```
            cout << "Canh thu: " << (*ei).id() << endl;
}
```

Quan hệ giữa các đỉnh và các cung

Một đỉnh của đồ thị có thể có các đỉnh kề, cung kề. Do đó, GTL xây dựng các bộ duyệt tương ứng là `adj_nodes_iterator` - bộ duyệt trên các đỉnh kề, `adj_edges_iterator` - bộ duyệt trên các cung kề. Khái niệm đỉnh kề và cung kề thay đổi khi đồ thị là vô hướng hay có hướng. Nếu đồ thị có hướng, cung kề là các cung đi ra từ đỉnh. Đỉnh kề là các đỉnh nằm trên cung kề. Nếu đồ thị vô hướng, cung kề là các cung nối tới đỉnh, đỉnh kề là các đỉnh nằm trên cung kề. Với đồ thị có hướng, để duyệt trên các cung đi vào hay cung đi ra, ta có các bộ duyệt `in_edges_iterator` và `out_edges_iterator`. Ngoài ra để duyệt trên tất cả các cung, không phân biệt đi vào hay đi ra ta có bộ duyệt `inout_edges_iterator`. Ví dụ sau minh họa các cách duyệt đỉnh kề, cung kề, cung đi vào, cung đi ra trên một nút.

```
#include <GTL/graph.h>

void main()
{

    graph g;
    node n[4];
    for (int i=0;i<4;i++)
          n[i] = g.new_node();
    edge e[4];
    e[0] = g.new_edge(n[1],n[0]);
    e[1] = g.new_edge(n[2],n[0]);
    e[2] = g.new_edge(n[0],n[1]);
    e[4] = g.new_edge(n[0],n[3]);

    cout << "So dinh cua do thi: " << g.number_of_nodes() <<
endl;
    cout << "So cung cua do thi: " << g.number_of_edges() <<
endl;
    // duyet dinh ke
    node::adj_nodes_iterator ani = n[0].adj_nodes_begin();
    cout << "Duyet dinh ke voi dinh " << n[0].id() << endl;
    for (;ani != n[0].adj_nodes_end(); ani++)
    {
          cout << "Duyet dinh " << (*ani).id() << endl;
    }
    // duyet cung ke
    node::adj_edges_iterator aei = n[0].adj_edges_begin();
    cout << "Duyet cung ke voi dinh " << n[0].id() << endl;
```

```
    for (;aei != n[0].adj_edges_end(); aei++)
    {
        cout << "Duyet cung " << (*aei).id() << endl;
    }
    // duyet cung di vao
    node::in_edges_iterator iei = n[0].in_edges_begin();
    cout << "Duyet cung di vao dinh " << n[0].id() << endl;
    for (;iei != n[0].in_edges_end(); iei++)
    {
        cout << "Duyet cung " << (*iei).id() << endl;
    }
    // duyet cung di ra
    node::out_edges_iterator oei = n[0].out_edges_begin();
    cout << "Duyet cung di ra tu dinh " << n[0].id() << endl;
    for (;oei != n[0].out_edges_end(); oei++)
    {
        cout << "Duyet cung " << (*oei).id() << endl;
    }
}
```

```
So dinh cua do thi: 4
So cung cua do thi: 4
Duyet dinh ke voi dinh 0
Duyet dinh 3
Duyet dinh 1
Duyet cung ke voi dinh 0
Duyet cung 3
Duyet cung 2
Duyet cung di vao dinh 0
Duyet cung 1
Duyet cung 0
Duyet cung di ra tu dinh 0
Duyet cung 3
Duyet cung 2
```

Kết quả trên cho thấy, các cung kề cũng là các cung đi ra. Điều này phù hợp với lý thuyết vì `g` được khởi tạo mặc định là đồ thị có hướng. Nếu thêm lệnh `g.make_undirected()` vào sau lệnh khởi tạo `g`, sẽ được kết quả khác:

```
So dinh cua do thi: 4
So cung cua do thi: 4
Duyet dinh ke voi dinh 0
Duyet dinh 3
Duyet dinh 1
Duyet dinh 2
Duyet dinh 1
Duyet cung ke voi dinh 0
Duyet cung 3
Duyet cung 2
```

```
Duyet cung 1
Duyet cung 0
Duyet cung di vao dinh 0
Duyet cung 1
Duyet cung 0
Duyet cung di ra tu dinh 0
Duyet cung 3
Duyet cung 2
```

Lúc này, số cung kề bằng tổng của số cung đi vào và số cung đi ra.

Gán dữ liệu cho đỉnh và cung của đồ thị

Lớp graph của GTL biểu diễn cho các đồ thị dưới góc độ toán học. Đồ thị tạo ra bởi graph chưa có dữ liệu thực, chỉ có các đỉnh, cung và mối quan hệ toán học giữa chúng. Với người lập trình, đồ thị còn cần có dữ liệu thực. Ví dụ, muốn vẽ một đồ thị, các đỉnh cần có tọa độ. Với bài toán tìm đường đi ngắn nhất giữa các thành phố, các cung của đồ thị cần có trọng số. Để giải quyết các vấn đề trên, có thể dùng bộ chứa map của STL. Nhưng GTL đưa ra một giải pháp thuận tiện hơn, đó là dùng các lớp khuôn hình node_map và edge_map.

Lớp node_map dùng ánh xạ giữa các đỉnh và dữ liệu. Lớp edge_map dùng ánh xạ giữa các cung và dữ liệu. Ví dụ để gán nhãn cho các đỉnh, ta có thể làm như sau:

```
void main()
{
    graph g;
    node n = g.new_node();
    node_map<string> node_label(g,"Nhan mac dinh");
    cout << "Nhan dinh n: " << node_label[n] << endl;
    node_label[n] = "A";
    cout << "Nhan dinh n: " << node_label[n] << endl;
}
```

```
Nhan dinh n: Nhan mac dinh
Nhan dinh n: A
```

Cấu tử để khởi tạo một đối tượng lớp node_map (hay edge_map) cần hai tham số: đồ thị muốn ánh xạ đỉnh (hay cung) và giá trị mặc định để ánh xạ. Nếu không có phép gán lại sau đó, các đỉnh (cung) sẽ nhận giá trị mặc định ban đầu như trong ví dụ trên.

Ví dụ sau minh họa rõ hơn cách dùng hai lớp này.

```
#include <GTL/graph.h>
#include <GTL/node_map.h>
#include <GTL/edge_map.h>
#include <math.h>
template <class T>
class point
{
private:
    T _x;  // toa do x
    T _y;  // toa do y
public:
    point(T x,T y):_x(x),_y(y){};
    point(){};
    ~point() {};
    T getx() { return _x; };
    T gety() { return _y; };
    void setxy(T x,T y) { _x = x; _y = y; };
};

void main()
{
    graph g;
    // anh xa giua cac dinh va cac toa do
    node_map<point<int> > node_point(g, point<int>(0,0));
    // anh xa giua cac cung va chieu dai cung
    edge_map<float> edge_length(g,0);

    point<int> p[4];
    p[0].setxy(0,0);
    p[1].setxy(0,1);
    p[2].setxy(1,0);
    p[3].setxy(1,1);

    node n[4];
    edge e[3];

    for (int i = 0; i<4; i++)
    {
        n[i] = g.new_node();
        // gan toa do cho cac dinh
        node_point[n[i]] = p[i];
    }

    e[0] = g.new_edge(n[0],n[1]);
    e[1] = g.new_edge(n[0],n[3]);
    e[2] = g.new_edge(n[2],n[3]);
    // gan chieu dai cho cac cung
    edge_length[e[0]] = 1;
    edge_length[e[1]] = sqrt(2);
```

```
    edge_length[e[2]] = 1;

    cout << "Toa do cac dinh cua do thi: " << endl;
    for (i = 0; i < 4; i++)
    {
        cout << "(" << node_point[n[i]].getx();
        cout << "," << node_point[n[i]].gety() << ")" <<
endl;
    }

    cout << "Do dai cac cung cua do thi: " << endl;
    for (i = 0; i < 3; i++)
    {
        node s = e[i].source();
        node t = e[i].target();
        cout << "Khoang cach tu (" << node_point[s].getx();
        cout << "," << node_point[s].gety() << ") toi (";
        cout << node_point[t].getx() << "," <<
node_point[t].gety();
        cout << ") la: " << edge_length[e[i]] << endl;
    }
}
```

```
Toa do cac dinh cua do thi:
(0,0)
(0,1)
(1,0)
(1,1)
Do dai cac cung cua do thi:
Khoang cach tu (0,0) toi (0,1) la: 1
Khoang cach tu (0,0) toi (1,1) la: 1.41421
Khoang cach tu (1,0) toi (1,1) la: 1
```

Các thuật toán trên đồ thị

Tất cả các thuật toán của GLT được xây dựng dưới dạng lớp mà không phải là giải thuật như trong STL. Điều này là do có rất nhiều thứ liên quan tới một thuật toán đồ thị: các điều kiện đủ để áp dụng thuật toán, các kết quả thực hiện thuật toán, các tùy chọn cho kết quả... Không thể đóng gói tất cả vào một giải thuật được. Rõ ràng, trong trường hợp này, xây dựng các thuật toán dưới dạng lớp sẽ có lợi hơn.

Do mới là phiên bản 1.0.0, các cài đặt cho thuật toán của GTL chưa nhiều. Chúng bao gồm:

- `dfs`: tìm kiếm theo chiều sâu trên đồ thị.
- `bfs`: tìm kiếm theo chiều rộng trên đồ thị.

- `biconnectivity`: kiểm tra tính liên thông hai phía.
- `components`: phát hiện các thành phần liên thông.
- `topsort`: sắp xếp topo.
- `st_number`: đánh số st.
- `maxflow_pp` và `maxflow_ff`: tìm luồng cực đại.
- `fm_partition` và `radio_cut_partition`: phân hoạch đồ thị.
- `planarity`: kiểm tra tính phẳng của đồ thị.

Tất cả các thuật toán trên đều xây dựng từ lớp `algorithm`, do đó có một giao diện chung:

- `check`: hàm thành phần kiểm tra thuật toán có áp dụng với đồ thị được không. Ứng với mỗi thuật toán, một đồ thị phải thỏa mãn một số điều kiện nào đó mới áp dụng được thuật toán. Do đó, nên gọi tới `check` trước khi gọi `run`.

- `run`: thực hiện thuật toán trên đồ thị. Sau khi thực hiện, một số thành phần của thuật toán dùng để lưu kết quả sẽ thay đổi. Ví dụ, với thuật toán BFS hay DFS ta sẽ có các số lưu thứ tự duyệt.

- `reset`: thiết lập lại trạng thái ban đầu của thuật toán để có thể thực hiện lại hoặc thực hiện trên một đồ thị khác. Lưu ý, chỉ các thành phần lưu kết quả của thuật toán bị thiết lập lại, các thành phần tùy chọn không bị thiết lập lại.

Ví dụ dưới đây minh họa hai thuật toán cơ bản nhất của đồ thị là DFS và BFS.

```
#include <GTL/graph.h>
#include <GTL/dfs.h>
#include <GTL/bfs.h>
void main()
{
     graph g;
     g.make_undirected();
     node n[6];
     for (int i = 0; i < 6; i++)
           n[i] = g.new_node();
     g.new_edge(n[0],n[1]);
     g.new_edge(n[0],n[3]);
     g.new_edge(n[0],n[5]);
     g.new_edge(n[2],n[5]);
     g.new_edge(n[3],n[4]);
     cout << "------------ LOADED GRAPH ------------" << endl;
     cout << g << endl;
```

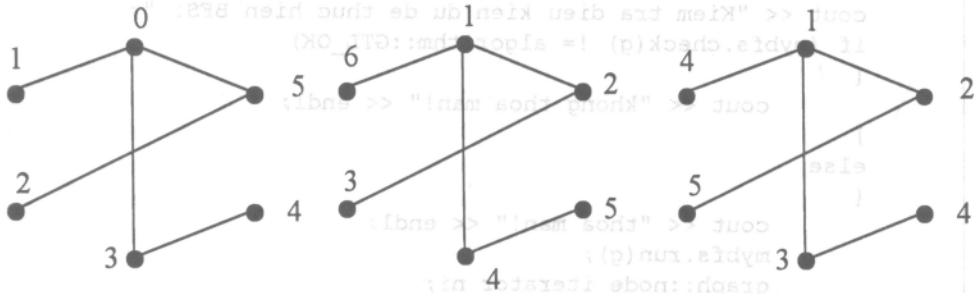
```
    cout << "----------------- DFS -----------------" << endl;
    dfs mydfs;
    mydfs.start_node(n[0]); // duyet tu nut n[0]
    cout << "Kiem tra dieu kien du de thuc hien DFS: ";
    if (mydfs.check(g) != algorithm::GTL_OK)
    {
          cout << "khong thoa man!" << endl;
    }
    else
    {
          cout << "thoa man!" << endl;
          mydfs.run(g);
          graph::node_iterator ni;
          for (ni = g.nodes_begin(); ni != g.nodes_end(); ni++)
          {
                cout << "   Dinh " << *ni;
                cout << " -- thu tu duyet bang dfs -- " <<
mydfs[*ni];
                cout << endl;
          }
    }
    cout << "----------------- BFS -----------------" << endl;
    bfs mybfs;
    mybfs.start_node(n[0]); // duyet tu nut n[0]
    mybfs.calc_level(true); // co tinh bac bfs cho moi dinh
    cout << "Kiem tra dieu kien du de thuc hien BFS: ";
    if (mybfs.check(g) != algorithm::GTL_OK)
    {
          cout << "khong thoa man!" << endl;
    }
    else
    {
          cout << "thoa man!" << endl;
          mybfs.run(g);
          graph::node_iterator ni;
          for (ni = g.nodes_begin(); ni != g.nodes_end(); ni++)
          {
                cout << "   Dinh " << *ni;
                cout << " -- thu tu duyet bang bfs -- " <<
mybfs[*ni];
                cout << " -- bac bfs -- " << mybfs.level(*ni)
<< endl;
          }
    }
}
```

```
------------ LOADED GRAPH ------------
[0]:: <-->[5]<-->[3]<-->[1]
[1]:: <-->[0]
[2]:: <-->[5]
```

```
[3]:: <-->[4]<-->[0]
[4]:: <-->[3]
[5]:: <-->[2]<-->[0]

----------------- DFS -----------------
Kiem tra dieu kien du de thuc hien DFS: thoa man!
   Dinh [0] -- thu tu duyet bang dfs -- 1
   Dinh [1] -- thu tu duyet bang dfs -- 6
   Dinh [2] -- thu tu duyet bang dfs -- 3
   Dinh [3] -- thu tu duyet bang dfs -- 4
   Dinh [4] -- thu tu duyet bang dfs -- 5
   Dinh [5] -- thu tu duyet bang dfs -- 2
----------------- BFS -----------------
Kiem tra dieu kien du de thuc hien BFS: thoa man!
   Dinh [0] -- thu tu duyet bang bfs -- 1 -- bac bfs -- 0
   Dinh [1] -- thu tu duyet bang bfs -- 4 -- bac bfs -- 1
   Dinh [2] -- thu tu duyet bang bfs -- 5 -- bac bfs -- 2
   Dinh [3] -- thu tu duyet bang bfs -- 3 -- bac bfs -- 1
   Dinh [4] -- thu tu duyet bang bfs -- 6 -- bac bfs -- 2
   Dinh [5] -- thu tu duyet bang bfs -- 2 -- bac bfs -- 1
```

Hình vẽ dưới đây minh họa đồ thị g, thứ tự các đỉnh được thăm bằng thuật toán BFS và DFS.



Đồ thị ban đầu Đồ thị DFS Đồ thị BFS

Vì số trang hạn chế của quyển sách, chúng tôi không thể giới thiệu chi tiết hơn về hai bộ thư viện GTL và MTL. Hy vọng với những gì đã trình bày, bạn đọc biết được thêm hai bộ thư viện hữu dụng nữa trong lập trình. Ngoài ra, bạn đọc có thể tham khảo tài liệu về GTL, MTL và một số bộ thư viện khác trong đĩa CD đi kèm quyển sách. Mọi thắc mắc về cách cài đặt, chạy thử chương trình, xin gửi tới chúng tôi theo địa chỉ: thuynt@it-hut.edu.vn.

Phụ lục C

CÁC TRANG WEB HỮU ÍCH VỀ STL

1. http://www.sgi.com/Tech/STL/ trang chủ về STL của hãng Silicon Graphic.
2. http://infosun.fmi.uni-passau.de/GTL/ trang chủ của GTL.
3. http://www.osl.iu.edu/research/mtl/ trang chủ của MTL.
4. http://www.boost.org/ trang chủ của tổ chức Boost, tổ chức quản lý thư viện BTL.
5. http://www.geocities.com/corwinjoy/dtl/index.htm trang chủ của DTL
6. http://dtemplatelib.sourceforge.net/index.htm trang về DTL trên sourceforge.net. Nếu geocities.com bị chặn proxy, có thể qua trang web này để lấy DTL.
7. http://www.mysql.com/downloads/api-mysql++.html: trang về MySQL++.
8. http://sourceforge.net/projects/cgi-plus-plus/ trang về CGI++ trên sourcforge.net.
9. http://www.stlport.org/ Trang chủ của STLPort.
10. http://directory.google.com/Top/Computers/Programming/Languages/C++/Class_Libraries/STL/ các địa chỉ về STL trên google.com. Trên trang này giới thiệu rất nhiều địa chỉ liên kết tới những trang có bài báo, sách, hướng dẫn sử dụng, thư viện, ... liên quan đến STL.
11. http://www.bruceeckel.com trang chủ của Bruce Eckel, tác giả Thinking In C++. Tại đây bạn có thể lấy về các sách của B.Eckel miễn phí.
12. http://www.research.att.com/~bs/homepage.html Trang chủ của Bjarne Stroustrup.
13. http://www.cygnus.com/misc/wp/index.html Chuẩn ANSI/ISO C++.

STL - LẬP TRÌNH KHÁI LƯỢC TRONG C++

TÁC GIẢ: NGUYỄN THANH THUỶ (Chủ biên)

NGUYỄN HỮU ĐỨC, ĐẶNG CÔNG KIÊN

DOÃN TRUNG TÙNG

Chịu trách nhiệm xuất bản: Pgs.Ts. Tô Đăng Hải

Biên tập: Nguyễn Thị Ngọc Khuê

Vẽ bìa: Trần Thắng

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ HTUẬT

70 TRẦN HƯNG ĐẠO - HÀ NỘI

In 1000 cuốn, khổ 16cm x 24cm tại Công ty in Hàng không Gia Lâm

Giấy phép xuất bản số: 486 - 34 - 17/12/2002

In xong và nộp lưu chiểu tháng 1 năm 2003

Người lập trình viên giỏi là người có khả năng tổ chức, xây dựng mã nguồn một cách hiệu quả, kế thừa và khai thác được tối đa thành quả của bản thân cũng như của các nhóm phát triển khác, được đóng gói dưới dạng các thư viện phần mềm. Xuất phát từ nhu cầu đó, trong cuốn sách này, chúng tôi giới thiệu với bạn đọc cách tiếp cận lập trình khái lược - một cách tiếp cận đặc biệt - thông qua **thư viện khuôn hình chuẩn STL** (**Standard Template Library**).

Cuốn sách này dành cho sinh viên chuyên ngành công nghệ thông tin đồng thời cũng nhằm tới đối tượng là các nhà thiết kế và phát triển phần mềm ứng dụng. Hy vọng đây sẽ là một tài liệu tham khảo tốt đối với các giáo viên công nghệ thông tin khi xây dựng giáo trình lập trình hướng đối tượng trong ngôn ngữ C++.

202254

8 935048 922545

Giá: 42.000đ