BỘ GIAO THÔNG VẬN TẢI
TRƯỜNG ĐẠI HỌC HÀNG HẢI VIỆT NAM
KHOA: CÔNG NGHỆ THÔNG TIN
BỘ MÔN: KHOA HỌC MÁY TÍNH

BÀI GIẢNG

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VÀ C++

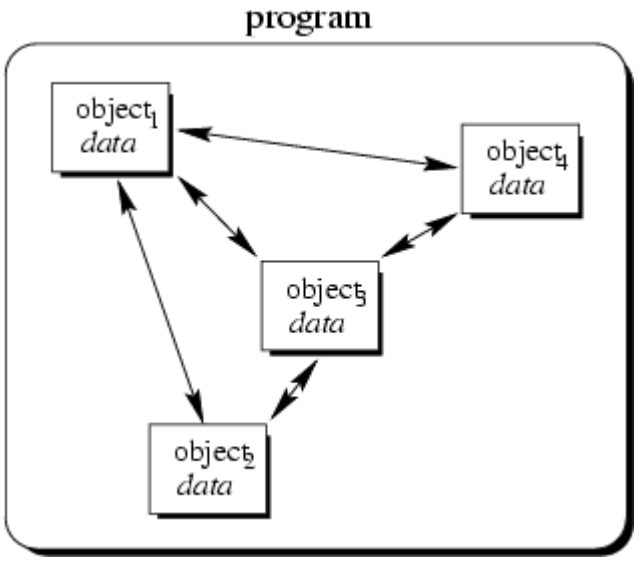
TÊN HỌC PHẦN: **Lập trình hướng đối tượng và C++**
MÃ HỌC PHẦN: 17209
TRÌNH ĐỘ ĐÀO TẠO: **ĐẠI HỌC CHÍNH QUY**
DÙNG CHO SV NGÀNH: **CÔNG NGHỆ THÔNG TIN**

HẢI PHÒNG - 2014

Mục lục

CHƯƠNG 1. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VÀ NGÔN NGỮ C++

1.1. Ưu điểm của lập trình hướng đối tượng



Lập trình hướng đối tượng. Các đối tượng tương tác với nhau bằng cách gửi các thông điệp.

Trong lập trình hướng đối tượng trong mỗi chương trình chúng ta có một số các đối tượng (object) có thể tương tác với nhau, thuộc các lớp (class) khác nhau, mỗi đối tượng tự quản lý lấy các dữ liệu của riêng chúng.

Chương trình chính sẽ bao gồm một số đối tượng là thể hiện (*instance*) của các lớp, các đối tượng này tương tác với nhau thực hiện các chức năng của chương trình. Các lớp trong lập trình hướng đối tượng có thể xem như là một sự trừu tượng ở mức cao hơn của các cấu trúc (struct hay record) hay kiểu dữ liệu do người dùng định nghĩa trong các ngôn ngữ lập trình có cấu trúc với sự tích hợp cả các toán tử và dữ liệu trên các kiểu đó.

Các ưu điểm của lập trình hướng đối tượng:

Lập trình hướng đối tượng ra đời đã giải quyết được nhiều nhược điểm tồn tại trong lập trình có cấu trúc. Trong lập trình OOP có ít lỗi hơn và việc gỡ lỗi cũng đơn giản hơn, đồng thời lập trình theo nhóm có thể thực hiện rất hiệu quả. Ít lỗi là một trong các ưu điểm chính của OOP vì theo thống kê thì việc bảo trì hệ thống phần mềm sau khi giao cho người dùng chiếm tới 70% giá thành phần mềm.

Việc thay đổi các cài đặt chi tiết bên dưới trong lập trình OOP không làm ảnh hưởng tới các phần khác của chương trình do đó việc mở rộng qui mô của một chương trình dễ dàng hơn, đồng thời làm giảm thời gian cần thiết để phát triển phần mềm.

Với khái niệm kế thừa các lập trình viên có thể xây dựng các chương trình từ các phần mềm sẵn có.

OOP có tính khả chuyển cao. Một chương trình viết trên một hệ thống nền (chẳng hạn Windows) có thể chạy trên nhiều hệ thống nền khác nhau (chẳng hạn Linux, Unix…).

OOP có hiệu quả cao. Thực tế cho thấy các hệ thống được xây dựng bằng OOP có hiệu năng cao.

1.2. Một số khái niệm cơ bản của lập trình hướng đối tượng

1.2.1. Kiểu dữ liệu trừu tượng ADT(Astract Data Type)

Định nghĩa về kiểu dữ liệu trừu tượng:

Một kiểu dữ liệu trừu tượng là một mô hình toán học của các đối tượng dữ liệu tạo thành một kiểu dữ liệu và các toán tử (phép toán) thao tác trên các đối tượng đó. Đặc tả về một kiểu dữ liệu trừu tượng không có bất kỳ một chi tiết cụ thể nào về cài đặt bên trong của kiểu dữ liệu.

Ví dụ về kiểu dữ liệu trừu tượng: Số nguyên.

1.2.2. Đối tượng (Objects) và lớp (Classes)

Trong một chương trình hướng đối tượng chúng ta có các đối tượng. Các đối tượng này là đại diện cho các đối tượng thực trong thực tế. Có thể coi khái niệm đối tượng trong OOP chính là các kiểu dữ liệu trong các ngôn ngữ lập trình có cấu trúc. Mỗi một đối tượng có các dữ liệu riêng của nó và được gọi là các member variable hoặc là các data member. Các toán tử thao tác trên các dữ liệu này được gọi là các member function.

Mỗi một đối tượng là thể hiện (instance) của một lớp. Như vậy lớp là đại diện cho các đối tượng có các member function giống nhau và các data member cùng kiểu. Lớp là một sự trừu tượng hóa của khái niệm đối tượng. Tuy nhiên lớp không phải là một ADT, nó là một cài đặt của một đặc tả ADT. Các đối tượng của cùng một lớp có thể chia sẻ các dữ liệu dùng chung, dữ liệu kiểu này được gọi là class variable.

1.2.3. Kế thừa (Inheritance)

Khái niệm kế thừa này sinh từ nhu cầu sử dụng lại các thành phần phần mềm để phát triển các phần mềm mới hoặc mở rộng chức năng của phần mềm hiện tại. Kế thừa là một cơ chế cho phép các đối tượng của một lớp có thể truy cập tới các member variable và function của một lớp đã được xây dựng trước đó mà không cần xây dựng lại các thành phần đó. Điều này cho phép chúng ta có thể tạo ra các lớp mới là một mở rộng hoặc cá biệt hóa của một lớp sẵn có. Lớp mới (gọi là derived class) kế thừa từ lớp cũ (gọi là lớp cơ sở base class). Các ngôn ngữ lập trình hướng đối tượng có thể hỗ trợ khái niệm đa kế thừa cho phép một lớp có thể kế thừa từ nhiều lớp cơ sở.

Lớp kế thừa derived class có thể có thêm các data member mới hoặc các member function mới. Thêm vào đó lớp kế thừa có thể tiến hành định nghĩa lại một hàm của lớp cơ sở và trong trường hợp này người ta nói rằng lớp kế thừa đã overload hàm thành viên của lớp cơ sở.

1.2.4. Dynamic Binding (tạm dịch là ràng buộc động) và Porlymorphism (đa xạ hoặc đa thể)

Chúng ta lấy một ví dụ để minh hoạ cho hai khái niệm này. Giả sử chúng ta có một lớp cơ sở là Shape, hai lớp kế thừa từ lớp Shape là Circle và Rectange. Lớp Shape là một lớp trừu tượng có một member function trừu tượng là draw(). Hai lớp Circle và Rectange thực hiện overload lại hàm draw của lớp Shape với các chi tiết cài đặt khác nhau chẳng hạn với lớp

Circle hàm draw sẽ vẽ một vòng tròn còn với lớp Rectange thì sẽ vẽ một hình chữ nhật. Và chúng ta có một đoạn chương trình chính hợp lệ như sau:

```
int main(){
  Shape shape_list[4];
  int choose, i;
  for(i=0;i<4;i++){
        cout << "Ngai muon ve hinh tron(0) hay hinh chu nhat(1)"; cin >> choose;
        if(choose==0){shape_list[i] = new Circle();}
        else{shape_list[i] = new Rectange();}
}
for(i=0;i<4;i++) {shape_list[i]->draw();}
}
```

Khi biên dịch chương trình này thành mã thực hiện (file .exe) trình biên dịch không thể xác định được trong mảng shape_list thì phần tử nào là Circle phần tử nào là Rectange và do đó không thể xác định được phiên bản nào của hàm draw sẽ được gọi thực hiện. Việc gọi tới phiên bản nào của hàm draw để thực hiện sẽ được quyết định tại thời điểm thực hiện chương trình, sau khi đã biên dịch và điều này được gọi là dynamic binding hoặc late binding. Ngược lại nếu việc xác định phiên bản nào sẽ được gọi thực hiện tương ứng với dữ liệu gắn với nó được quyết định ngay trong khi biên dịch thì người ta gọi đó là static binding.

Ví dụ này cũng cung cấp cho chúng ta một minh họa về khả năng đa thể (polymorphism). Khái niệm đa thể được dùng để chỉ khả năng của một thông điệp có thể được gửi tới cho các đối tượng của nhiều lớp khác nhau tại thời điểm thực hiện chương trình. Chúng ta thấy rõ lời gọi tới hàm draw sẽ được gửi tới cho các đối tượng của hai lớp Circle và Rectange tại thời điểm chương trình được thực hiện.

Ngoài các khái niệm cơ bản trên OOP còn có thêm một số khái niệm khác chẳng hạn như name space và exception handling nhưng không phải là các khái niệm bản chất.

1.3. Ngôn ngữ lập trình C++ và OOP.

C++ là một ngôn ngữ lập trình hướng đối tượng được Bjarne Stroustrup (AT & T Bell Lab) (giải thưởng ACM Grace Murray Hopper năm 1994) phát triển từ ngôn ngữ C. C++ kế thừa cú pháp và một số đặc điểm ưu việt của C: ví dụ như xử lý con trỏ, thư viện các hàm phong phú đa dạng, tính khả chuyển cao, chương trình chạy nhanh …. Tuy nhiên về bản chất thì C++ khác hoàn toàn so với C, điều này là do C++ là một ngôn ngữ lập trình hướng đối tượng và có nhiều mở rộng hơn so với C. Ta sẽ tìm hiểu sự khác biệt của C++ ở các chương kế tiếp.

1.4. Cấu trúc một chương trình C++

Một chương trình C/C++ có thể được đặt trong một hoặc nhiều file văn bản khác nhau. Mỗi file văn bản chứa một số phần nào đó của chương trình. Với những chương trình đơn giản và ngắn thường chỉ cần đặt chúng trên một file.

Một chương trình gồm nhiều hàm, nhiều lớp, nhiều đối tượng, … Mỗi hàm phụ trách một công việc khác nhau của chương trình. Đặc biệt trong các hàm này có một hàm duy nhất có tên hàm là main(). Khi chạy chương trình, các câu lệnh trong hàm main() sẽ được thực hiện đầu tiên. Trong hàm main() có thể có các câu lệnh gọi đến các hàm khác khi cần thiết, và các hàm này khi chạy lại có thể gọi đến các hàm khác nữa đã được viết trong chương trình (trừ việc gọi quay lại hàm main()). Sau khi chạy đến lệnh cuối cùng của hàm main() chương trình sẽ kết thúc.

Cụ thể, thông thường một chương trình gồm có các nội dung sau:

− Phần khai báo các tệp nguyên mẫu: khai báo tên các tệp chứa những thành phần có sẵn (như các hằng chuẩn, kiểu chuẩn và các hàm chuẩn) mà NSD sẽ dùng trong chương trình.

− Phần khai báo các kiểu dữ liệu, các biến, hằng ... do NSD định nghĩa và được dùng chung trong toàn bộ chương trình.

− Danh sách các hàm của chương trình (do NSD viết, bao gồm cả hàm main()).

Cấu trúc chi tiết của mỗi hàm sẽ được đề cập đến trong các chương sau.

Ví dụ 1: In ra màn hình dòng chữ: “Chao ban. Day la chuong trinh C++ dau tien”

```
1.#include<iostream> //khai bao thu vien
2. using namespace std;
3.int main()
4.{
5.cout<<"Chao ban. Day la chuong trinh C++ dau tien";
6.}
```

Dòng đầu tiên của chương trình là khai báo tệp nguyên mẫu iostream. Đây là khai báo bắt buộc vì trong chương trình có sử dụng phương thức chuẩn “cout <<” (in ra màn hình), phương thức này được khai báo và định nghĩa sẵn trong iostream.

Không riêng hàm main(), mọi hàm khác đều phải bắt đầu tập hợp các câu lệnh của mình bởi dấu { và kết thúc bởi dấu }. Tập các lệnh bất kỳ bên trong cặp dấu này được gọi là khối lệnh.

1.5. Kiểu dữ liệu trong C++

Các kiểu dữ liệu cơ bản của C++ hầu hết đều kế thừa của C ngoại trừ kiểu bool với hai hằng số true và false.

Các kiểu dữ liệu cơ bản và các biến thể của chúng là cần thiết tuy nhiên nếu chỉ dùng chúng thì cũng không thể tạo nên các chương trình có ý nghĩa được. C và C++ cung cấp cho chúng ta rất nhiều cơ chế khác nhau để xây dựng lên các kiểu tích hợp có ý nghĩa hơn, phức tạp hơn và phù hợp với nhu cầu của chương trình hơn. Kiểu dữ liệu người dùng định nghĩa quan trọng nhất của C là struct, và của C++ là class. Tuy nhiên các dễ nhất để định nghĩa một kiểu mới là dùng từ khóa **typedef** để đặt bí danh cho một kiểu sẵn có.

Thiết lập các tên bí danh với từ khóa typedef

typedef có nghĩa là “type definition” nhưng những gì mà từ khóa này thực sự làm không giống như đúng ngữ nghĩa của hai từ “type definition”. Cú pháp sử dụng với từ typedef:

typedef <một kiểu đã có sẵn> <tên kiểu muốn đặt>

Ví dụ:

```
typedef unsigned long ulong;
typedef struct str_list{
  int data;
  struct str_list * next;
}list;
```

Sau khi khai báo như trên chúng ta có thể khai báo trong chương trình như sau:

```
list * aList, anotherList;
```

Kiểu dữ liệu cấu trúc với từ khóa struct

Kiểu dữ liệu cấu trúc là một cách cho phép các lập trình viên nhóm một nhóm các biến thuộc các kiểu dữ liệu khác nhau tạo thành một cấu trúc. Với một kiểu struct chúng ta có thể truy cập tới các thành phần dữ liệu qua các toán tử tham chiếu "." và "->" với một con trỏ cấu trúc. Có một điều đặc biệt khi chúng ta khai báo và sử dụng các cấu trúc trong C++:

```
typedef struct{ // hoặc có thể là: typedef struct list
  int data;
  list *next;
}list;
main(){
  list * ptr;
}
```

Kiểu dữ liệu liệt kê (enum)

Kiểu dữ liệu liệt kê được khai báo bằng từ khóa enum và thường được dùng để làm chương trình sáng sủa hơn. Ví dụ:

```
enum Bool{true = 1, false = 0};
```

Thường trong C++ kiểu enum được dùng ít hơn do một số nguyên nhân ví dụ như kiểm tra kiểu, chẳng hạn chúng ta không thể thực hiện lệnh true++.

Kiểu hợp nhất (union)

Đôi khi trong chương trình chúng ta cần phải làm việc với nhiều kiểu dữ liệu khác nhau với cùng một biến số., khi đó chúng ta có thể thực hiện theo hai cách: một là dùng kiểu dữ liệu cấu trúc hoặc nếu có thể sử dụng kiểu hợp nhất để tiết kiệm bộ nhớ.

Kiểu union có cách khai báo giống hệt kiểu struct chỉ có khác ở cách sử dụng: tại một thời điểm chúng ta chỉ có thể truy cập tới một thành phần của một biến kiểu union và kích thước của một kiểu dữ liệu union chính bằng kích thước của thành phần có kích thước lớn nhất của kiểu.

Ví dụ:

```
union test_type{
  char c; int i;    float f; double d;
```

```
};
#include <iostream>
union Packed { // khai báo giống như một lớp
  char i;   short j;  int k;  long l;  float f;  double d;
};
int main() {
  cout << "sizeof(Packed) = "  << sizeof(Packed) << endl;
  Packed x;
  x.i = 'c';
  cout << x.i << endl;
  x.d = 3.14159;
  cout << x.d << endl;
}
```

khi đó sizeof(test_type) sẽ bằng 8 bằng kích thước của kiểu double. Chú ý là sau khi thực hiện gán giá trị cho các thành phần của một biến có kiểu union thì các giá trị được gán trước đó của một thành phần khác sẽ bị sửa đổi. Ví dụ chúng ta chỉ cần thêm một lệnh: cout << x.i << endl; vào cuối chương trình trên sẽ nhận được kết quả là ‘n’ chứ không phải ‘c’.

Kiểu dữ liệu mảng: Mảng là một kiểu dữ liệu tích hợp rất hay được dùng, nó cho phép chúng ta kết hợp nhiều biến đơn lẻ có cùng kiểu thành một kiểu dữ liệu tích hợp. Việc truy cập tới các thành phần của một mảng được thực hiện bằng cách lấy chỉ mục (index) của nó: []. Việc khai báo mảng có thể được thực hiện kèm với việc gán các giá trị cho các thành phần của nó.

Mảng và con trỏ.

Con trỏ là một công cụ tuyệt vời cho phép truy cập tới các thành phần của một mảng bất kỳ. Đặc biệt là khi chúng ta cần truyền một mảng làm tham số của một hàm. Ví dụ:

```
#include <iostream> #include <string.h>
void func1(int a[], int size) {  for(int i = 0; i < size; i++) a[i] = i * i - i; }
void func2(int* a, int size) {  for(int i = 0; i < size; i++) a[i] = i * i + i; }
void print(int a[], string name, int size) {
  for(int i = 0; i < size; i++)    cout << name << "[" << i << "] = " << a[i] << endl;
}
int main() {  int a[5], b[5];  func1(a, 5);  func1(b, 5);  func2(a, 5);  func2(b, 5); }
```

Một trường hợp đặc biệt của các hàm kiểu này chính là bản thân hàm main. Hàm main có hai tham số là (int n, char * ags[]). Để thuận tiện cho việc sử dụng các tham số này chúng ta nên dùng các hàm chuyển kiểu chẳng hạn như: atoi(), atof(), atol().

1.6. Các câu lệnh trong C++

Các câu lệnh cơ bản của C++ kế thừa của C. Như lệnh rẽ nhánh if else, switch, lệnh lặp for, while, do…

Bài tập

1. Viết chương trình giải pt ax+b=0. Với a, b là các số nguyên (thực) nhập từ bàn phím

2. Viết chương trình giải pt $ax^2+bx+c=0$. Với a, b,c là các số nguyên (thực) nhập từ bàn phím

3. Viết chương trình giải hệ pt bậc nhất 2 ẩn

4. Viết chương trình nhập vào một số nguyên từ 1, 2, …, 9. In ra dạng La mã của số tương ứng. Ví dụ: nhập vào 5 in ra V, nhập vào 9 in ra IX

5. Viết chương trình in ra tổng các chữ số của một số nguyên

CHƯƠNG 2. HÀM

Các hàm là công cụ chính cho phép xây dựng các chương trình lớn trong C và C++. Với các hàm nhỏ một chương trình lớn có thể được chia thành các chương trình nhỏ hơn, dễ giải quyết hơn. Với các ngôn ngữ lập trình khác nhau người ta dùng các thuật ngữ khác nhau để chỉ một chương trình con, trong C và C++ các chương trình con được gọi là các hàm.

2.1. Xây dựng hàm

2.1.1. Nguyên mẫu hàm

Một hàm trong C và C++ thường được khai báo nguyên mẫu trước khi thực sự cài đặt (định nghĩa). Cú pháp khai báo nguyên mẫu của một hàm như sau:

<kiểu của hàm> <tên hàm>(<danh sách tham số>);

Trong đó <kiểu của hàm> là kiểu dữ liệu mà hàm trả về, <tên hàm> là tên mà chúng ta muốn đặt cho hàm (tên này được dùng để gọi hàm), danh sách tham số là danh sách các tham biến và kiểu của chúng được sử dụng với hàm. Ví dụ:

int max(int a, int b);

Thường các nguyên mẫu hàm được đặt trong các file .h mà người ta gọi là các file header và để gọi tới một hàm chúng ta cần có chỉ thị #include file header tương ứng với hàm mà chúng ta định sử dụng. Khi đó trong quá trình biên dịch trình biên dịch sẽ tự tìm các hàm chuẩn (được coi là một phần cơ bản của ngôn ngữ) còn với các hàm người dùng định nghĩa chúng ta cần chỉ rõ đường dẫn tới các file chứa phần cài đặt của hàm (thường là file .cpp hoặc một file thư viện tự tạo nào đó).

2.1.2. Định nghĩa hàm

Một hàm sau khi khai báo nguyên mẫu cần phải được xây dựng tường minh theo cấu trúc sau:

<kiểu của hàm> <tên hàm>(<danh sách tham số>)

{

Nội dung của hàm

[return biểu thức;]

}

2.1.3. Hàm và các biến

Các biến nằm trong hàm cũng như các biến là tham số được truyền cùng với một hàm được gọi là các biến cục bộ của một hàm. Các biến nằm ngoài các hàm được gọi là các biến toàn cục.

2.2. Truyền tham số

Việc truyền tham số cho một hàm có thể được tiến hành theo 3 hình thức khác nhau: Truyền theo giá trị, Truyền theo địa chỉ, Truyền theo tham chiếu.

Việc thay đổi các biến ngoài cũng có thể được thực hiện bằng cách truyền theo tham chiếu. So với cách truyền theo địa chỉ truyền theo tham số an toàn hơn và do đó cũng kém linh hoạt hơn.

Ví dụ: void swap(int &a, int &b);

2.3. Chồng hàm (overload) và tham số mặc định của hàm

Chồng hàm (overload): C++ cho phép lập trình viên có khả năng viết các hàm có tên giống nhau, khả năng này được gọi là chồng hàm (overload hoặc polymorphism function mean many formed).Ví dụ chúng ta có thể có các hàm như sau:

int myFunction(int);

int myFunction(int, int);

int myFunction(int, int, int);

Các hàm overload cần thoả mãn một điều kiện là danh sách các tham số của chúng phải khác nhau (về số lượng tham số và kiểu tham số). Kiểu các hàm overload có thể giống nhau hoặc khác nhau. Danh sách kiểu các tham số của một hàm được gọi là ***chữ ký*** (signature) của hàm đó.

Có sự tương tự khi sử dụng chồng hàm và các tham số mặc định và sự lựa chọn sử dụng tuỳ thuộc vào kinh nghiệm của lập trình viên. Với các hàm lớn và phức tạp chúng ta nên sử dụng chồng hàm, ngoài ra việc sử dụng các hàm chồng nhau cũng làm cho chương trình sáng sủa và dễ gỡ lỗi hơn.

Chú ý là không thể overload các hàm static.

Các tham số mặc định: Các biến được truyền làm tham số khi thực hiện gọi một hàm phải có kiểu đúng như nó đã được khai báo trong phần prototype của hàm. Chỉ có một trường hợp khác đó là khi chúng ta khai báo hàm với tham số có giá trị mặc định ví dụ:

int myFunction(int x=10);

Khi đó khi chúng ta thực hiện gọi hàm và không truyền tham số, giá trị 10 sẽ được dùng trong thân hàm. Vì trong prototype không cần tên biến nên chúng ta có thể thực hiện khai báo như sau:

int myFunction(int =10);

Trong phần cài đặt của hàm chúng ta vẫn tiến hành bình thường như các hàm khác:

```
int myFunction(int x){
…
}
```

Bất kỳ một tham số nào cũng có thể gán các giá trị mặc định chỉ có một hạn chế: nếu một tham số nào đó không được gán các giá trị mặc định thì các tham số đứng trước nó cũng không thể sử dụng các giá trị mặc định. Ví dụ với hàm:

int myFunction(int p1, int p2, int p3);

Nếu p3 không được gán các giá trị mặc định thì cũng không thể gán cho p2 các giá trị mặc định.

Các giá trị mặc định của tham số hàm thường được sử dụng trong các hàm cấu tử của các lớp.

2.4. Hàm inline

Hàm inline: Các hàm inline được xác định bằng từ khóa inline. Ví dụ: inline myFunction(int);

Khi chúng ta sử các hàm trong một chương trình C hoặc C++ thường thì phần thân hàm sau khi được biên dịch sẽ là một tập các lệnh máy. Mỗi khi chương trình gọi tới hàm, đoạn mã của hàm sẽ được nạp vào stack để thực hiện sau đó trả về 1 giá trị nào đó nếu có và thân hàm được loại khỏi stack thực hiện của chương trình. Nếu hàm được gọi 10 lần sẽ có 10 lệnh nhảy tương ứng với 10 lần nạp thân hàm để thực hiện. Với chỉ thị inline chúng ta muốn gợi ý cho trình biên dịch là thay vì nạp thân hàm như bình thường hãy chèn đoạn mã của hàm vào đúng chỗ mà nó được gọi tới trong chương trình. Điều này rõ ràng làm cho chương trình thực hiện nhanh hơn bình thường. Tuy nhiên inline chỉ là một gợi ý và không phải bao giờ cũng được thực hiện. Với các hàm phức tạp (chẳng hạn như có vòng lặp) thì không nên dùng inline. Các hàm inline do đó thường rất gắn chẳng hạn như các hàm chỉ thực hiện một vài thao tác khởi tạo các biến (các hàm cấu tử của các lớp). Với các lớp khi khai báo các hàm inline chúng ta có thể không cần dùng từ khóa inline mà thực hiện cài đặt ngay sau khi khai báo là đủ.

Hàm đệ qui: Đệ qui là một cơ chế cho phép một hàm có thể gọi tới chính nó. Kỹ thuật đệ qui thường gắn với các vấn đề mang tính đệ qui hoặc được xác định đệ qui. Để giải quyết các bài toán có các chu trình lồng nhau người ta thường dùng đệ qui. Ví dụ như bài toán tính giai thừa, bài toán sinh các hoán vị của n phần tử

Sử dụng từ khóa const: Đôi khi chúng ta muốn truyền một tham số theo địa chỉ nhưng không muốn thay đổi tham số đó, để tránh các lỗi có thể xảy ra chúng ta có thể sử dụng từ khóa const. Khi đó nếu trong thân hàm chúng ta vô ý thay đổi nội dung của biến trình biên dịch sẽ báo lỗi. Ngoài ra việc sử dụng từ khóa const còn mang nhiều ý nghĩa khác liên quan tới các phương thức của lớp (chúng ta sẽ học trong chương 5).

Bài tập

1. Viết hàm tính tổng các ước của một số nguyên n
2. Viết hàm kiểm tra một số nguyên dương n có là số nguyên tố không
3. Viết hàm kiểm tra một số nguyên dương n có là số hoàn hảo không
4. Viết hàm tìm bội số chung nhỏ nhất của hai số nguyên dương n
5. Viết hàm tính chu vi và diện tích tam giác khi biết ba cạnh a, b, c

CHƯƠNG 3. KÊNH NHẬP XUẤT

3.1. Tổng quan về các luồng vào ra của C++

Để thuận tiện cho việc vào ra dữ liệu trong các chương trình C++ người ta đã đưa vào thư viện chuẩn iostream. Việc không coi các thao tác vào ra dữ liệu là một phần cơ bản của ngôn ngữ và kiểm soát chúng trong các thư viện làm cho ngôn ngữ có tính độc lập về nền tảng cao. Một chương trình viết bằng C++ trên một hệ thống nền này có thể biên dịch lại và chạy tốt trên một hệ thống nền khác mà không cần thay đổi mã nguồn của chương trình. Các nhà cung cấp trình biên dịch chỉ việc cung cấp đúng thư viện tương thích với hệ thống và mọi thứ thế là ổn ít nhất là trên lý thuyết.

Chú ý: Thư viện là một tập các file OBJ có thể liên kết với chương trình của chúng ta khi biên dịch để cung cấp thêm một số chức năng (qua các hàm, hằng, biến được định nghĩa trong chúng). Đây là dạng cơ bản nhất của việc sử dụng lại mã chương trình.

Các lớp iostream coi luồng dữ liệu từ một chương trình tới màn hình như là một dòng (stream) dữ liệu gồm các byte (các ký tự) nối tiếp nhau. Nếu như đích của dòng này là một file hoặc màn hình thì nguồn thường là một phần nào đó trong chương trình. Hoặc có thể là dữ liệu được nhập vào từ bàn phím, các file và được rót vào các biến dùng để chứa dữ liệu trong chương trình.

Một trong các mục đích chính của các dòng là bao gói các vấn đề trong việc lấy và kết xuất dữ liệu ra file hay ra màn hình. Khi một dòng được tạo ra chương trình sẽ làm việc với dòng đó và dòng sẽ đảm nhiệm tất cả các công việc chi tiết cụ thể khác (làm việc với các file và việc nhập dữ liệu từ bàn phím).

Bộ đệm: Việc ghi dữ liệu lên đĩa là một thao tác tương đối đắt đỏ (về thời gian và tài nguyên hệ thống). Việc ghi và đọc dữ liệu từ các file trên đĩa chiếm rất nhiều thời gian và thường thì các chương trình sẽ bị chậm lại do các thao tác đọc và ghi dữ liệu trực tiếp lên đĩa cứng. Để giải quyết vấn đề này các luồng được cung cấp cơ chế sử dụng đệm. Dữ liệu được ghi ra luồng nhưng không được ghi ra đĩa ngay lập tức, thay vào đó bộ đệm của luồng sẽ được làm đầy từ từ và khi đầy dữ liệu nó sẽ thực hiện ghi tất cả lên đĩa một lần.

Điều này giống như một chiếc bình đựng nước có hai van, một van trên và một van dưới. Nước được đổ vào bình từ van trên, trong quá trình đổ nước vào bình van dưới được khóa kín, chỉ khi nào nước trong bình đã đầy thì van dưới mới mở và nước chảy ra khỏi bình. Việc thực hiện thao tác cho phép nước chảy ra khỏi bình mà không cần chờ cho tới khi nước đầy bình được gọi là “flush the buffer”.

3.2. Các luồng và các bộ đệm

C++ thực hiện cài đặt các luồng và các bộ đệm theo cách nhìn hướng đối tượng:

Lớp streambuf quản lý bộ đệm và các hàm thành viên của nó cho phép thực hiện các thao tác quản lý bộ đệm: fill, empty, flush.

Lớp ios là lớp cơ sở của các luồng vào ra, nó có một đối tượng streambuf trong vai trò của một biến thành viên.

Các lớp istream và ostream kế thừa từ lớp ios và cụ thể hóa các thao tác vào ra tương ứng.

Lớp iostream kế thừa từ hai lớp istream và ostream và có các phương thức vào ra để thực hiện kết xuất dữ liệu ra màn hình.

Lớp fstream cung cấp các thao tác vào ra với các file.

3.3. Các đối tượng vào ra chuẩn

Thư viện iostream là một thư viện chuẩn được trình biên dịch tự động thêm vào mỗi chương trình nên để sử dụng nó chúng ta chỉ cần có chỉ thị include file header iostream vào chương trình. Khi đó tự động có 4 đối tượng được định nghĩa và chúng ta có thể sử dụng chúng cho tất cả các thao tác vào ra cần thiết.

cin: quản lý việc vào dữ liệu chuẩn hay chính là bàn phím

cout: quản lý kết xuất dữ liệu chuẩn hay chính là màn hình

cer: quản lý việc kết xuất (không có bộ đệm) các thông báo lỗi ra thiết bị báo lỗi chuẩn (là màn hình). Vì không có cơ chế đệm nên dữ liệu được kết xuất ra cer sẽ được thực hiện ngay lập tức.

clo: quản lý việc kết xuất (có bộ đệm) các thông báo lỗi ra thiết bị báo lỗi chuẩn (là màn hình). Thường được tái định hướng vào một file log nào đó trên đĩa.

3.3.1. Nhập dữ liệu với cin

cin là môt đối tượng toàn cục chịu trách nhiệm nhập dữ liệu cho chương trình. Để sử dụng cin cần có chỉ thị tiền xử lý include file header iostream. Toán tử >> được dùng với đối tượng cin để nhập dữ liệu cho một biến nào đó của chương trình.

Toán tử >> là một toán tử được overload và kết quả của việc sử dụng toán tử này là ghi tất cả nội dung trong bộ đệm của cin vào một biến cục bộ nào đó trong chương trình. Do >> là một toán tử được overload của cin nên nó có thể được dùng để nhập dữ liệu cho rất nhiều biến có kiểu khác nhau ví dụ:

int someVar; cin >> someVar;

Các xâu (strings)

cin cũng có thể làm việc với các biến xâu, hay các mảng ký tự, ví dụ:

char stdName[255]; cin >> stdName;

Chú ý là chúng ta có thể thực hiện nhập nhiều tham số một lúc ví dụ:

cin >> intVar >> floatVar;

Các hàm thành viên khác của cin

Ngoài việc overload toán tử >> cin còn có rất nhiều hàm thành viên khác có thể được sử dụng để nhập dữ liệu.

Hàm get

Hàm get có thể sử dụng để nhập các ký tự đơn, khi đó chúng ta gọi tới hàm get() mà không cần có đối số, gía trị trả về là ký tự được nhập vào ví dụ:

#include <iostream>

```
int main() {
   char ch;
   while ( (ch = cin.get()) != EOF)  {     cout << "ch: " << ch << endl;          }
  cout << "\nDone!\n";
  return 0; }
```

Chương trình trên sẽ cho phép người dùng nhập vào các xâu có độ dài bất kỳ và in ra lần lượt các ký tự của xâu đó cho tới khi gặp ký tự điều khiển Ctrl-D hoặc Ctrl-Z.

Chú ý là không phải tất cả các cài đặt của istream đều hỗ trợ phiên bản này của hàm get().

Có thể gọi tới hàm get() để nhập các ký tự bằng cách truyền vào một biến kiểu char ví dụ:

```
char a, b;
cin.get(a).get(b);
```

Ngoài cách nhập xâu bằng cách sử dụng hàm get() chúng ta có thể dùng hàm getline(). Hàm getline hoạt động tương tự như hàm get() chỉ trừ một điều là ký tự kết thúc sẽ được loại khỏi bộ đệm trong trường hợp nó được nhập trước khi đầy xâu.

Sử dụng hàm ignore: Đôi khi chúng ta muốn bỏ qua tất cả các ký tự còn lại của một dòng dữ liệu nào đó cho tới khi gặp ký tự kết thúc dìng (EOL) hoặc ký tự kết thúc file (EOF), hàm ignore của đối tượng cin nhằm phục vụ cho mục đích này. Hàm này có 2 tham số, tham số thứ nhất là số tối đa các ký tự sẽ bỏ qua cho tới khi gặp ký tự kết thúc được chỉ định bởi tham số thứ hai. Chẳng hạn với câu lệnh cin.ignore(80, ’\n’) thì tối đa 80 ký tự sẽ bị loại bỏ cho tới khi gặp ký tự xuống dòng, ký tự này sẽ được loại bỏ trước khi hàm ignore kết thúc công việc của nó. Ví dụ:

```
#include <iostream>
int main()       {
char stringOne[255];
char stringTwo[255];
cout << "Nhập xâu thứ nhất:"; cin.get(stringOne,255);
cout << "Xâu thứ nhất" << stringOne << endl;
cout << "Nhập xâu thứ hai: ";  cin.getline(stringTwo,255);
  cout << "Xâu thứ hai: " << stringTwo << endl;
  cout << "\n\nNhập lại...\n";
  cout << "Nhập xâu thứ nhất: "; cin.get(stringOne,255);
  cout << "Xâu thứ nhất: " << stringOne<< endl;
  cin.ignore(255,'\n');
  cout << "Nhập xâu thứ hai: "; cin.getline(stringTwo,255);
  cout << "Xâu thứ hai: " << stringTwo<< endl;
  return 0;       }
```

cin có hai phương thức khác là peek và putback với mục đích làm cho công việc trở nên dễ dàng hơn. Hàm peek() cho ta biết ký tự tiếp theo nhưng không nhập chúng còn hàm putback() cho phép chèn một ký tự vào dòng input. Ví dụ:

```
while ( cin.get(ch) ){
        if (ch == '!') cin.putback('$');
        else     cout << ch;
  while (cin.peek() == '#') cin.ignore(1,'#');
}
```

Các hàm này thường được dùng để thực hiện phân tích các xâu hay các dữ liệu khác chẳng hạn trong các chương trình phân tích cú pháp của ngôn ngữ chẳng hạn.

3.3.2. Kết xuất dữ liệu với cout

Chúng ta đã từng sử dụng đối tượng cout cho việc kết xuất dữ liệu ra màn hình, ngoài ra chúng ta cũng có thể sử dụng cout để định dạng dữ liệu, căn lề các dòng kết xuất dữ liệu và ghi dữ liệu kiểu số dưới dạng số thập phân hay hệ hexa.

Xóa bộ đệm ouput: Việc xóa bộ đệm output được thực hiện khi chúng ta gọi tới hàm endl. Hàm này thực chất là gọi tới hàm flush() của đối tượng cout. Chúng ta có thể gọi trực tiếp tới hàm này:

```
cout << flush;
```

Các hàm liên quan

Tương tự như cin có các hàm get() và getline() cout có các hàm put() và write() phục vụ cho việc kết xuất dữ liệu. Hàm put() được dùng để ghi một ký tự ra thiết bị output và cũng như hàm get() của cin chúng ta có thể dùng hàm này liên tiếp:

```
cout.put('a').put('e');
```

Hàm write làm việc giống hệt như toán tử chèn chỉ trừ một điều là nó các hai tham số: tham số thứ nhất là con trỏ xâu và tham số thứ hai là số ký tự sẽ in ra, ví dụ:

```
char str[] = "no pain no gain"; cout.write(str,3);
```

Các chỉ thị định dạng, các cờ và các thao tác với cout

Dòng output có rất nhiều cờ được sử dụng vào các mục đích khác nhau chẳng hạn như quản lý cơ số của các biến sẽ được kết xuất ra màn hình, kích thước của các trường… Mỗi cờ trạng thái là một byte có các bit được gán cho các ý nghĩa khác nhau, các cờ này có thể được đặt các giá trị khác nhau bằng cách sử dụng các hàm.

Sử dụng hàm width của đối tượng cout

Độ rộng mặc định khi in ra các biến là vừa đủ để in dữ liệu của biến đó, điều này đôi khi làm cho output nhận được không đẹp về mặt thẩm mỹ. Để thay đổi điều này chúng ta có thể dùng hàm width, ví dụ:

```
cout.width(5);
```

Bình thường cout lấp các chỗ trống khi in ra một biến nào đó (với độ rộng được thiết lập bằng hàm width) bằng các dấu trống, nhưng chúng ta có thể thay đổi điều này bằng cách gọi tới hàm fill, ví dụ:

cout.fill('*');

Thiết lập các cờ: Các đối tượng iostream quản lý trạng thái của chúng bằng cách sử dụng các cờ. Chúng ta có thể thiết lập các cờ này bằng cách sử dụng hàm setf() với tham số là một hằng kiểu liệt kê đã được định nghĩa trước. Chẳng hạn với một biến kiểu float có giá trị là 20.000, nếu chúng ta sử dụng toán tử >> để in ra màn hình kết quả nhận được sẽ là 20, nếu chúng ta thiết lập cờ showpoint kết quả sẽ là 20.000.

Sau đây là một số cờ thường dùng:

showpos	dấu của các biến kiểu số
hex	In ra số dưới dạng hexa
dec	In ra số dưới dạng cơ số 10
oct	In ra số dưới dạng cơ số 8
left	Căn lề bên trái
right	Căn lề bên phải
internal	Căn lề giữa
precision	Hàm thiết lập độ chính xác của các biến thực: cout.precision(2);

Ngoài ra còn phải kể đến hàm setw() cũng có thể được dùng thay cho hàm width.

Chú ý là các hàm trên đều cần có chỉ thị include file header iomanip.h.

Các hàm width, fill và precision đều có một phiên bản không có tham số cho phép đọc các giá trị được thiết lập hiện tại (mặc định).

3.3.3 Các dòng vào ra và hàm printf

Hầu hết các cài đặt của C++ đều cung cấp các thư viện C chuẩn, trong đó bao gồm lệnh printf(). Mặc dù về mặt nào đó hàm prinf dễ sử dụng hơn so với các dòng của C++ nhưng nó không thể theo kịp được các dòng: hàm printf không có đảm bảo về kiểu dữ liệu do đó khi chúng ta in ra một ký tự lại có thể là một số nguyên và ngược lại, hàm printf cũng không hỗ trợ các lớp do đó không thể overload để làm việc với các lớp.

Ngoài ra hàm printf cũng thực hiện việc định dạng dữ liệu dễ dàng hơn, ví dụ:

printf("%15.5f", tf);

Có rất nhiều lập trình viên C++ có xu hướng thích dùng hàm printf() nên cần phải chú ý kỹ tới hàm này.

3.4. Vào ra dữ liệu với các file

Các dòng cung cấp cho chúng ta một cách nhất quán để làm việc với dữ liệu được nhập vào từ bàn phím cũng như dữ liệu được nhập vào từ các file. Để làm việc với các file chúng ta tạo ra các đối tượng ofstream và ifstream.

ofstream

Các đối tượng cụ thể mà chúng ta dùng để đọc dữ liệu ra hoặc ghi dữ liệu vào được gọi là các đối tượng ofstream. Chúng được kế thừa từ các đối tượng iostream.

Để làm việc với một file trước hết chúng ta cần tạo ra một đối tượng ofstream, sau đó gắn nó với một file cụ thể trên đĩa, và để tạo ra một đối tượng ofstream chúng ta cần include file fstream.h.

Các trạng thái điều kiện

Các đối tượng iostream quản lý các cờ báo hiệu trạng thái sau khi chúng ta thực hiện các thao tác input và output chúng ta có thể kiểm tra các cờ này bằng cách sử dụng các hàm Boolean chẳng hạn như eof(), bad(), fail(), good(). Hàm bad() cho giá trị TRUE nếu một thao tác là không hợp lệ, hàm fail() cho giá trị TRUE nếu như hàm bad() cho giá trị TRUE hoặc một thao tác nào đó thất bại. Cuối cùng hàm good() cho giá trị TRUE khi và chỉ khi tất cả 3 hàm trên đều trả về FALSE.

Mở file

Để mở một file cho việc kết xuất dữ liệu chúng ta tạo ra một đối tượng ofstream:

ofstream fout(“out.txt”);

và để mở file nhập dữ liệu cũng tương tự:

ifstream fin(“inp.txt”);

Sau khi tạo ra các đối tượng này chúng ta có thể dùng chúng giống như các thao tác vẫn được thực hiện với cout và cin nhưng cần chú ý có hàm close() trước khi kết thúc chương trình.

Ví dụ:

ifstream fin(“data.txt”);

while(fin.get(ch)) cout << ch;

fin.close();

Thay đổi thuộc tính mặc định khi mở file

Khi chúng ta mở một file để kết xuất dữ liệu qua một đối tượng ofstream, nếu file đó đã tồn tại nội dung của nó sẽ bị xóa bỏ còn nếu nó không tồn tại thì file mới sẽ được tạo ra. Nếu chúng ta muốn thay đổi các hành động mặc định này có thể truyền thêm một biến tường minh vào cấu tử của lớp ofstream.

Các tham số hợp lệ có thể là:

ios::app – append, mở rộng nội dung một file nếu nó đã có sẵn.

ios:ate -- đặt vị trí vào cuối file nhưng có thể ghi lên bất cứ vị trí nào trong file

ios::trunc -- mặc định

ios::nocreate -- nếu file không có sẵn thao tác mở file thất bại

ios::noreplace -- Nếu file đã có sẵn thao tác mở file thất bại.

Chú ý: nên kiểm tra khi thực hiện mở một file bất kỳ. Nên sử dụng lại các đối tượng ifstream và ofstream bằng hàm open().

File text và file nhị phân

Một số hệ điều hành chẳng hạn DOS phân biệt các file nhị phân và các file text. Các file text lưu trữ dữ liệu dưới dạng text chẳng hạn một số sẽ được lưu thành một xâu, việc lưu trữ

theo kiểu này có nhiều bất lợi song chúng ta có thể xem nội dung file bằng các chương trình rất đơn giản.

Để giúp phân biệt giữa các file text và nhị phân C++ cung cấp cờ ios::binary. Trên một số hệ thống cờ này thường được bỏ qua vì thường thì dữ liệu được ghi dưới dạng nhị phân.

Các file nhị phân không chỉ lưu các số và ký tự chúng có thể được sử dụng để lưu các cấu trúc. Để ghi một biến cấu trúc lên một file nhị phân chúng ta dùng hàm write(), ví dụ:

fout.write((char*) &Bear,sizeof Bear);

Để thực hiện việc đọc ngược lại chúng ta dùng hàm read.

fin.read((char*) &BearTwo, sizeof BearTwo);

Làm việc với máy in

Làm việc với máy in tương đối giống với làm việc với các file:

ofstream prn("PRN");

Bài tập

1. Viết chương trình ghi một mảng n số nguyên vào file du_lieu.txt
2. Viết chương trình đọc dữ liệu từ file du_lieu.txt và in kết quả ra màn hình

CHƯƠNG 4. ĐỐI TƯỢNG VÀ LỚP.

4.1. Định nghĩa đối tượng, lớp

Rất nhiều các đối tượng trong thế giới thực có hai đặc tính sau: chúng có một trạng thái (state) (các thuộc tính có thể thay đổi) và các năng lực (công việc mà chúng có thể thực hiện).

Đối tượng thực = Trạng thái (các thuộc tính)+ Các năng lực (hành vi)

Đối tượng lập trình = Dữ liệu + Các hàm

Kết quả của việc trừu tượng hóa các đối tượng của thế giới thực thành các đối tượng lập trình là sự kết hợp giữa dữ liệu và các hàm.

Lớp là một kiểu dữ liệu mới được dùng để định nghĩa các đối tượng. Một lớp có vai trò như một kế hoạch hay một bản mẫu. Nó chỉ rõ dữ liệu nào (các thuộc tính - trạng thái) và các hàm (các năng lực) nào sẽ thuộc về các đối tượng của lớp đó. Việc viết hay tạo ra một lớp mới không sinh ra bất cứ một đối tượng nào trong chương trình.

Một lớp là sự trừu tượng hóa, tổng quát hóa các đối tượng có các thuộc tính giống nhau và các đối tượng là thể nghiệm của các lớp.

4.2. Khai báo lớp, đối tượng

4.2.1. Khai báo lớp

Để khai báo một lớp, ta sử dụng từ khoá class như sau:

class tên_lớp{

// Khai báo các thành phần dữ liệu (thuộc tính)

// Khai báo các phương thức (hàm)

};

Chi tiết hơn ta có khai báo lớp như sau :

class tên_lớp{

private :

// Khai báo các thành phần dữ liệu (thuộc tính) riêng

// Khai báo các phương thức (hàm) riêng

protected:

// Khai báo các thành phần dữ liệu (thuộc tính) được bảo vệ

// Khai báo các phương thức (hàm) được bảo vệ

public:

// Khai báo các thành phần dữ liệu (thuộc tính) chung

// Khai báo các phương thức (hàm) chung

};

Chú ý: Việc khai báo một lớp không chiếm giữ bộ nhớ, chỉ các đối tượng của lớp mới thực sự chiếm giữ bộ nhớ.

Thuộc tính của lớp có thể là các biến, mảng, con trỏ có kiểu chuẩn (int, float, char, char*, long,...) hoặc kiểu ngoài chuẩn đã định nghĩa trước (cấu trúc, hợp, lớp,...). Thuộc tính

của lớp không thể có kiểu của chính lớp đó, nhưng có thể là con trỏ của lớp này, ví dụ:

```
class A{
A x;  //Không cho phép, vì x có kiểu lớp A
A* p ; //Cho phép , vì p là con trỏ kiểu lớp A
} ;
```

4.2.2. Khai báo các thành phần của lớp (thuộc tính và phương thức)

a. Các từ khóa private và public, protected

Khi khai báo các thành phần dữ liệu và phương thức có thể dùng các từ khoá private, *protected* và public để quy định phạm vi sử dụng của các thành phần này.

- Từ khóa private: qui định các thành phần (được khai báo với từ khóa này) chỉ được sử dụng bên trong lớp (trong thân các phương thức của lớp) hoặc các hàm bạn. Các hàm bên ngoài lớp (không phải là phương thức của lớp) không được phép sử dụng các thành phần này. Đặc trưng này thể hiện tính che giấu thông tin trong nội bộ của lớp, để đến được các thông tin này cần phải thông qua chính các thành phần hàm của lớp đó. Do vậy thông tin có tính toàn vẹn cao và việc xử lý thông tin (dữ liệu) này mang tính thống nhất hơn và hầu như dữ liệu trong các lớp đều được khai báo với từ khóa này.
- Từ khóa public: các thành phần được khai báo với từ khóa public được phép sử dụng ở cả bên trong và bên ngoài lớp, điều này cho phép trong chương trình có thể sử dụng các hàm này để truy nhập đến dữ liệu của lớp. Hiển nhiên nếu các thành phần dữ liệu đã khai báo là privte thì các thành phần hàm phải có ít nhất một vài hàm được khai báo dạng public để chương trình có thể truy cập được, nếu không toàn bộ lớp sẽ bị đóng kín và điều này không giúp gì cho chương trình. Do vậy cách khai báo lớp tương đối phổ biến là các thành phần dữ liệu được ở dạng private và thành phần hàm dưới dạng public. Nếu không quy định cụ thể (không dùng các từ khoá private và public) thì C++ hiểu đó là private.
- Từ khóa protected : các thành phần được khai báo với từ khóa *protected* được sử dụng từ bên trong lớp, các hàm bạn, các lớp thùa kế.

b. Các thành phần dữ liệu (thuộc tính)

Được khai báo như khai báo các thành phần trong kiểu cấu trúc hay hợp. Bình thường các thành phần này được khai báo là private để bảo đảm tính giấu kín, bảo vệ an toàn dữ liệu của lớp không cho phép các hàm bên ngoài xâm nhập vào các dữ liệu này.

c. Các phương thức (hàm thành viên)

Thường khai báo là public để chúng có thể được gọi tới (sử dụng) từ các hàm khác trong chương trình.

Các phương thức có thể được khai báo và định nghĩa bên trong lớp hoặc chỉ khai báo bên trong còn định nghĩa cụ thể của phương thức có thể được viết bên ngoài. Thông thường, các phương thức ngắn được viết (định nghĩa) bên trong lớp, còn các phương thức dài thì viết bên ngoài lớp.

Một phương thức bất kỳ của một lớp, có thể sử dụng bất kỳ thành phần (thuộc tính và phương thức) nào của lớp đó và bất kỳ hàm nào khác trong chương trình (vì phạm vi sử dụng của hàm là toàn chương trình).

Giá trị trả về của phương thức có thể có kiểu bất kỳ (chuẩn và ngoài chuẩn)

Ví dụ 1: Lớp point định nghĩa các điểm ảnh trong một chương trình đồ họa.

Mỗi điểm phải có hai trạng thái là hoành độ và tung độ, chúng ta có thể dùng hai biến kiểu int để biểu diễn chúng.

Trong chương trình của chúng ta các điểm phải có các khả năng sau:

Các điểm có thể di chuyển trên màn hình: điều này được thực hiện qua hàm move

Các điểm có thể in ra các tọa độ của chúng lên màn hình: hàm print

Các điểm có thể trả lời câu hỏi xem chúng có đang ở vị trí gốc tạo độ hay không: hàm isZero()

```
class Point{ // khai báo lớp Point
int x, y; // hoành độ và tung độ
public:
void move(int,int);
void print();
bool isZero();
};
```

Các hàm và các biến trong một lớp được gọi là các thành viên của lớp. Trong ví dụ này chỉ có các nguyên mẫu hàm được đặt trong khai báo lớp, phần cài đặt sẽ được đặt ở các phần khác (có thể trong một file khác) của chương trình.

Nếu như phần định nghĩa hay cài đặt của một hàm được đặt ngay trong phần khai báo của lớp thì hàm đó được gọi là một hàm inline (macro).

Một số chú ý về hàm inline:

Chúng ta có thể sử dụng từ khóa inline để chỉ định một hàm là inline:

```
inline void Point::move(int newX, int newY){ }
```

Không phải lúc nào hàm inline cũng được thực hiện: có 2 trường hợp mà trình biên dịch sẽ từ chối chấp nhận một hàm là hàm inline trường hợp thứ nhất là khi hàm đó quá phức tạp, trường hợp thứ hai là nó gọi tới một hàm có địa chỉ không xác định, ví dụ:

```
class Forward {
  int i;
public:
  Forward() : i(0) {}
  // gọi tới hàm chưa được khai báo:
  int f() const { return g() + 1; }
  int g() const { return i; }
};
int main() {
  Forward frwd;
  frwd.f();
```

```
} ///:~
```

Các hàm inline thường thấy chính là các cấu tử và các huỷ tử, trong thân các hàm này chúng ta hay thực hiện một số thao tác khởi tạo và dọn dẹp ẩn.

```
// hàm di chuyển tới vị trí mới
void Point::move(int new_x, int new_y){
x = new_x;
y = new_y;
}
// in tọa độ ra màn hình
void Point::print(){
  cout << "X= " << x << ", Y= " << y << endl;
}
// kiểm tra xem có trùng với gốc tọa độ không (0,0)
bool Point::is_zero(){
  return (x == 0) && (y == 0); // if x=0 AND y=0 returns true
}
```

4.2.3. Đối tượng

Các lớp có thể được dùng để khai báo các biến giống như các kiểu dữ liệu cơ bản khác. Các biến này được gọi là các đối tượng. Cách thức khai báo đối tượng giống khai báo biến thông thường.

Cú pháp: Tên_lớp đối_tượng;

Ví dụ:

```
int main(){
Point point1, point2; // khai báo 2 đối tượng: point1 and point2
point1.move(100,50); // di chuyển tới điểm (100,50)
point1.print(); // in tọa độ ra màn hình
point1.move(20,65); // point1 di chuyển tới điểm (20,65)
point1.print(); // in tọa độ ra màn hình
if(point1.is_zero()) // kiểm tra xem có trùng với gốc tọa độ không
cout << "point1 is now on zero point(0,0)" << endl;
else
cout << "point1 is NOT on zero point(0,0)" << endl;
point2.move(0,0); // point2 moves to (0,0)
if(point2.is_zero()) // is point2 on (0,0)?
cout << "point2 is now on zero point(0,0)" << endl;
else
```

```
cout << "point2 is NOT on zero point(0,0)" << endl;
return 0;
}
```

Nhắc lại một số khái niệm và thuật ngữ:

Lớp: một lớp là kết quả của việc gom dữ liệu và các hàm. Khái niệm lớp rất giống với khái niệm cấu trúc được sử dụng trong C hay khái niệm bản ghi được sử dụng trong Pascal, nó là một kiểu dữ liệu được dùng để tạo ra các biến có thể được dùng trong một chương trình nào đó.

Đối tượng: Một đối tượng là một thể nghiệm của một lớp, điều này tương tự như đối với một biến được định nghĩa là một thể hiện của một kiểu dữ liệu, nó mang tính cụ thể và xác định.

Phương thức (Method) (member function) là một hàm được khai báo trong một lớp.

Thông điệp: đây là khái niệm tương đối giống với việc gọi tới một hàm tuy nhiên có sự khác nhau về cơ bản ở hai điểm sau đây:

Một thông điệp phải được gửi tới cho một đối tượng nào đó dù có thể là tường minh hay cụ thể gọi là receiver.

Hành động được thực hiện để đáp lại thông điệp được quyết định bởi receiver (đối tượng nhận thông điệp), các receiver có thể có các hành động khác nhau đối với cùng một thông điệp.

Ví dụ:

```
anObject.doSomething(….);
```

Nhiều người mới học C++ cho rằng hai khái niệm gọi hàm và gửi thông điệp trong các chương trình là giống nhau hoàn toàn.

Kết luận:

Sau phần này chúng ta cần chú ý các điểm sau:

Các chương trình được xây dựng theo kiểu hướng đối tượng bao gồm các đối tượng, và việc này thường được bắt đầu bằng việc xây dựng và thiết kế các lớp. Thay vì các lời gọi hàm trong một chương trình hướng đối tượng chúng ta thực hiện gửi các thông điệp tới cho các đối tượng để bảo chúng thực hiện một công việc nào đó.

Ví dụ các con trỏ trỏ tới các đối tượng:

```
int main(){
Point p;
Point *pp1 = new Point; // cấp phát bộ nhớ cho 1 đối tượng, pp1 trỏ tới
Point *pp2 = new Point; // cấp phát bộ nhớ cho 1 đối tượng, pp2 trỏ tới
pp1->move(50,50);
pp1->print();
pp2->move(100,150);
if(pp2->is_zero()) cout << " Object pointed by pp2 is on zero." << endl;
```

```
else cout << " Object pointed by pp2 is NOT on zero." << endl;
delete pp1; // giải phóng bộ nhớ
delete pp2;
return 0;
}
```

Ví dụ: Mảng các đối tượng (có thể là mảng tĩnh hoặc mảng động):

```
int main(){
Point array[10]; // khai báo 1 mảng có 10 phần tử
array[0].move(15,40);
array[1].move(75,35);
for (int i= 0; i < 10; i++) array[i].print();
return 0;
}
```

4.2.4. Sử dụng các đối tượng trong vai trò là tham số của hàm

Các đối tượng nên được truyền và trả về theo kiểu tham chiếu trừ khi có các lý do đặc biệt đòi hỏi chúng ta phải truyền hoặc trả về chúng theo kiểu truyền biến và trả về theo kiểu giá trị. Việc truyền tham biến và giá trị của hàm đặc biệt không hiệu quả khi làm việc với các đối tượng. Chúng ta hãy nhớ lại đối tượng được truyền hoặc trả lại theo giá trị phải được copy vào stack và dữ liệu có thể rất lớn, và do đó có thể làm lãng phí bộ nhớ. Bản thân việc copy này cũng tốn thời gian. Nếu như lớp chứa một cấu tử copy thì trình biên dịch sẽ sử dụng hàm này để copy đối tượng vào stack.

Chúng ta nên truyền tham số theo kiểu tham chiếu vì chúng ta không muốn có các bản copy được tạo ra. Và để ngăn chặn việc hàm thành viên có thể vô tình làm thay đổi đối tượng ban đầu chúng ta sẽ khai báo tham số hình thức có kiểu là hằng tham chiếu (**const reference**).

Ví dụ:

```
ComplexT & ComplexT::add(constComplexT &z){
  ComplexT result;
  result.re = re + z.re;    result.im = im + z.im;
  return result; // Sai các biến cục bộ không thể trả về qua tham chiếu.
}
```

Ví dụ trên có thể sửa lại cho đúng như sau:

```
ComplexT ComplexT::add(constComplexT &z){
  ComplexT result;
  result.re = re + z.re;    result.im = im + z.im;
  return result; // Sai các biến cục bộ không thể trả về qua tham chiếu.
}
```

Tuy nhiên do có một đối tượng tạm thời được tạo ra như vậy các hàm huỷ tử và cấu tử sẽ được gọi đến. Để tránh việc tạo ra một đối tượng tạm thời (để tiết kiệm thời gian và bộ nhớ) người ta có thể làm như sau:

```
ComplexT ComplexT::add(constComplexT &z){
double re_new, im_new;
re_new = re + z.re;        im_new = im + z.im;
  return ComplexT(re_new, im_new);
}
```

Chỉ có đối tượng trả về trên stack là được tạo ra (là thứ luôn cần thiết khi trả về theo giá trị). Đây có thể là một cách tiếp cận tốt hơn: chúng ta sẽ tạo ra và hủy bỏ các phần tử dữ liệu riêng biệt, cách này sẽ nhanh hơn là tạo ra và hủy bỏ cả một đối tượng hoàn chỉnh.

4.2.5. Kiểm soát việc truy cập tới các biến và phương thức của lớp

Chúng ta có thể chia các lập trình viên thành hai nhóm: nhóm tạo các lớp (nhóm gồm những người tạo ra các kiểu dữ liệu mới) và các lập trình viên khách (các lập trình viên sử dụng các kiểu dữ liệu do nhóm thứ nhất tạo ra trong chương trình của họ).

Mục đích của nhóm thứ nhất là xây dựng một lớp bao gồm tất cả các thuộc tính và khả năng cần thiết. Lớp này sẽ cung cấp các hàm giao diện cần thiết, chỉ những gì cần thiết cho các lập trình viên sử dụng chúng và giữ bí mật các phần còn lại.

Mục đích của các lập trình viên khách là tập hợp một hộp công cụ với đầy các lớp để nhanh chóng sử dụng phát triển xây dựng chương trình ứng dụng của mình.

Lý do đầu tiên cho việc kiểm soát truy cập này là đảm bảo các lập trình viên khách không thể thò tay vào những phần mà họ không nên thò tay vào. Phần được ẩn đi là phần cần thiết cho cấu trúc bên trong của lớp và không phải là phần giao diện mà người dùng cần để giải quyết vấn đề của họ.

Lý do thứ hai là nếu như việc kiểm soát truy cập bị ẩn đi thì các lập trình viên khách không thể sử dụng nó, có nghĩa là người tạo ra các lớp có thể thay đổi các phần bị ẩn mà không cần phải lo lắng sẽ ảnh hưởng tới bất kỳ ai.

Sự bảo vệ này cũng ngăn chặn các thay đổi ngoài ý muốn đối với các trạng thái của các đối tượng.

Để phục vụ cho việc kiểm soát truy cập các thành viên của một lớp C++ cung cấp 3 nhãn: public, priavte và protected.

Các thành viên đứng sau một nhãn sẽ được gán nhãn đó cho tới khi nhãn mới xuất hiện.

Các thành viên được gán nhãn private (mặc định) chỉ có thể được truy cập tới bởi các thành viên khác của lớp.

Mục đích chính của các thành viên được gán nhãn public là cung cấp cho các client danh sách các dịch vụ mà lớp đó hỗ trợ hay cung cấp. Tập các thành viên này tạo nên phần giao diện công cộng của một lớp.

Chúng ta có thể thấy rõ qua lớp Point trong ví dụ ở phần đầu, việc truy cập vào biến thành viên chẳng hạn x là bất hợp lệ.

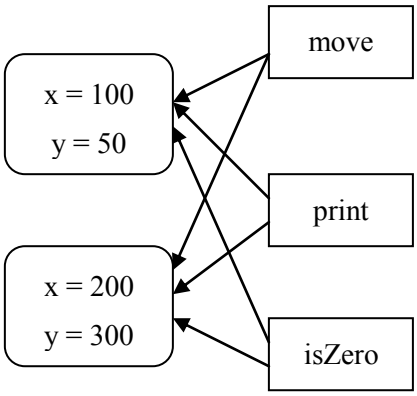
Các cấu trúc trong một chương trình C++ có chế độ truy cập mặc định là public.

Từ khóa protected có ý nghĩa giống hệt ý nghĩa của từ khóa private ngoại trừ một điều là các lớp kế thừa có thể truy cập vào các thành viên protected của lớp cơ sở mà nó kế thừa.

4.2.6. Con trỏ this

Mỗi đối tượng có không gian dữ liệu riêng của nó trong bộ nhớ của máy tính. Khi mỗi đối tượng được định nghĩa, phần bộ nhớ được khởi tạo chỉ dành cho phần lưu dữ liệu của đối tượng đó.

Mã của các hàm thành viên chỉ được tạo ra một lần. Các đối tượng của cùng một lớp sẽ sử dụng chung mã của các hàm thành viên.



Vậy làm thế nào để trình biên dịch có thể đảm bảo được rằng việc tham chiếu này là đúng đắn. Để đảm bảo điều này trình biên dịch duy trì một con trỏ được gọi là con trỏ this. Với mỗi đối tượng trình biên dịch đều sinh ra một con trỏ this gắn với nó. Khi một hàm thành viên được gọi đến, con trỏ this chứa địa chỉ của đối tượng sẽ được sử dụng và chính vì vậy các hàm thành viên sẽ truy cập tới đúng các thành phần dữ liệu của đối tượng thông qua địa chỉ của đối tượng. Chúng ta cũng có thể sử dụng con trỏ this này trong các chương trình một cách rõ ràng, ví dụ:

```
Point *Point::far_away(Point &p){
unsigned long x1 = x*x;
unsigned long y1 = y*y;
unsigned long x2 = p.x * p.x;
unsigned long y2 = p.y * p.y;
if ( (x1+y1) > (x2+y2) ) return this; // trả về địa chỉ của đối tượng
else return &p; // trả về đối tượng đang được tạo ra
}
```

4.2.7. Khai báo các lớp với các file header

Ví dụ:

File stack.h

```
#ifndef Stack_H
```

```
#define Stack_H
class Stack {
// LIFO objects
  public:
    Stack(int MaxStackSize = 10);
    ~Stack() {delete [] stack;}
    bool IsEmpty() const {return top == -1;}
    bool IsFull() const {return top == MaxTop;}
    int top() const;
    void push(const int & x);
    int pop();
  private:
    int top;    // chỉ số phần tử ở đỉnh
    int MaxTop; // giá trị lớp nhất của biến top
    int *stack;   // dữ liệu của stack
};
#endif
file stack.cpp:
Stack::Stack(int MaxStackSize)
{// cấu tử của lớp Stack
  MaxTop = MaxStackSize - 1;
  stack = new int[MaxStackSize];
  top = -1;
}
int Stack::top() const
{// trả về phần tử ở đỉnh
  if (IsEmpty()){
cout << "OutOfBounds"; // stack rỗng
return –1;
      }
  else return stack[top];
}
void Stack::push(const int& x)
{// Add x to stack.
  if (IsFull()) {
cout << "NoMem"; // hết bộ nhớ
```

```
return;
  }
  stack[++top] = x;
}
int Stack::pop()
{// loại bỏ phần tử ở đỉnh
  if (IsEmpty()){
cout << "OutOfBounds"; // delete fails
return –1;
  }
}
```

Việc sử dụng các file header để khai báo các lớp và các hàm nhằm mục đích chính là tách biệt phần giao diện và phần cài đặt của các lớp, các hàm. Như chúng ta đã biết các chương trình hiện nay thường được viết theo nhóm làm việc. Trong nhóm mỗi người được giao viết một phần của chương trình và do đó cần dùng phần chương trình của người khác, và việc sử dụng các file header cũng là một phần giải pháp để đạt được điều đó. Khi một lập trình viên thay đổi các cài đặt bên dưới của một hàm hay một lớp nào đó thì việc thay đổi này không làm ảnh hưởng tới phần chương trình do người khác viết dựa trên các lớp và hàm mà anh ta đưa ra.

Cấu trúc của một file header thường có 3 chỉ thị tiền xử lý chính, chỉ thị đầu tiên là chỉ thị kiểm tra sự tồn tại của một cờ định danh, nó có tác dụng kiểm tra xem nội dung của file đã được include vào một file nào đó hay chưa, nếu rồi thì phần nội dung tiếp theo của file sẽ được bỏ qua. Chỉ thị thứ hai là chỉ thị định nghĩa, xác định một định danh dùng cho việc kiểm tra được thực hiện trong quá trình biên dịch.

Chỉ thị cuối cùng là chỉ thị kết thúc nội dung của file.

Nếu không có các chỉ thị này chúng ta sẽ gặp rắc rối to với vấn đề include các file header trong các chương trình.

Nếu muốn tìm hiểu kỹ hơn về các chỉ thị và cấu trúc của file header chúng ta có thể tham khảo nội dung các file haeder chuẩn được cung cấp với trình biên dịch.

Việc biên dịch chương trình đối với các chương trình sử dụng lớp Stack được thực hiện như sau:

Trong chương trình có sử dụng lớp Stack chúng ta cần include file header stack.h. Để thử xem có lỗi không nhấn tổ hợp phím ALT-C để kiểm tra các lỗi biên dịch. Việc dịch tiếp theo (phần build file .exe) có thể được thực hiện theo hai cách: cách thứ nhất là tạo một project và dịch như các chương trình chỉ có 1 file .cpp mà chúng ta đã biết; cách thứ hai là tạo một file .bat để thực hiện biên dịch, ví dụ với một chương trình gồm các file: main.cpp, stack.h, stack.cpp thì nội dung file build.bat sẽ có thể là:

```
tcc –c main.cpp
tcc –c stack.cpp
```

tcc –e<tên file.exe> main.obj stack.obj

Hoặc gọn hơn nữa là:

tcc –c *.cpp

tcc –e<tên file.exe> *.obj

del *.obj

Hoặc chúng ta cũng có thể thực hiện trực tiếp các lệnh này trên dòng lệnh trong các cửa sổ giả lập DOS.

4.3. Cấu tử, hủy tử

4.3.1. Cấu tử

Như chúng ta đã biết các đối tượng trong mỗi chương trình C++ đều có hai loại thành viên: các dữ liệu thành viên và các hàm thành viên. Các hàm thành viên làm việc dựa trên hai loại dữ liệu: một loại được lấy từ bên ngoài thông qua việc gọi các thông điệp, loại kia chính là các dữ liệu bên trong thuộc về mỗi đối tượng và muốn sử dụng các dữ liệu bên trong này thông thường chúng ta cần phải thực hiện một thao tác gọi là khởi tạo đối với chúng. Việc này có thể được thực hiện bằng cách viết một hàm public riêng biệt và sau đó người dùng có thể gọi chúng nhưng điều này sẽ phá vỡ các qui tắc về sự tách biệt giữa các lập trình viên tạo ra các lớp và những người dùng chúng hay nói một cách khác đây là một công việc quan trong không thể giao cho các lập trình viên thuộc loại client đảm nhiệm.

C++ cung cấp một loại hàm đặc biệt cho phép chúng ta thực hiện điều này, các hàm đó được gọi là các hàm cấu tử (constructor). Nếu như lớp có một hàm cấu tử trình biên dịch sẽ tự động gọi tới nó khi một đối tượng nào đó của lớp được tạo ra, trước khi các lập trình viên client có thể sử dụng chúng. Việc gọi tới các cấu tử này không phụ thuộc vào việc sử dụng hay khai báo các đối tượng, nó được thực hiện bởi trình biên dịch vào thời điểm mà đối tượng được tạo ra.

Các hàm cấu tử này thường thực hiện các thao tác gán các giá trị khởi tạo cho các biến thành viên, mở các file input, thiết lập các kết nối tới các máy tính khác trên mạng…

Hàm tạo: là một phương thức của lớp dùng để tạo dựng một đối tượng mới. Chương trình dịch sẽ cấp phát bộ nhớ cho đối tượng, sau đó sẽ gọi đến hàm tạo. Hàm tạo sẽ gán giá trị cho các thuộc tính của đối tượng và có thể thực hiện một số công việc khác nhằm chuẩn bị cho đối tượng mới.

Tên của các hàm cấu tử là tên của lớp, chúng buộc phải là các hàm public, vì nếu không trình biên dịch sẽ không thể gọi tới chúng. Các hàm cấu tử có thể có các tham số nếu cần thiết nhưng nó không trả về bất cứ giá trị nào và cũng không phải là hàm kiểu void.

Dựa vào các tham số đối với một cấu tử người ta chia chúng ra thành một số loại hàm cấu tử:

Cấu tử mặc định (default constructor)

Cấu tử mặc định là cấu tử mà mọi tham số đều là mặc định hoặc không có tham số, cấu tử mặc định có thể được gọi mà không cần bất kỳ tham số nào.

Ví dụ:

```
#include <iostream>
class Point{ // Khai báo lớp Point
  int x,y;   // Properties: x and y coordinates
 public:
  Point(); // Declaration of the default constructor
  bool move(int, int); // A function to move points
  void print(); // to print coordinates on the screen
};
Point::Point(){
  cout << "Constructor is called..." << endl;
  x = 0; // Assigns zero to coordinates
  y = 0;
}
bool Point::move(int new_x, int new_y){
  if (new_x >=0 && new_y>=0){
  x = new_x; // assigns new value to x coordinat
            y = new_y;      // assigns new value to y coordinat
            return true;
  }
  return false;
}
void Point::print(){       cout << "X= " << x << ", Y= " << y << endl; }
int main(){
  Point p1,p2;                    // Default construct is called 2 times
  Point *pp = new Point;          // Default construct is called once
  p1.print();               // p1's coordinates to the screen
  p2.print();               // p2's coordinates to the screen
  pp->print();              // Coordinates of the object pointed by pp to the screen
  return 0;
}
```

Cấu tử có tham số

Giống như các hàm thành viên, các cấu tử cũng có thể có các tham số, khi sử các lớp với các cấu tử có tham số các lập trình viên cần cung cấp các tham số cần thiết.

Ví dụ:

```
class Point{       // Declaration Point Class
   int x,y;          // Properties: x and y coordinates
```

```
 public:
   Point(int, int);          // Declaration of the constructor
   bool move(int, int);      // A function to move points
   void print();             // to print cordinates on the screen
};
Point::Point(int x_first, int y_first){
  cout << "Constructor is called..." << endl;
  if ( x_first < 0 )                // If the given value is negative
        x = 0;                      // Assigns zero to x
  else
        x = x_first;
  if ( y_first < 0 )                // If the given value is negative
        y = 0;                      // Assigns zero to x
  else
        y = y_first;
}
```

Cấu tử Point(int, int) có nghĩa là chúng ta cần có hai tham số kiểu int khi khai báo một đối tượng của lớp Point. Ví dụ:

```
Point p1(20,100), p2(-10,45);   // Constructor is called 2 times
Point *pp = new Point(10,50);  // Constructor is called once
```

Cấu tử với các tham số có giá trị mặc định

Giống như các hàm khác, các tham số của các cấu tử cũng có thể có các giá trị mặc định:

```
Point::Point(int x_first=0, int y_first=0){
  cout << "Constructor is called..." << endl;
  if ( x_first < 0 )                // If the given value is negative
        x = 0;                      // Assigns zero to x
  else
        x = x_first;
  if ( y_first < 0 )                // If the given value is negative
        y = 0;                      // Assigns zero to x
  else
        y = y_first;
}
```

Khi đó chúng ta có thể thực hiện khai báo các đối tượng của lớp Point như sau:

```
Point p1(19, 20); // x = 19, y = 20
```

Point p1(19); // x = 19, y = 0

Và hàm cấu tử trong đó tất cả các tham số đều có thể nhận các giá trị mặc định có thể được sử dụng như một cấu tử mặc định:

Point p3; // x = 0, y = 0

Chồng hàm cấu tử

Một lớp có thể có nhiều cấu tử khác nhau bằng cách chồng hàm cấu tử:

```
class Point{
  public:
        Point(); // Cấu tử mặc định
        Point(int, int); // Cấu tử có tham số
};
```

Khởi tạo mảng các đối tượng

Khi một mảng các đối tượng được tạo ra, cấu tử mặc định của lớp sẽ được gọi đối với mỗi phần tử (là một đối tượng) của mảng. Ví dụ:

Point a[10]; // Cấu tử mặc định sẽ được gọi tới 10 lần

Cần chú ý là nếu lớp Point không có cấu tử mặc định thì khai báo như trên sẽ là sai.

Chúng ta cũng có thể gọi tới các cấu tử có tham số của lớp bằng cách sử dụng một danh sách các giá trị khởi tạo, ví dụ:

Point::Point(int x, int y=0);

Point a[] = {{20}, {30}, Point(20,40)}; // mảng có 3 phần tử

Nếu như lớp Point có thêm một cấu tử mặc định chúng ta cũng có thể khai báo như sau:

Point a[5] = {{20}, {30}, Point(20,40)}; // Mảng có 5 phần tử

Khởi tạo dữ liệu với các cấu tử

Các hàm cấu tử có thể thực hiện khởi tạo các thành phần dữ liệu trong thân hàm hoặc bằng một cơ chế khác, cơ chế này đặc biệt được sử dụng khi khởi tạo các thành phần là hằng số. Ví dụ:

Chúng ta xem xét lớp sau đây:

```
class C{
  const int ci;
  int x;
  public:
C(){
x = 0; // đúng vì x không phải là hằng mà là biến
ci = 0; // Sai vì ci là một hằng số
}
};
```

Thậm chí ví dụ sau đây cũng không đúng:

```
class C{
  const int ci = 0;
  int x;
};
```

và giải pháp của chúng ta là sử dụng cơ chế khởi tạo (constructor initializer) của cấu tử:

```
class C{
  const int ci;
  int x;
  public:
        C():ci(0){
x = 0;
}
};
```

Cơ chế này cũng có thể được sử dụng để khởi tạo các thành phần không phải là hằng của lớp:

```
class C{
  const int ci;
  int x;
  public:
        C():ci(0),x(0){}
};
```

4.3.2. Cấu tử copy

Cấu tử copy là một cấu tử đặc biệt và nó được dùng để copy nội dung của một đối tượng sang một đối tượng mới trong quá trình xây dựng đối tượng mới đó.

Tham số input của nó là một tham chiếu tới các đối tượng cùng kiểu. Nó nhận tham số như là một tham chiếu tới đối tượng sẽ được copy sang đối tượng mới.

Cấu tử copy thường được tự động sinh ra bởi trình biên dịch nếu như tác giả không định nghĩa cho tác phẩm tương ứng.

Nếu như trình biên dịch sinh ra nó, cấu tử copy sẽ làm việc theo kiểu học máy, nó sẽ copy từng byte từng byte một. Đối với các lớp đơn giản không có biến con trỏ thì điều này là đủ, nhưng nếu có một con trỏ là thành viên của lớp thì việc copy theo kiểu học máy này (byte by byte) sẽ làm cho cả hai đối tượng (mới và cũ) cùng trỏ vào một địa chỉ. Do đó khi chúng ta thay đổi đối tượng mới sẽ làm ảnh hưởng tới đối tượng cũ và ngược lại. Đây không phải là điều chúng ta muốn, điều chúng ta muốn là thay vì copy địa chỉ con trỏ cấu tử copy sẽ copy nội dung mà biến con trỏ trỏ tới và để đạt được điều đó chúng ta buộc phải tự xây dựng cấu tử copy đối với các lớp có các thàn viên là biến con trỏ.

Ví dụ:

```
#include <iostream>
#include <string.h>
class String{
   int size;
   char *contents;
 public:
   String(const char *);            // Constructor
   String(const String &);          // Copy Constructor
   void print();                    // Prints the string on the screen
   ~String();                       // Destructor
};
// Constructor
// copies the input character array to the contents of the string
String::String(const char *in_data){
   cout<< "Constructor has been invoked" << endl;
   size = strlen(in_data);
   contents = new char[size +1];   // +1 for null character
   strcpy(contents, in_data);              // input_data is copied to the contents
 }
// Copy Constructor
String::String(const String &object_in)   {
   cout<< "Copy Constructor has been invoked" << endl;
    size = object_in.size;
    contents = new char[size + 1];         //  +1 for null character
    strcpy(contents, object_in.contents);
}
void String::print(){
   cout<< contents << " " << size << endl;
}
// Destructor
// Memory pointed by contents is given back to the heap
String::~String(){
   cout << "Destructor has been invoked" << endl;
   delete[] contents;
}
```

```
//------- Main Function -------
int main(){
  String my_string("string 1");
  my_string.print();
  String other = my_string;              // Copy constructor is invoked
  String more(my_string);                // Copy constructor is invoked
  other.print();
  more.print();
  return 0;
}
```

4.3.3. Hủy tử

Ý tưởng và khái niệm về huỷ tử rất giống với cấu tử ngoại trừ việc huỷ tử được tự động gọi đến khi một đối tượng không được sử dụng nữa (out of scope) (thường là các đối tượng cục bộ) hoặc một đối tượng động (sinh ra bởi việc sử dụng toán tử new) bị xóa khỏi bộ nhớ bằng toán tử delete.

Trái ngược với các hàm cấu tử các hàm hủy tử thường được gọi đến nhằm mục đích giải phóng vùng nhớ đang bị một đối tượng nào đó sử dụng, ngắt các kết nối, đóng các file hay ví dụ trong các chương trình đồ họa là xóa những gì mà đối tượng đã vẽ ra trên màn hình.

Hàm hủy: là một pt của lớp, có chức năng ngược với hàm tạo. Hàm hủy được gọi trước khi giải phóng (xóa bỏ) đối tượng khỏi bộ nhớ. Việc gọi hàm hủy nhằm thực hiện một số công việc có tính "dọn dẹp" trước khi đối tượng được hủy bỏ.

Huỷ tử của một lớp cũng có tên trùng với tên lớp nhưng thêm một ký tự "~" đứng trước tên lớp. Một hàm huỷ tử không có kiểu trả về (không phải là hàm kiểu void) và không nhận tham số, điều này có nghĩa là mỗi lớp chỉ có một hủy tử khác với cấu tử.

Ví dụ:

```
class String{
   int size;
   char *contents;
 public:
   String(const char *);   // Constructor
   void print();           // A member function
   ~String();              // Destructor
};
```

Thực tế thư việc chuẩn của C++ có xây dựng một lớp string. Các lập trình viên không cần xây dựng lớp string riêng cho mình. Chúng ta xây dựng lớp String trong ví dụ trên chỉ để minh họa cho khái niệm về hàm hủy tử.

```
String::String(const char *in_data){
```

```
   cout<< "Constructor has been invoked" << endl;
   size = strlen(in_data);
   contents = new char[size +1];   // +1 for null character
   strcpy(contents, in_data);      // input_data is copied to the contents
 }
void String::print(){
   cout<< contents << " " << size << endl;
}
// Destructor
// Memory pointed by contents is given back to the heap
String::~String(){
   cout << "Destructor has been invoked" << endl;
   delete[] contents;
}
//------- Main Function -------
int main(){
   cout << "--------- Start of Blok 1 ------" << endl;
   String string1("string 1");
   String string2("string 2");
   {
            cout << "--------- Start of Blok 2 ------" << endl;
            string1.print();          string2.print();          String string3("string 3");
            cout << "--------- End of Blok 2 ------" << endl;
   }
   cout << "--------- End of Blok 1 ------" << endl;
   return 0;
 }
```

Ví dụ:

```
// Listing 11.13
// Linked list simple implementation
#include <iostream>
// object to add to list
class CAT{
  public:
            CAT() { itsAge = 1;}
            CAT(int age):itsAge(age){}
```

```
        ~CAT(){};
        int GetAge() const { return itsAge; }
  private:
        int itsAge;
};
// manages list, orders by cat's age!
class Node{
  public:
        Node (CAT*);
        ~Node();
        void SetNext(Node * node) { itsNext = node; }
        Node * GetNext() const { return itsNext; }
        CAT * GetCat() const { return itsCat; }
        void Insert(Node *);
        void Display();
  private:
        CAT *itsCat;
        Node * itsNext;
};
Node::Node(CAT* pCat):itsCat(pCat),itsNext(0){}
Node::~Node(){
  cout << "Deleting node...\n";
  delete itsCat;
  itsCat = 0;
  delete itsNext;
  itsNext = 0;
}
// ************************************
void Node::Insert(Node* newNode){
  if (!itsNext)
  itsNext = newNode;
  else     {
        int NextCatsAge = itsNext->GetCat()->GetAge();
        int NewAge =  newNode->GetCat()->GetAge();
        int ThisNodeAge = itsCat->GetAge();
        if (  NewAge >= ThisNodeAge && NewAge < NextCatsAge  )      {
```

```
                newNode->SetNext(itsNext);
                itsNext = newNode;
            }
  else      itsNext->Insert(newNode);
  }
}
void Node::Display(){
  if (itsCat->GetAge() > 0)       {
  cout << "My cat is ";  cout << itsCat->GetAge() << " years old\n";
  }
  if (itsNext)              itsNext->Display();
}
int main(){
  Node *pNode = 0;
  CAT * pCat = new CAT(0);
  int age;
  Node *pHead = new Node(pCat);
  while (1)         {
  cout << "New Cat's age? (0 to quit): ";                cin >>  age;
            if (!age)          break;
            pCat = new CAT(age);
            pNode = new Node(pCat);
            pHead->Insert(pNode);
  }
  pHead->Display();
  delete pHead;
  cout << "Exiting...\n\n";
  return 0;
}
```

4.4. Thành phần tĩnh, các hàm bạn và các lớp bạn

4.4.1. Các hàm bạn và các lớp bạn

Một hàm hoặc một thực thể lớp có thể được khai báo là một bạn bè của một lớp khác.

Một bạn bè của một lớp có quyền truy cập vào tất cả các thành viên (private, protected, public) của lớp. Ví dụ:

```
class A{
friend class B;
```

```
priavte:
        int i;
        float f;
public:
        void fonk1(char *c);
};
class B{
int j;
public:
        void fonk2(A &s){ cout << s.i;} // lớp B có thể truy cập tới mọi thành
        // viên của A nhưng ngược lại thì không thể vì A không là bạn của B
};
```

Một hàm bạn có quyền truy cập vào tất cả các thành viên của lớp:

```
class Point{
friend void zero(Point &);
int x, y;
public:
        bool move(int x, int y);
        void print();
        bool is_zero();
};
void zero(Point & p){
  p.x = p.y = 0;
}
```

Hoặc chúng ta cũng có thể khai báo một hàm thành viên của một lớp nào đó là hàm bạn của một lớp:

```
class X;
class Y {
void f(X*);
};
class X { // Definition
private:
        int i;
public:
void initialize();
        friend void g(X*, int); // hàm bạn toàn cục
```

```
        friend void Y::f(X*);  // hàm bạn của lớp khác
        friend void h();
};
```

Trong ví dụ này chúng ta thấy lớp Y có một hàm thành viên f(), hàm này sẽ làm việc với một đối tượng thuộc lớp X. Điều này hơi phức tạp vì trình biên dịch C++ yêu cầu tất cả mọi thứ đều phải được khai báo trước khi chúng ta tham chiếu tới chúng, vì thế lớp Y phải khai báo thành viên của nó là hàm Y::f(X *) trước khi có thể khai báo hàm này là một hàm bạn của lớp X. Nhưng hàm Y::f(X *) có truy cập tới đối tượng thuộc lớp X vì thế nên lớp X cũng phải được khai báo trước đó.

Điều này khá nan giải, tuy nhiên chúng ta chú ý tới hàm f(), tham số truyền vào cho hàm là một biến con trỏ có nghĩa là chúng ta sẽ truyền một địa chỉ và thật may mắn là trình biên dịch luôn biết cách làm việc với các biến địa chỉ dù cho đó là địa chỉ của một đối tượng thuộc kiểu gì chăng nữa dù cho nó không biết chính xác về định nghĩa hay đơn giản là kích thước của kiểu đó. Điều này cũng có nghĩa là nếu hàm thành viên chúng ta cần khai báo lớp X một cách đầy đủ trước khi muốn khai báo một hàm chẳng hạn như Y::g(X).

Bằng cách truyền theo địa chỉ trình biên dịch cho phép chúng ta thực hiện một đặc tả kiểu không hoàn chỉnh (incomplete type specification) về lớp X trước khi khai báo Y::f(X *). Điều này được thực hiện bằng cách khai báo:

```
class X;
```

Khai báo này đơn giản báo cho trình biên dịch biết là có một lớp có tên là X, và vì thế việc tham chiếu tới lớp đó sẽ là hợp lệ miễn là việc tham chiếu đó không đòi hỏi nhiều hơn 1 cái tên.

Một điều nữa mà chúng ta cần chú ý là trong trường hợp khai báo các lớp lồng nhau, lớp ngoài cũng không có quyền truy cập vào các thành phần private của lớp trong, để đạt được điều này chúng ta cũng cần thực hiện tương tự như trên, ví dụ:

```
#include <iostream>
#include <string.h> // memset()
const int sz = 20;
class Holder {
private:
        int a[sz];
public:
void initialize();
        class Pointer;
friend Pointer;
        class Pointer {
                private:
Holder* h;
int* p;
```

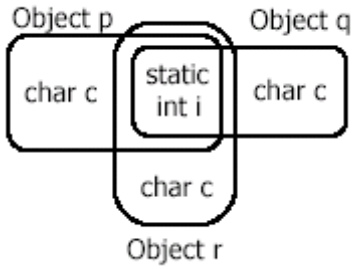
```
public:
void initialize(Holder* h);
// Move around in the array:
void next();
void previous();
void top();
void end();
// Access values:
int read();
void set(int i);
};
};
void Holder::initialize() {
memset(a, 0, sz * sizeof(int));
}
void Holder::Pointer::initialize(Holder* rv) {
h = rv;
p = rv->a;
}
void Holder::Pointer::next() {
if(p < &(h->a[sz - 1])) p++;
}
void Holder::Pointer::previous() {
if(p > &(h->a[0])) p--;
}
void Holder::Pointer::top() {
p = &(h->a[0]);
}
void Holder::Pointer::end() {
p = &(h->a[sz - 1]);
}
int Holder::Pointer::read() {
return *p;
}
void Holder::Pointer::set(int i) {
*p = i;
```

```
}
int main() {
Holder h;
Holder::Pointer hp, hp2;
int i;
  h.initialize();
hp.initialize(&h);
hp2.initialize(&h);
for(i = 0; i < sz; i++) { hp.set(i); p.next();}
hp.top();
hp2.end();
for(i = 0; i < sz; i++) {cout << "hp = " << hp.read()<< ", hp2 = " << hp2.read() << endl;
hp.next();
hp2.previous();
}
}
```

Các hàm friend không phải là thành viên của lớp nhưng chúng buộc phải xuất hiện trong khai báo lớp và do đó tất cả mọi người đều biết đó là một hàm ưu tiên và điều này thực sự là không an toàn. Bản thân C++ không phải là một ngôn ngữ hướng đội tượng hoàn toàn và việc sử dụng các lớp và hàm friend là nhằm giải quyết các vấn đề thực tế đồng thời cũng làm cho tính hướng đối tượng của ngôn ngữ giảm đi đáng kể.

4.4.2. Các thành viên tĩnh của lớp

Thông thường mỗi đối tượng của một lớp đều có một bản copy riêng tất cả các thành viên dữ liệu của lớp. Trong các trường hợp cụ thể đôi khi chúng ta muốn là tất cả các đối tượng của lớp sẽ chia sẻ cùng một thành viên dữ liệu nào đó. Và đó chính là lý do tồn tại của các thành viên dữ liệu tĩnh hay còn gọi là biến của lớp.




```
class A{ char c; static int i;};
int main(){
  …
```

```
  A p, q, r;
  …
}
```

Các thành viên tĩnh tồn tại ngay cả khi không có đối tượng nào của lớp được tạo ra trong chương trình. Chúng cũng có thể là các thành phần public hoặc private.

Để truy cập vào các thành phần dữ liệu tĩnh **public** của một lớp khi không có đối tượng nào của lớp tồn tại chúng ta sẽ sử dụng tên lớp và toán tử "::", ví dụ: A::i = 5;

Để truy cập vào các thành phần dữ liệu tĩnh **private** của một lớp khi không có đối tượng nào của lớp tồn tại, chúng ta cần có một hàm thành viên tĩnh **public**.

Các biến tĩnh bắt buộc phải được khởi tạo một lần (và chỉ một lần) trước khi chúng được sử dụng.

```
#include <iostream>
class A{
  char c;
  static int number;          //  Number of created objects (static data)
 public:
  static void setNum(){number=0;}       // Static function to initialize number
  A(){number++; cout<< "\n"<< "Constructor "<< number;} //Constructor
  ~A(){number--; cout<< "\n"<< "Destructor "<< number;} //Destructor
};
int A::number;     // Allocating memory for number
// Chú ý nếu không có đoạn này chương trình sẽ báo lỗi
// ----- Main function -----
int main(){
cout<<"\n Entering 1. BLOCK.....";
  A::setNum();   // The static function is called
  A a,b,c;
  {
                    cout<<"\n Entering 2. BLOCK.....";
                    A d,e;
                    cout<<"\n Exiting 2. BLOCK.....";
  }
  cout<<"\n Exiting 1. BLOCK.....";
  return 0;
}
```

4.4.3 Đối tượng hằng và các hàm thành viên hằng

Chúng ta có thể sử dụng từ khóa const để chỉ ra rằng một đối tượng là không thể thay đổi (not modifiable) hay là đối tượng hằng. Tất cả các cố gắng nhằm thay đổi nội dung của các đối tượng hằng đều gây ra các lỗi. Ví dụ:

const ComplexT cz(0,1);

C++ hoàn toàn ngăn cấm việc gọi tới các hàm thành viên của các đối tượng hằng trừ khi các hàm thành viên đó là các hàm hằng, có nghĩa là các hàm không thay đổi các thành phần bên trong của đối tượng.

Để khai báo một hàm như vậy chúng ta cũng sử dụng từ khóa const:

Ví dụ:

```
class Point{              // Declaration Point Class
  int x,y;                // Properties: x and y coordinates
 public:
  Point(int, int);        // Declaration of the constructor
  bool move(int, int);    // A function to move points
  void print() const;     // constant function: prints coordinates on the screen
};
```

Khi đó chúng ta có thể khai báo các đối tượng hằng của lớp Point và gọi tới hàm print của nó.

4.5. Chồng toán tử

Chúng ta đã biết rằng có thể thực hiện chồng hàm, bản thân các toán tử cũng là các hàm và vì thế nên hoàn toàn có thể thực hiện chồng các toán tử (hay các hàm toán tử) chẳng hạn như các toán tử +, -, *, >= hay ==, khi đó chúng sẽ gọi tới các hàm khác nhau tùy thuộc vào các toán hạng của chúng. Ví dụ với toán tử +, biểu thức a + b sẽ gọi tới một hàm cộng hai số nguyên nếu a và b thuộc kiểu int nhưng sẽ gọi tới một hàm khác nếu chúng là các đối tượng của một lớp nào đó mà chúng ta mới tạo ra.

Chồng toán tử (operator overloading) là một đặc điểm thuận tiện khác của C++ làm cho các chương trình dễ viết hơn và cũng dễ hiểu hơn.

Thực chất chồng toán tử không thêm bất cứ một khả năng mới nào vào C++. Tất cả những gì chúng ta có thể thực hiện với một toán tử chồng đều có thể thực hiện được với một hàm nào đó. Tuy nhiên việc chồng các toán tử làm cho chương trình dễ viết, dễ đọc và dễ bảo trì hơn.

Chồng toán tử là cách duy nhất để gọi một nào đó hàm theo cách khác với cách thông thường. Xem xét theo khía cạnh này chúng ta không có bất cứ lý do nào để thực hiện chồng một toán tử nào đó trừ khi nó làm cho việc cài đặt các lớp trong chương trình dễ dàng hơn và đặc biệt là dễ đọc hơn (lý do này quan trọng hơn cả).

Các hạn chế của chồng toán tử

Hạn chế thứ nhất là chúng ta không thể thực hiện cài đặt các toán tử không có trong C++. Chẳng hạn không thể cài hàm toán tử ** để thực hiện lấy luỹ thừa. **Chúng ta chỉ có thể thực hiện chồng các toán tử thuộc loại built-in của C++**.

Thậm chí một số các toán tử sau đây: toán tử dấu chấm (.), toán tử phân giải tầm hoạt động (::), toán tử điều kiện (?:), toán tử sizeof cũng không thể overload.

Các toán tử của C++ có thể chia thành hai loại là toán tử một ngôi và toán tử hai ngôi. Và nếu một toán tử thuộc kiểu binary thì toán tử được chồng của nó cũng là toán tử hai ngôi và tương tự đối với toán tử một ngôi. Độ ưu tiên của toán tử cũng như số lượng hay cú pháp của các toán hạng là không đổi đối với hàm chồng toán tử. Ví dụ như toán tử * bao giờ cũng có độ ưu tiên cao hơn toán tử +. Tất cả các toán tử được sử dụng trong biểu thức chỉ nhận các kiểu dữ liệu built-in không thể thay đổi. Chẳng hạn chúng ta không bao giờ chồng toán tử + để biểu thức a = 3 + 5 hay 1<< 4 có ý nghĩa khác đi.

Ít nhất thì một toán hạng phải thuộc kiểu dữ liệu người dùng định nghĩa (lớp).

Ví dụ:

```
class ComplexT{
   double re,im;
 public:
   ComplexT(double re_in=0,double im_in=1);      // Constructor
   ComplexT operator+(const ComplexT & ) const; // Function of operator +
   void print() const;
};
ComplexT ComplexT::operator+(const ComplexT &c) const
{
  double re_new, im_new;
  re_new=re+c.re;
  im_new=im+c.im;
  return ComplexT(re_new,im_new);
}
int main()
{
ComplexT z1(1,1),z2(2,2),z3;
z3=z1+z2;         // like z3 = z1.operator+(z2);
z3.print();
return 0;
}
```

Chồng toán tử gán (=)

Việc gán một đối tượng này cho một đối tượng khác cùng kiểu (cùng thuộc một lớp) là một công việc mà hầu hết mọi người (các lập trình viên) đều mong muốn là có thể thực hiện

một cách dễ dàng nên trình biên dịch sẽ tự động sinh ra một hàm để thực hiện điều này đối với mỗi lớp được người dùng tạo ra nếu họ không có ý định cài đặt hàm đó:

type::operator(const type &);

Hàm này thực hiện theo cơ chế gán thành phần, có nghĩa là nó sẽ thực hiện gán từng biến thành viên của đối tượng này cho một đối tượng khác có cùng kiểu (cùng lớp). Nếu như đối với các lớp không có gì đặc biệt, thao tác này là đủ thì chúng ta cũng không cần thiết phải thực hiện cài đặt hàm toán tử này, chẳng hạn việc cài đặt hàm toán tử gán đối với lớp ComplexT là không cần thiết:

```
void ComplexT::operator=(const ComplexT & z){
  re = z.re;
  im = z.im;
}
```

Nói chung thì chúng ta thường có xu hướng tự cài đặt lấy hàm toán tử gán đối với các lớp được sử dụng trong chương trình và đặc biệt là với các lớp tinh vi hơn chẳng hạn:

```
class String{
  int size;
  char *contents;
 public:
  String();                                          //default constructor
  String(const char *);              // constructor
  String(const String &);   // copy constructor
  const String& operator=(const String &);  // assignment operator
  void print() const ;
  ~String();              // Destructor
}
```

Chú ý là trong trường hợp trên hàm toán tử = có kiểu là void do đó chúng ta không thể thực hiện các phép gán nối tiếp nhau kiểu như (a = b = c;).

```
const String& String::operator=(const String &in_object)  {
  cout<< "Assignment operator has been invoked" << endl;
  size = in_object.size;
  delete[] contents;        // delete old contents
  contents = new char[size+1];
  strcpy(contents, in_object.contents);
  return *this;     // returns a reference to the object
}
```

Sự khác biệt giữa hàm toán tử gán và cấu tử copy là ở chỗ cấu tử copy sẽ thực sự tạo ra một đối tượng mới trước khi copy dữ liệu sang cho nó còn hàm toán tử gán thì chỉ thực hiện việc copy dữ liệu sang cho một đối tượng có sẵn.

Chồng toán tử chỉ số []

Các qui luật chung chúng ta đã trình bày được áp dụng đối với mọi toán tử. Vì thế chúng ta không cần thiết phải bàn luận về từng loại toán tử. Tuy nhiên chúng ta sẽ khảo sát một vài toán tử được người ta cho là thú vị. Và một trong các toán tử đó chính là toán tử chỉ số.

Toán tử này có thể được khai báo theo hai cách như sau:

```
class C{
  returntype & operator [](paramtype);
  hoặc:
  const  returntype & operator[](paramtype)const;
};
```

Cách khai báo thứ nhất được sử dụng khi việc chồng toán tử chỉ số làm thay đổi thuộc tính của đối tượng. Cách khai báo thứ hai được sử dụng đối với một đối tượng hằng; trong trường hợp này, toán tử chỉ số được chồng có thể truy cập nhưng không thể làm thay đổi các thuộc tính của đối tượng.

Nếu c là một đối tượng của lớp C, biểu thức

c[i]

sẽ được dịch thành

c.operator[](i)

Ví dụ: chúng ta sẽ cài đặc hàm chồng toán tử chỉ số cho lớp String. Toán tử sẽ được sử dụng để truy cập vào ký tự thứ i của xâu. Nếu i nhỏ hơn 0 và lớn hơn độ dài của xâu thì ký tự đầu tiên và cuối cùng sẽ được truy cập.

```
char & String::operator[](int i) {
if(i < 0)          return contents[0];              // return first character
if(i >= size)      return contents[size-1];         // return last character
  return contents[i];                        // return i th character
}
```

Chồng toán tử gọi hàm ()

Toán tử gọi hàm là duy nhất, nó duy nhất ở chỗ cho phép có bất kỳ một số lượng tham số nào.

```
class C{ returntype operator()(paramtypes); };
```

Nếu c là một đối tượng của lớp C, biểu thức

c(i,j,k)

sẽ được thông dịch thành:

c.operator()(i,j,k);

Ví dụ toán tử gọi hàm được chồng để in ra các số phức ra màn hình. Trong ví dụ này toán tử gọi hàm không nhận bất cứ một tham số nào.

```
void ComplexT::operator()()const{      cout << re << “, “ << im << endl;}
```

Ví dụ: toán tử gọi hàm được chồng để copy một phần nội dung của một xâu tới một vị trí bộ nhớ xác định.

Trong ví dụ này toán tử gọi hàm nhận hai tham số: địa chỉ của bộ nhớ đích và số lượng ký tự cần sao chụp.

```
void String::operator()(char * dest, int num) const{
  // numbers of characters to be copied may not exceed the size
if (num>size) num=size;
for (int k=0; k< num; k++)         dest[k]=contents[k];
}
int main(){
  String s1("Example Program");
  char *c=new char[8];  // Destination memory
  s1(c,7);            // Function call operator is invoked
  c[7]='\0';          // End of String (null)
  cout << c << endl;
  delete[] c;
  return 0;
}
```

Chồng các toán tử một ngôi

Các toán tử một ngôi chỉ nhận một toán hạng để làm việc, một vài ví dụ điển hình về chúng chẳng hạn như: ++, --, - và !.

Các toán tử một ngôi không nhận tham số, chúng thao tác trên chính đối tượng gọi tới chúng. Thông thường toán tử này xuất hiện bên trái của đối tượng chẳng hạn như –obj, ++obj…

Ví dụ: Chúng ta định nghĩa toán tử ++ cho lớp ComplexT để tăng phần thực của số phức lên 1 đơn vị 0,1.

```
void ComplexT::operator++(){ re = re + 0.1;}
int main(){        ComplexT z(0.2, 1); ++z; }
```

Để có thể thực hiện gán giá trị được tăng lên cho một đối tượng mới, hàm toán tử cần trả về một tham chiếu tới một đối tượng nào đó:

```
const ComplexT & ComplexT::operator++(){
  re = re + 0.1;
  return this;
}
```

```
int main(){        ComplexT z(0.2, 1), z1; z1 = ++z; }
```

Chúng ta nhớ lại rằng các toán tử ++ và - - có hai dạng sử dụng theo kiểu đứng trước và đứng sau toán hạng và chúng có các ý nghĩa khác nhau. Việc khai báo như trong hai ví dụ trên sẽ chồng toán tử ở dạng đứng trước toán hạng. Các khai báo có dạng operator(int) sẽ chồng dạng đứng sau của toán tử.

```
ComplexT  ComplexT::operator++(int)  {
  ComplexT temp;
  temp=*this;     // saves old value
  re=re+0.1;
  return temp;     // return old value
}
```

Lớp String:

```
enum bool{true = 1, false = 0};
class String{
  int size;
  char *contents;
 public:
  String();                                                    //default constructor
  String(const char *);              // constructor
  String(const String &);   // copy constructor
  const String& operator=(const String &);  // assignment operator
  bool operator==(const String &);  // assignment operator
  bool operator!=(const String &rhs){return !(*this==rhs);};  // assignment operator
  void print() const ;
  ~String();              // Destructor
  friend ostream & operator <<(ostream &, const String &);
};
// Creates an empty string (only NULL character)
String::String(){
  size = 0;
  contents = new char[1];
  strcpy(contents, "");
}
String::String(const char *in_data){
   size = strlen(in_data);                  // Size of input data
   contents = new char[size + 1];         // allocate mem. for the string, +1 is for NULL
   strcpy(contents, in_data);
```

```
 }
String::String(const String &in_object){
  size = in_object.size;
  contents = new char[size+1];
  strcpy(contents, in_object.contents);
}
// Assignment operator
const String& String::operator=(const String &in_object){
  size = in_object.size;
  delete[] contents;
  // delete old contents
  contents = new char[size+1];
  strcpy(contents, in_object.contents);
  return *this;              // returns a reference to the object
}
bool String::operator==(const String &rhs){
  if(size == rhs.size){
        for(int i=0;i<=size&&(contents[i]==rhs.contents[i]);i++);
        if(i>size)return true;
  }
  return false;
}
// This method prints strings on the screen
void String::print() const{ cout<< contents << " " << size << endl; }
//Destructor
String::~String(){        delete[] contents; }
ostream & operator <<(ostream & out, const String & rhs){ out << rhs.contents; return
out; }
```

Bài tập

1. Xây dựng lớp sinh viên
2. Xây dựng lớp nhân viên
3. Xây dựng lớp môn học
4. Xây dựng lớp mặt hàng
5. Xây dựng lớp sinh viên với các toán tử >>, << để nhập, xuất thông tin
6. Xây dựng lớp phân số với các toán tử >>, << để nhập, xuất thông tin. Các toán tử +, -, *, / hai phân số.

CHƯƠNG 5. THỪA KẾ.

Kế thừa là một cách trong lập trình hướng đối tượng để có thể thực hiện được khả năng "sử dụng lại mã chương trình". Sử dụng lại mã chương trình có nghĩa là dùng một lớp sẵn có trong một khung cảnh chương trình khác. Bằng cách sử dụng lại các lớp chúng ta có thể làm giảm thời gian và công sức cần thiết để phát triển một chương trình đồng thời làm cho chương trình phần mềm có khả năng và qui mô lớn hơn cũng như tính tin cậy cao hơn.

5.1. Lớp cơ sở, lớp dẫn xuất

5.1.1. Sử dụng lại mã chương trình

Cách tiếp cận đầu tiên nhằm mục đích sử dụng lại mã chương trình đơn giản là viết lại các đoạn mã đã có. Chúng ta có một đoạn mã chương trình nào đó đã sử dụng tốt trong một chương trình cũ nào đó, nhưng không thực sự đáp ứng được yêu cầu của chúng ta trong một dự án mới.

Chúng ta sẽ paste đoạn mã cũ đó vào một file mã nguồn mới, thực hiện một vài sửa đổi để nó phù hợp với môi trường mới. Tất nhiên chúng ta lại phải thực hiện gỡ lỗi đoạn mã đó từ đầu và thường thì chúng ta lại thấy tiếc là tại sao không viết hẳn một đoạn mã chương trình mới.

Để làm giảm các lỗi có thể có khi thay sửa đổi mã chương trình, các lập trình viên cố gắng tạo ra các đoạn mã có thể được sử dụng lại mà không cần băn khoăn về khả năng gây lỗi của chúng và đó được gọi là các hàm.

Các hàm thư viện là một bước tiến nữa nhằm sử dụng lại mã chương trình tuy nhiên các thư viện có nhược điểm là chúng mô hình hóa thế giới thực không được tốt lắm vì chúng không bao gồm các dữ liệu quan trọng. Và thường xuyên chúng ta cần thay đổi chúng để có thể phù hợp với môi trường mới và tất nhiên sự thay đổi này lại dẫn đến các lỗi có thể phát sinh.

5.1.2. Sử dụng lại mã chương trình trong OOP

Một cách tiếp cận đầy sức mạnh để sử dụng lại mã chương trình trong lập trình hướng đối tượng là thư viện lớp. Vì các lớp mô hình hóa các thực thể của thế giới thực khá sát nên chúng cần ít các thay đổi hơn các hàm để có thể phù hợp với hoàn cảnh mới. Khi một lớp đã được tạo ra và kiểm thử cẩn thận, nó sẽ là một đơn vị mã nguồn có ích. Và nó có thể được sử dụng theo nhiều cách khác nhau:

Cách đơn giản nhất để sử dụng lại một lớp là sử dụng một đối tượng của lớp đó một cách trực tiếp. Thư viện chuẩn của C++ có rất nhiều đối tượng và lớp có ích chẳng hạn cin và cout là hai đối tượng kiểu đó.

Cách thứ hai là đặt một đối tượng của lớp đó vào trong một lớp khác. Điều này được gọi là "tạo ra một đối tượng thành viên".

Lớp mới có thể được xây dựng bằng cách sử dụng số lượng bất kỳ các đối tượng thuộc các lớp khác theo bất kỳ cách thức kết hợp nào để đạt được các chức năng mà chúng ta mong muốn trong lớp mới. Vì chúng ta xây dựng lên lớp mới (composing) từ các lớp cũ nên ý tưởng này được gọi là composition và nó cũng thường được đề cập tới như là một quan hệ "has a".

Cách thứ ba để sử dụng lại một lớp là kế thừa. Kế thừa là một quan hệ kiểu "is a" hoặc "a kind of".

5.2. Quy tắc thừa kế

5.2.1. Cú pháp khai báo lớp dẫn xuất

OOP cung cấp một cơ chế để thay đổi một lớp mà không làm thay đổi mã nguồn của nó. Điều này đạt được bằng cách dụng kế thừa để sinh ra một lớp mới từ một lớp cũ. Lớp cũ được gọi là lớp cơ sở sẽ không bị sửa đổi, nhưng lớp mới (được gọi là lớp dẫn xuất) có thể sử dụng tất cả các đặc điểm của lớp cũ và các đặc điểm thêm khác của riêng nó. Nếu có một quan hệ cùng loài (kind of) giữa hai đối tượng thì chúng ta có thể sinh một đối tượng này từ đối tượng kia bằng cách sử dụng kế thừa.

Ví dụ chúng ta đều biết Lớp Animal bao gồm tất cả các loài động vật, lớp Fish là một loài động vật nên các đối tượng của lớp Fish có thể được sinh ra từ một đối tượng của lớp Animal.

Ví dụ kế thừa đơn giản nhất đòi hỏi phải có 2 lớp: một lớp cơ sở và một lớp dẫn xuất. Lớp cơ sở không có yêu cầu gì đặc biệt, lớp dẫn xuất ngược lại cần chỉ rõ nó được sinh ra từ lớp cơ sở và điều này được thực hiện bằng cách sử dụng một dấu : sau tên lớp dẫn xuất sau đó tới một từ khóa chẳng hạn public và tên lớp cơ sở.

Cú pháp chung khai báo lớp dẫn xuất như sau: Giả sử lớp A là lớp cơ sở, lớp B là lớp dẫn xuất từ A.

```
class B : kiểu_thừa_kế A{
- Các thành phần dữ liệu của lớp B
- Các phương thức của lớp B
};
```

Trong đó: kiểu_thừa_kế có thể là public, protected hoặc private.

Ví dụ: chúng ta cần mô hình hóa các giáo viên và hiệu trưởng trong trường học. Trước hết giả sử rằng chúng ta có một lớp định nghĩa các giáo viên, sau đó chúng ta có thể sử dụng lớp này để mô hình hóa hiệu trưởng vì hiệu trưởng cũng là một giáo viên:

```
class Teacher{
  protected:
        String name;
        int age, numOfStudents;
  public:
        void setName(const String & new_name){name = new_name;}
};
class Principal: public Teacher{
  String school_name;
  int numOfTeachers;
public:
```

```
  void setSchool(const & String s_name){school_name = s_name;}
};
int main(){
Teacher t1;
Principal p1;
p1.setName("Principal 1"); t1.setName("Teacher 1"); p1.setSchool("Elementary School");
return 0;
}
```

Một đối tượng dẫn xuất kế thừa tất cả các thành phần dữ liệu và các hàm thành viên của lớp cơ sở. Vì thế đối tượng con (dẫn xuất) p1 không chỉ chứa các phần tử dữ liệu school_name và numOfTeachers mà còn chứa cả các thành phần dữ liệu name, age và numOfStudents.

Đối tượng p1 không những có thể truy cập vào hàm thành viên riêng của nó là setSchool() mà còn có thể sử dụng hàm thành viên của lớp cơ sở mà nó kế thừa là hàm setName().

Các thành viên thuộc kiểu private của lớp cơ sở cũng được kế thừa bởi lớp dẫn xuất nhưng chúng không nhìn thấy ở lớp kế thừa. Lớp kế thừa chỉ có thể truy cập vào các thành phần này qua các hàm public giao diện của lớp cơ sở.

5.2.2. Kiểm soát truy cập

a) Kế thừa public

Đây là kiểu kế thừa mà chúng ta hay dùng nhất:

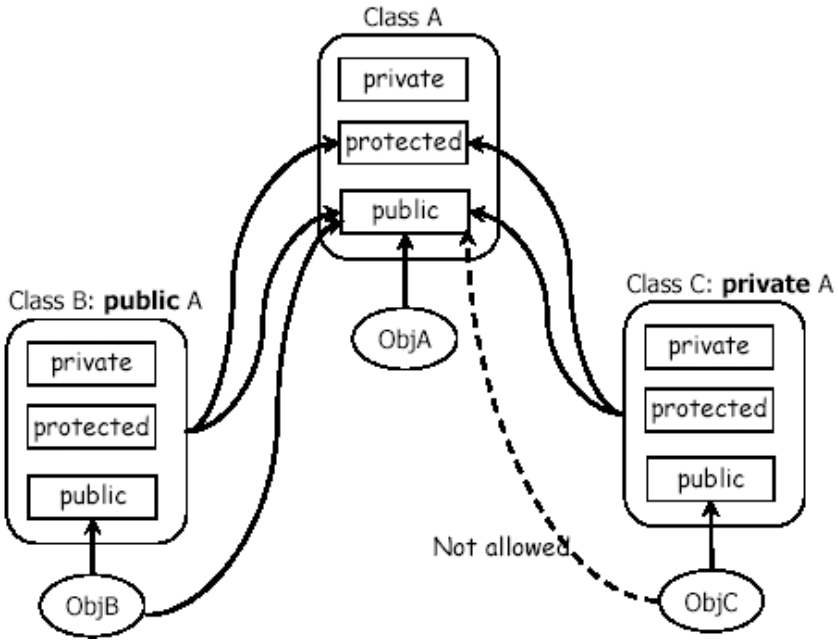
```
class Base{… };
class Derived: public Base{… };
```

Kiểu kế thừa này được gọi là public inheritance hay public derivation. Quyền truy cập của các thành viên của lớp cơ sở không thay đổi. Các đối tượng của lớp dẫn xuất có thể truy cập vào các thành viên public của lớp cơ sở. Các thành viên public của lớp cơ sở cũng sẽ là các thành viên public của lớp kế thừa.

b) Kế thừa private

```
class Base{… };
class Derived: private Base{ …};
```

Kiểu kế thừa này có tên gọi là private inheritance. Các thành viên public của lớp cơ sở trở thành các thành viên private của lớp dẫn xuất. Các đối tượng của lớp dẫn xuất không thể truy cập vào các thành viên của lớp cơ sở. Các hàm thành viên của lớp dẫn xuất có thể truy cập vào các thành viên public và protected của lớp cơ sở.



c) Kiểm soát truy cập

Hãy nhớ rằng khi chưa sử dụng kế thừa, các hàm thành viên của lớp có thể truy cập vào bất cứ thành viên nào của lớp cho dù đó là public, private nhưng các đối tượng của lớp đó chỉ có thể truy cập vào các thành phần public.

Khi kế thừa được đưa ra các khả năng truy cập tới các thành viên khác đã ra đời. Các hàm thành viên của lớp dẫn xuất có thể truy cập vào các thành viên public và protected của lớp cơ sở, trừ các thành viên private. Các đối tượng của lớp dẫn xuất chỉ có thể truy cập vào các thành viên public của lớp cơ sở.

Chúng ta có thể xem rõ hơn trong bảng sau đâu:

	Truy cập từ chính lớp đó	Truy cập từ lớp dẫn xuất	Truy cập từ đối tượng của lớp
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	Yes	No	No

Chúng ta có thể định nghĩa lại hai lớp Teacher và Principal như sau:

```
class Teacher{
  private:
        String name;
  protected:
        int age, numOfStudents;
  public:
        void setName(const String & new_name){name = new_name;}
```

```
        void print() const;
};
void Teacher::print() const{
  cout << "Name: " << name << " Age: " << age << endl;
  cout << "Number of Students: " << numOfStudents << endl;
};
class Principal: public Teacher{
  private:
        String school_name;
        int numOfTeachers;
public:
  void setSchool(const & String s_name){school_name = s_name;}
  void print() const;
  int getAge() const{return age;}
  const String & getName(){return name;}
};
int main(){
  Teacher t1;
  Principal p1;
  t1.numOfStudents = 100; // sai
  t1.setName("Ali Bilir");
  p1.setSchool("Istanbul Lisesi");
  return 0;
}
```

Sự khác nhau giữa các thành viên private và các thành viên protected:

Nói chung dữ liệu của lớp nên là private. Các thành viên dữ liệu public có thể bị sửa đổi bất kỳ lúc nào trong chương trình nên được tránh sử dụng. Các thành viên dữ liệu protected có thể bị sửa đổi bởi các hàm trong bất kỳ lớp kế thừa nào. Bất kỳ người dùng nào cũng có thể kế thừa một lớp nào đó và truy cập vào các thành viên dữ liệu protected của lớp cơ sở. Do đó sẽ an toàn và tin cậy hơn nếu các lớp kế thừa không thể truy cập vào các dữ liệu của lớp cơ sở một cách trực tiếp.

Nhưng trong các hệ thống thời gian thực, nơi mà tốc độ là rất quan trọng, các lời gọi hàm truy cập vào các thành viên private có thể làm chậm chương trình. Trong các hệ thống như vậy dữ liệu có thể được định nghĩa là protected để các lớp kế thừa có thể truy cập tới chúng trực tiếp và nhanh hơn.

Ví dụ:

```
class A{
```

```
  private:
        int i;
  public:
        void access(int new_i){
              if( new_i > 0 && new_i <= 100) i = new_i;
        }
};
class B: public A{
  private:
        int k;
  public:
        void set(int new_i, int new_k){
              A::access(new_i); // an toàn nhưng chậm
              ….
        }
};
class A{
  protected:
        int i;
  public:
        …..
};
class B: public A{
  private:
        int k;
  public:
        void set(int new_i, int new_k){
              i = new_i; // nhanh
              ….
        }
};
```

5.2.3. Các hàm không thể kế thừa

Một vài hàm sẽ cần thực hiện các công việc khác nhau trong lớp cơ sở và lớp dẫn xuất. Chúng là các hàm toán tử gán =, hàm hủy tử và tất cả các hàm cấu tử. Chúng ta hãy xem xét một hàm cấu tử, đối với lớp cơ sở hàm cấu tử của nó có trách nhiệm khởi tạo các thành viên dữ liệu và cấu tử của lớp dẫn xuất có trách nhiệm khởi tạo các thành viên dữ liệu của lớp dẫn

xuất. Và bởi vì các cấu tử của lớp dẫn xuất và lớp cơ sở tạo ra các dữ liệu khác nhau nên chúng ta không thể sử dụng hàm cấu tử của lớp cơ sở cho lớp dẫn xuất và do đó các hàm cấu tử là không thể kế thừa.

Tương tự như vậy toán tử gán của lớp dẫn xuất phải gán các giá trị cho dữ liệu của lớp dẫn xuất, và toán tử gán của lớp cơ sở phải gán các giá trị cho dữ liệu của lớp cơ sở. Chúng làm các công việc khác nhau vì thế toán tử này không thể kế thừa một cách tự động.

5.2.4. Các hàm cấu tử và kế thừa

Khi chúng ta định nghĩa một đối tượng của một lớp dẫn xuất, cấu tử của lớp cơ sở sẽ được gọi tới trước cấu tự của lớp dẫn xuất. Điều này là bởi vì đối tượng của lớp cơ sở là một đối tượng con - một phần - của đối tượng lớp dẫn xuất, và chúng ta cần xây dựng nó từng phần trước khi xây dựng toàn bộ nội dung của nó. Ví dụ:

```
class Parent{
  public:          Parent(){ cout << endl<< "   Parent constructor"; }
};
class Child : public Parent{
  public:          Child(){ cout << endl<<"   Child constructor"; }
};
int main(){
  cout <<  endl<<"Starting";
  Child ch;              // create a Child object
  cout <<  endl<<"Terminating";
  return 0;
}
```

Nếu như cấu tử của lớp cơ sở là một hàm có tham số thì nó cũngcần phải được gọi tới trước cấu tử của lớp dẫn xuất:

```
class Teacher{
  String name;
  int age, numOfStudents;
  public:          Teacher(const String & new_name): name(new_name){}
};
class Principal: public Teacher{
  int numOfTeachers;
  public:          Principal(const String &, int);
};
Principal::Principal(const String & new_name, int numOT):Teacher(new_name){
  NumOfTeachers = numOT;
}
```

Hãy nhớ lại rằng toán tử khởi tạo cấu tử cũng có thể được dùng để khởi tạo các thành viên:

```
Principal::Principal(const String & new_name, int numOT)
:Teacher(new_name),NumOfTeachers(numOT){
}
int main(){        Principal p1(“Ali Baba”, 20); return 0;}
```

Nếu lớp cơ sở có một cấu tử và hàm cấu tử này cần có các tham số thì lớp dẫn xuất phải có một cấu tử gọi tới cấu tử đó với các giá trị tham số thích hợp.

Các hàm hủy tử được gọi tới một cách tự động. Khi một đối tượng của lớp dẫn xuất ra ngoài tầm hoạt động các cấu tử sẽ được gọi tới theo thứ tự ngược lại với thứ tự của các hàm cấu tử. Đối tượng dẫn xuất sẽ thực hiện các thao tác dọn dẹp trước sau đó là đối tượng cơ sở.

Ví dụ:

```
class Parent {
  public:
   Parent() { cout << "Parent constructor" << endl; }
   ~Parent() { cout << "Parent destructor" << endl; }
};

class Child : public Parent {
  public:
   Child() { cout << "Child constructor" << endl; }
   ~Child() { cout << "Child destructor" << endl; }
};
int main(){
   cout << "Start" << endl;
   Child ch;                          // create a Child object
   cout << "End" << endl;
   return 0;
}
```

5.3. Tương thích lớp cơ sở và lớp dẫn xuất

5.3.1. Định nghĩa lại các thành viên

Một vài thành viên (hàm hoặc dữ liệu) của lớp cơ sở có thể không phù hợp với lớp dẫn xuất. Các thành viên này nên được định nghĩa lại trong lớp dẫn xuất. Chẳng hạn lớp Teacher có một hàm thành viên in ra các thuộc tính của cấc giáo viên lên màn hình. Nhưng hàm này là không đủ đối với lớp Principal vì các hiệu trưởng có nhiều thuộc tính hơn các giáo viên bình thường. Vì thế hàm này sẽ được định nghĩa lại:

```
class Teacher{
```

```
  protected:
        String name;
        int age, numOfStudents;
  public:
        void setName(const String & new_name){name = new_name;}
        void print() const;
};
void Teacher::print() const{
  cout << “Name: “ << name << “ Age: “ << age << endl;
  cout << “Number of Students: “ << numOfStudents << endl;
};

class Principal: public Teacher{
  String school_name;
  int numOfTeachers;
public:
  void setSchool(const & String s_name){school_name = s_name;}
  void print() const;
};
void Principal::print() const{
  cout << “Name: “ << name << “ Age: “ << age << endl;
  cout << “Number of Students: “ << numOfStudents << endl;
  cout << “Name of the school: “ << school_name << endl;
};
```

Hàm print() của lớp Principal override (hoặc hide) hàm print() của lớp Teacher. Lớp Principal giờ đây có hai hàm print(). Hàm print() của lớp cơ sở có thể được truy cập bằng cách sử dụng toán tử “::”.

```
void Principal::print() const{    Teacher::print();
  cout << “Name of the school: “ << school_name << endl;
};
```

Chú ý: overloading khác với overriding. Nếu chúng ta thay đổi signature hoặc kiểu trả về của một hàm thành viên thuộc lớp cơ sở thì lớp dẫn xuất sẽ có hai hàm thành viên có tên giống nhau nhưng đó không phải là overloading mà là overriding.

Và nếu như tác giả của lớp dẫn xuất định nghĩa lại một hàm thành viên, thì điều đó có nghĩa là họ muốn thay đổi giao diện của lớp cơ sở. Trong trường hợp này hàm thành viên của lớp cơ sở sẽ bị che đi.

Ví dụ:

```
class A{                    // Base class
public:
        int ia1,ia2;
        void fa1();
int fa2(int);
};
class B: public A{              // Derived class
public:
float ia1;          // overrides ia1
float fa1(float);  // overloads fa1
};
void A::fa1(){    cout << "fa1 of A has been called" << endl;}
int A::fa2(int i){cout << "fa2 of A has been called" << endl; return i; }
float B::fa1(float f){cout << "fa1 of A has been called" << endl;       return f; }
int main(){
  B b;
  int j=b.fa2(1);                   // A::fa2
  b.ia1=4;                          // float fa1 of B
  b.ia2=3;                          // ia2 of A. If it is public
  float y=b.fa1(3.14);              // OK, fa1 of B is called
  //b.fa1();                        // ERROR! fa1 of B needs a floar argument
  b.A::fa1();
  b.A::fa1();
  b.A::ia1=1;
  return 0;
}
```

5.3.2. Con trỏ và các đối tượng

Các đối tượng được lưu trong bộ nhớ nên các con trỏ cũng có thể trỏ tới các đối tượng giống như chúng có thể trỏ tới các biến có kiểu cơ bản.

Các toán tử new và delete được cũng được sử dụng bình thường đối với các con trỏ trỏ tới các đối tượng của một lớp. Toán tử new thực hiện cấp phát bộ nhớ và trả về điểm bắt đầu của vùng nhớ nếu thành công, nếu thất bại nó trả về 0. Khi chúng ta dùng toán tử new nó không chỉ thực hiện cấp phát bộ nhớ mà còn tạo ra đối tượng bằng cách gọi tới cấu tử của lớp tương ứng. Toán tử delete được dùng để giải phóng vùng nhớ mà một con trỏ trỏ tới chiếm giữ.

Danh sách liên kết các đối tượng

Một lớp có thể chứa một con trỏ tới các đối tượng của chính lớp đó. Con trỏ này có thể được sử dụng để xây dựng các cấu trúc dữ liệu chẳng hạn như một danh sách liên kết các đối tượng của một lớp:

```
class Teacher{
  friend class Teacher_list;
  String name;
  int age, numOfStudents;
  Teacher * next;       // Pointer to next object of teacher
 public:
  Teacher(const String &, int, int); // Constructor
  void print() const;
  const String& getName() const {return name;}
  ~Teacher() {    // only to show that the destructor is called
            cout<<" Destructor of teacher" << endl;
   }
};
Teacher::Teacher(const String &new_name,int a,int nos){
  name = new_name;
  age=a;
  numOfStudents=nos;
  next=0;
}
void Teacher::print() const{
   cout <<"Name: "<< name<<"  Age: "<< age<< endl;
   cout << "Number of Students: " <<numOfStudents << endl;
}
class Teacher_list{   // linked list for teachers
Teacher *head;
public:
Teacher_list(){head=0;}
bool append(const String &,int,int);
bool del(const String &);
void print() const ;
~Teacher_list();
};
// Append a new teacher to the end of the list
```

```
// if there is no space returns false, otherwise true
bool Teacher_list::append(const String & n, int a, int nos){
  Teacher *previous, *current, *new_teacher;
  new_teacher=new Teacher(n,a,nos);
  if (!new_teacher) return false;   // if there is no space return false
  if(head) // if the list is not empty
  {
          previous=head;
          current=head->next;
          while(current)            // searh for the end of the list
          {
                  previous=current;
                  current=current->next;
          }
          previous->next=new_teacher;
  }
  else          // if the list is empty
          head=new_teacher;
  return true;
}
// Delete a teacher with the given name from the list
// if the teacher is not found returns false, otherwise true
bool Teacher_list::del(const String & n){
  Teacher *previous, *current;
  if(head)       // if the list is not empty
  {
          if (n==head->getName()) //1st element is to be deleted
          {
                   previous=head; head=head->next; delete previous;   return true;
          }
          previous=head;
          current=head->next;
          while( (current) && (n!=current->getName()) ) // searh for the end of the list
          {
                  previous=current;       current=current->next;
          }
```

```
            if (current==0) return false;
            previous->next=current->next;          delete current;            return true;
  }  //if (head)
  else            // if the list is empty
            return false;
}
// Prints all elements of the list on the screen
void Teacher_list::print() const{
  Teacher *tempPtr;
  if (head){
     tempPtr=head;
     while(tempPtr){
                  tempPtr->print(); tempPtr=tempPtr->next;
            }
  }
  else       cout << "The list is empty" << endl;
}
// Destructor
// deletes all elements of the list
Teacher_list::~Teacher_list(){
  Teacher *temp;
  while(head)        // if the list is not empty
  {
            temp=head;     head=head->next;      delete temp;
  }
}
// ----- Main Function -----
int main(){
  Teacher_list theList;
  theList.print(); theList.append("Teacher1",30,50);
  theList.append("Teacher2",40,65);     theList.append("Teacher3",35,60);
  theList.print();
  if (!theList.del("TeacherX")) cout << " TeacherX not found" << endl;
  theList.print();
  if (!theList.del("Teacher1")) cout << " Teacher1 not found" << endl;
  theList.print();
```

```
  return 0;
}
```

Trong ví dụ trên lớp Teacher phải có một con trỏ trỏ tới đối tượng tiếp theo trong lớp danh sách và lớp danh sách phải được khai báo như là một lớp bạn, để người dùng của lớp này cps thể xây dựng lên các danh sách liên kết. Nếu như lớp này được viết bởi những người làm việc trong một nhóm thì chẳng có vấn đề gì nhưng thường thì chúng ta muốn xây dựng danh sách các đối tượng đã được xây dựng chẳng hạn các danh sách các đối tượng thuộc các lớp thư viện chẳng hạn, và tất nhiên là các lớp này không có các con trỏ tới đối tượng tiếp theo cùng lớp với nó. Để xây dựng các danh sách như vậy chúng ta sẽ xây dựng các lớp lá, mỗi đối tượng của nút lá sẽ lưu giữ các địa chỉ của một phần tử trong danh sách:

```
class Teacher_node{
  friend class Teacher_list;
  Teacher * element;     Teacher_node * next;
  Teacher_node(const String &,int,int); // constructor
  ~Teacher_node();                      // destructor
};
Teacher_node::Teacher_node(const String & n, int a, int nos){
  element = new Teacher(n,a,nos);
  next = 0;
}
Teacher_node::~Teacher_node(){
  delete element;
}
// *** class to define a linked list of teachers ***
class Teacher_list{   // linked list for teachers
   Teacher_node *head;
  public:
   Teacher_list(){head=0;}
   bool append(const String &,int,int);
   bool del(const String &);
   void print() const ;
   ~Teacher_list();
};
// Append a new teacher to the end of the list
// if there is no space returns false, otherwise true
bool Teacher_list::append(const String & n, int a, int nos){
  Teacher_node *previous, *current;
```

```
  if(head)       // if the list is not empty
  {
          previous=head;
          current=head->next;
          while(current)          // searh for the end of the list
          {
                  previous=current;       current=current->next;
          }
          previous->next = new Teacher_node(n, a, nos);
          if (!(previous->next)) return false;              // If memory is full
  }
  else          // if the list is empty
  {
          head = new Teacher_node(n, a, nos); // Memory for new node
          if (!head) return false;                          // If memory is full
  }
  return true;
}
// Delete a teacher with the given name from the list
// if the teacher is not found returns false, otherwise true
bool Teacher_list::del(const String & n){
  Teacher_node *previous, *current;
  if(head)       // if the list is not empty
  {
          if (n==(head->element)->getName()) //1st element is to be deleted
          {
                  previous=head; head=head->next; delete previous;
                  return true;
          }
          previous=head; current=head->next;
          //searh for the end of the list
          while( (current) && (n != (current->element)->getName())) {
                  previous=current;
                  current=current->next;
          }
          if (current==0) return false;
```

```
            previous->next=current->next;
            delete current;
            return true;
  }  //if (head)
  else            // if the list is empty
            return false;
}
// Prints all elements of the list on the screen
void Teacher_list::print() const{
  Teacher_node *tempPtr;
  if (head){
            empPtr=head;
            while(tempPtr){
                      tempPtr->element)->print();
                      empPtr=tempPtr->next;
            }
  }else    cout << "The list is empty" << endl;
}
// Destructor
// deletes all elements of the list
Teacher_list::~Teacher_list(){
  Teacher_node *temp;
  while(head)       // if the list is not empty
  {
            temp=head; head=head->next; delete temp;
  }
}
```

Con trỏ và kế thừa

Nếu như một lớp dẫn xuất Derived có một lớp cơ sở public Base thì một con trỏ của lớp Derived có thể được gán cho một biến con trỏ của lớp Base mà không cần có các thao tác chuyển kiểu tường minh nhưng thao tác ngược lại cần phải được chỉ rõ ràng, tường minh.

Ví dụ một con trỏ của lớp Teacher có thể trỏ tới các đối tượng của lớp Principal. Một đối tượng Principal thì luôn là một đối tượng Teacher nhưng điều ngược lại thì không phải luôn đúng.

```
class Base{};
class Derived: public Base{};
Derived d;
```

```
Base * bp = &d; // chuyển kiểu không tường minh
Derived * dp = bp; // lỗi
dp = static_cast<Derived *>(bp);
```

Nếu như là kế thừa private thì chúng ta không thể thực hiện việc chuyển kiểu không tường minh từ lớp dẫn xuất về lớp cơ sở vì trong trường hợp đó một thành phần public của lớp cơ sở chỉ có thể được truy cập qua một con trỏ lớp cơ sở chứ không thể qua một con trỏ lớp dẫn xuất:

```
class Base{
  int m1;
  public: int m2;
};
class Derived: public Base{};
Derived d;
d.m2 = 5; // Lỗi
Base * bp = &d; // chuyển kiểu không tường minh, lỗi
bp = static_cast<Derived *>(&d);
bp->m2 = 5;
```

Việc kết hợp con trỏ với kế thừa cho phép chúng ta có thể xây dựng các danh sách liên kết hỗn hợp có khả năng lưu giữ các đối tượng thuộc các lớp khác nhau, chúng ta sẽ học kỹ phần này trong chương sau.

5.3.3. Định nghĩa lại các đặc tả truy cập

Các đặc tả truy cập của các thành viên public của lớp cơ sở có thể được định nghĩa lại trong lớp kế thừa. Khi chúng ta kế thừa theo kiểu private, tất cả các thành viên public của lớp cơ sở sẽ trở thành private. Nếu chúng ta muốn chúng vẫn là public trong lớp kế thừa chúng ta sẽ sử dụng từ khóa using (chú ý là Turbo C++ 3.0 không hỗ trợ từ khóa này) và tên thành viên đó (không có danh sách tham số và kiểu trả về) trong phần public của lớp kế thừa:

```
class Base{
  private:
        int k;
  public:
        int i;
        void f();
};
class Derived: public Base{
  private:
        int m;
  public:
```

```
        using Base::f;
        void fb1();
};
int main(){
  Base b;
  Derived d;
  b.i = 5;
  d.i = 0; // Sai
  b.f();
  d.f(); // Ok
  return 0;
}
```

5.4. Các kiểu kế thừa

5.4.1. Đơn thừa kế

Đa kế thừa là trường hợp mà một lớp kế thừa các thuộc tính từ một lớp cơ sở, ví dụ:

```
class Base{       // Base 1
public:
  int a;
  void fa(){cout << "Base1 fa1" << endl;}
  char *fa(int){cout << "Base1 fa2" << endl;return 0;}
};
class Deriv : public Base{
public:
  int a;
  float fa(float){cout << "Deriv fa1" << endl; return 1.0;}
};
int main(){
  Deriv d;
  d.a=4;                                  //Deriv::a
  float y=d.fa(3.14);                     // Deriv::fa1
  return 0;
}
```

5.4.2. Đa kế thừa

Đa kế thừa là trường hợp mà một lớp kế thừa các thuộc tính từ hai hoặc nhiều hơn các lớp cơ sở, ví dụ:

```
class Base1{        // Base 1
public:
  int a;
  void fa1(){cout << "Base1 fa1" << endl;}
  char *fa2(int){cout << "Base1 fa2" << endl;return 0;}
};
class Base2{      // Base 2
public:
  int a;
  char *fa2(int, char){cout << "Base2 fa2" << endl;return 0;}
  int fc(){cout << "Base2 fc" << endl;return 0;}
};
class Deriv : public Base1 , public Base2{
public:
  int a;
  float fa1(float){cout << "Deriv fa1" << endl;return 1.0;}
  int fb1(int){cout << "Deriv fb1" << endl;return 0;}
};
int main(){
  Deriv d;
  d.a=4;                                  //Deriv::a
  d.Base2::a=5;                           //Base2::a
  float y=d.fa1(3.14);                    // Deriv::fa1
  int i=d.fc();                           // Base2::fc
  //char *c = d.fa2(1);                   // ERROR
  return 0;
}
```

Chú ý là câu lệnh char * c = d.fa2(1); là sai vì trong kế thừa các hàm không được overload mà chúng bị override chúng ta cần phải viết là: char * c = d.Base1::fa1(1); hoặc char * c = d.Base::fa2(1,"Hello");

5.4.3. Lặp lại lớp cơ sở trong đa kế thừa và lớp cơ sở ảo

Chúng ta hãy xét ví dụ sau:

```
class Gparent{};
class Mother: public Gparent{};
class Father: public Gparent{};
class Child: public Mother, public Father{};
```

Cả hai lớp Mother và Father đều kế thừa từ lớp Gparent và lớp Child kế thừa từ hai lớp Mother và Father. Hãy nhớ lại rằng mỗi đối tượng được tạo ra nhờ kế thừa đều chứa một đối tượng con của lớp cơ sở. Mỗi đối tượng của lớp Mother và Father đều chứa các đối tượng con của lớp Gparent và một đối tượng của lớp Child sẽ chứa các đối tượng con của hai lớp Mother và Father vì thế một đối tượng của lớp Child sẽ chứa hai đối tượng con của lớp Gparent, một được kế thừa từ lớp Mother và một từ lớp Father.

Đây là một trường hợp lạ vì có hai đối tượng con trong khi chỉ nên có 1.

Ví dụ giả sử có một phần tử dữ liệu trong lớp Gparent:

```
class Gparent{   protected: int gdata; };
```

Và chúng ta sẽ truy cập vào phần tử dữ liệu này trong lớp Child:

```
class Child: public Mother, public Father{
  public:
        void Cfunc(){        int item = gdata; // Sai        }
};
```

Trình biên dịch sẽ phàn nàn rằng việc truy cập tới phần tử dữ liệu gdata là mập mờ và lỗi. Nó không biết truy cập tới phần tử gdata nào: của đối tượng con Gparent trong đối tượng con Mother hay của đối tượng con Gparent trong đối tượng con Father.

Để giải quyết trường hợp này chúng ta sẽ sử dụng một từ khóa mới, **virtual**, khi kế thừa Mother và Father từ lớp Gparent:

```
class Gparent{};
class Mother: virtual public Gparent{};
class Father: virtual public Gparent{};
class Child: public Father, public Mother{};
```

Từ khóa virtual báo cho trình biên dịch biết là chỉ kế thừa duy nhất một đối tượng con từ một lớp trong các lớp dẫn xuất. Việc sử dụng từ khóa virtual giải quyết được vấn đề nhập nhằng trên song lại làm nảy sinh rất nhiều vấn đề khác.

Nói chung thì chúng ta nên tránh dùng đa kế thừa mặc dù có thể chúng ta đã là một chuyên gia lập trình C++, và nên suy nghĩ xem tại sao lại phải dùng đa kế thừa trong các trường hợp hiếm hoi thực sự cần thiết.

Để tìm hiểu kỹ hơn về đa kế thừa chúng ta có thể xem chương 6: đa kế thừa của sách tham khảo: ”Thinking in C++, 2nd Edition”.

5.5. Ràng buộc tĩnh, động

Một số đặc điểm của ràng buộc động

- Khi thiết kế một hệ thống thường các nhà phát triển hệ thống gặp một trong số các tình huống sau đây:

Hiểu rõ về các giao diện lớp mà họ muốn mà không hiểu biết chính xác về cách trình bày hợp lý nhất.

Hiểu rõ về thuật toán mà họ muốn sử dụng song lại không biết cụ thể các thao tác nào nên được cài đặt.

Trong cả hai trường hợp thường thì các nhà phát triển mong muốn trì hoãn một số các quyết định cụ thể càng lâu càng tốt. Mục đích là giảm các cố gắng đòi hỏi để thay đổi cài đặt khi đã có đủ thông tin để thực hiện một quyết định có tính chính xác hơn.

Vì thế sẽ rất tiện lợi nếu có một cơ chế cho phép trừu tượng hóa việc "đặt chỗ trước".

Che dấu thông tin và trừu tượng dữ liệu cung cấp các khả năng "place – holder" phụ thuộc thời điểm biên dịch và thời điểm liên kết. Ví dụ: các thay đổi về việc representation đòi hỏi phải biên dịch lại hoặc liên kết lại.

Ràng buộc động cho phép thực hiện khả năng "place – holder" một cách linh hoạt. Ví dụ trì hoãn một số quyết định cụ thể cho tới thời điểm chương trình được thực hiện mà không làm ảnh hưởng tới cấu trúc mã chương trình hiện tại.

- Ràng buộc động không mạnh bằng các con trỏ hàm nhưng nó mang tính tổng hợp hơn và làm giảm khả năng xuất hiện lỗi hơn vì một số lý do chẳng hạn trình biên dịch sẽ thực hiện kiểm tra kiểu tại thời điểm biên dịch.

- Ràng buộc động cho phép các ứng dụng có thể gọi tới các phương thức mang tính chung chung qua các con trỏ tới lớp cơ sở. Tại thời điểm chương trình thực hiện các lời gọi hàm này sẽ được chỉ định tới các phương thức cụ thể được cài đặt tại các lớp dẫn xuất thích hợp.

5.6. Hàm ảo

5.6.1. Các hàm thành viên bình thường được truy cập qua các con trỏ

Ví dụ đầu tiên sẽ cho chúng ta thấy điều gì sẽ xảy ra khi một lớp cơ sở và các lớp dẫn xuất đều có các hàm có cùng tên và các hàm này được truy cập thông qua các con trỏ nhưng không sử các hàm ảo (không phải đa thể).

```
class Teacher{ // Base class
String *name;
int numOfStudents;
public:
Teacher(const String &, int); // Constructor of base
void print() const{
cout << "Name: "<< name << endl;
cout << " Num of Students:"<< numOfStudents << endl;
}
};
void Teacher::print() const       // Non-virtual function{
  cout << "Name: "<< name << endl;
  cout << " Num of Students:"<< numOfStudents << endl;
    }
```

```
class Principal : public Teacher{ // Derived class
String *SchoolName;
public:
Principal(const String &, int , const String &);
void print() const{
teacher::print();
cout << " Name of School:"<< SchoolName << endl;
}
};
void Principal::print() const     // Non-virtual function{
  Teacher::print();
  cout << " Name of School:"<< SchoolName << endl;
}
int main(){
Teacher t1("Teacher 1",50);
Principal p1("Principal 1",40,"School");
Teacher *ptr;
char c;
cout << "Teacher or Principal "; cin >> c;
if (c=='t') ptr=&t1;
else ptr=&p1;
ptr->print(); // which print ??
}
```

Lớp Principal kế thừa từ lớp cơ sở Teacher. Cả hai lớp đều có hàm thành viên print(). Trong hàm main() chương trình tạo ra các đối tượng của hai lớp Teacher và Principal và một con trỏ trỏ tới lớp Teacher. Sau đó nó truyền địa chỉ của đối tượng thuộc lớp dẫn xuất vào con trỏ của lớp cơ sở bằng lệnh:

ptr = &p1; // địa chỉ của lớp dẫn xuất trong con trỏ trỏ tới lớp cơ sở.

Hãy nhớ rằng hoàn toàn hợp lệ khi thực hiện gán một địa chỉ của một đối tượng thuộc lớp dẫn xuất cho một con trỏ của lớp cơ sở, vì các con trỏ tới các đối tượng của một lớp dẫn xuất hoàn toàn tương thích về kiểu với các con trỏ tới các đối tượng của lớp cơ sở.

Bây giờ câu hỏi đặt ra là khi thực hiện câu lệnh:

ptr->print();

thì hàm nào sẽ được gọi tới? Là hàm print() của lớp dẫn xuất hay hàm print() của lớp cơ sở.

Hàm print() của lớp cơ sở sẽ được thực hiện trong cả hai trường hợp. Trình biên dịch sẽ bỏ qua nội dung của con trỏ ptr và chọn hàm thành viên khớp với kiểu của con trỏ.

5.6.2. Các hàm thành viên ảo được truy cập qua các con trỏ

Bây giờ chúng ta sẽ thay đổi một chút trong chương trình: đặt thêm từ khóa **virtual** trước khai báo của hàm print() trong lớp cơ sở.

```
class Teacher{   // Base class
  String name;
  int numOfStudents;
  public:
  Teacher(const String & new_name,int nos){ // Constructor of base
                    name=new_name;numOfStudents=nos;
  }
  virtual void print() const;                    // print is a virtual function
};
void Teacher::print() const       // virtual function
{
  cout << "Name: "<< name << endl;
  cout << " Num of Students:"<< numOfStudents << endl;
}
class Principal : public Teacher{                    // Derived class
  String SchoolName;
  public:
          Principal(const String & new_name,int nos, const String & sn)
                                  :Teacher(new_name,nos)      {
                  SchoolName=sn;
          }
          void print() const;
};
void Principal::print() const                        // Non-virtual function
{
  Teacher::print();
  cout << " Name of School:"<< SchoolName << endl;
}
int main(){
  Teacher t1("Teacher 1",50);
  Principal p1("Principal 1",40,"School");
  Teacher *ptr;
  char c;
```

```
  cout << "Teacher or Principal "; cin >> c;
  if (c=='t') ptr=&t1;
     else    ptr=&p1;
  ptr->print();                    // which print compare with example e81.cpp
  return 0;
}
```

Giờ thì các hàm khác nhau sẽ được thực hiện, phụ thuộc vào nội dung của con trỏ ptr. Các hàm được gọi dựa trên nội dung của con trỏ ptr, chứ không dựa trên kiểu của con trỏ. Đó chính là cách thức làm việc của đa thể. Chúng ta đã làm cho hàm print() trở thành đa thể bằng cách gán cho nó kiểu hàm ảo.

5.7. Đa thể và Ràng buộc động

5.7.1. Đa thể (Polymorphism)

Trong lập trình hướng đối tượng có 3 khái niệm chính là: Các lớp; Kế thừa; Đa thể, được cài đặt trong ngôn ngữ C++ bằng các hàm ảo.

Trong cuộc sống thực tế, thường có một tập các đối tượng khác nhau có các chỉ thị (instruction) (message) giống nhau, nhưng lại thực hiện các hành động khác nhau. Ví dụ như là hai lớp các đối tượng giáo viên và hiệu trưởng trong một trường học.

Giả sử ông bộ trưởng bộ giáo dục muốn gửi một chỉ thị xuống cho tất cả các nhân viên "In thông tin cá nhân của ông ra và gửi cho tôi". Các loại nhân viên khác nhau của bộ giáo dục (giáo viên bình thường và hiệu trưởng) sẽ in ra các thông tin khác nhau. Nhưng ông bộ trưởng không cần gửi các thông điệp khác nhau cho các nhóm nhân viên khác nhau của ông ta. Chỉ cần một thông điệp cho tất cả các nhân viên vì tất cả các nhân viên đều biết in ra thông tin hay lý lịch cá nhân của mình như thế nào.

Đa thể (polymorphism) có nghĩa là "take many shapes". Câu lệnh hay chỉ thị đơn của bộ trưởng chính là một trường hợp đa thể vì nó sẽ có dạng khác nhau đối với các loại nhân lực khác nhau.

Thường thường đa thể xảy ra trong các lớp có mối liên hệ kế thừa lẫn nhau. Trong C++ đa thể có nghĩa là một lời gọi tới một hàm thành viên sẽ tạo ra một hàm khác nhau để thực hiện phụ thuộc vào loại đối tượng có hàm thành viên được gọi tới.

Điều này nghe có vẻ giống như là overload hàm, nhưng thực ra không phải, đa thể mạnh hơn là chồng hàm về mặt kỹ thuật. Một sự khác nhau giữa đa thể và chồng hàm đó là cách thức lựa chọn hàm để thực hiện.

Với chồng hàm sự lựa chọn được thực hiện bởi trình biên dịch vào thời điểm biên dịch. Với đa thể việc lựa chọn hàm để thực hiện được thực hiện khi chương trình đang chạy.

5.7.2. Ràng buộc động

Ràng buộc động hay ràng buộc muộn là một khái niệm gán liền với khái niệm đa thể. Chúng ta hãy xem xét câu hỏi sau: làm thế nào trình biên dịch biết được hàm nào để biên dịch? Trong ví dụ trước trình biên dịch không có vấn đề gì khi nó gặp câu lệnh: ptr->print();

Câu lệnh này sẽ được biên dịch thành một lời gọi tới hàm print() của lớp cơ sở. Nhưng trong ví dụ sau trình biên dịch sẽ không nội dung của lớp nào được ptr trỏ tới. Đó có thể là nội dung của một đối tượng thuộc lớp Teacher hoặc lớp Principal. Phiên bản nào của hàm print() sẽ được gọi tới? Trên thực tế tại thời điểm biên dịch chương trình trình biên dịch sẽ không biết làm thế nào vì thế nó sẽ sắp xếp sao cho việc quyết định chọn hàm nào để thực hiện được trì hoãn cho tới khi chương trình thực hiện.

Tại thời điểm chương trình được thực hiện khi lời gọi hàm được thực hiện mã mà trình biên dịch đặt vào trong chương trình sẽ tìm đúng kiểu của đối tượng mà địa chỉ của nó được lưu trong con trỏ ptr và gọi tới hàm print() thích hợp của lớp Teacher hay của lớp Principal phụ thuộc vào lớp của đối tượng.

Chọn lựa một hàm để thực hiện tại thời điểm chương trình thực hiện được gọi là ràng buộc muộn hoặc ràng buộc động (Binding có nghĩa là kết nối lời gọi hàm với hàm). Kết nối các hàm theo cách bình thường, trong khi biên dịch, được gọi là ràng buộc trước (early binding) hoặc ràng buộc tĩnh (static binding). Ràng buộc động đòi hỏi chúng ta cần xài sang hơn một chút (lời gọi hàm đòi hỏi khoảng 10 phần trăm mã hàm) nhưng nó cho phép tăng năng lực cũng như sự linh hoạt của các chương trình lên gấp bội.

Ràng buộc động làm việc như thế nào

Hãy nhớ lại rằng, lưu trữ trong bộ nhớ, một đối tượng bình thường – không có hàm thành viên ảo chỉ chứa các thành phần dữ liệu của chính nó ngoài ra không có gì khác. Khi một hàm thành viên được gọi tới với một đối tượng nào đó trình biên dịch sẽ truyền địa chỉ của đối tượng cho hàm. Địa chỉ này là luôn sẵn sàng đối với các hàm thông qua con trỏ this, con trỏ được các hàm sử dụng để truy cập vào các thành viên dữ liệu của các đối tượng trong phần cài đặt của hàm. Địa chỉ này thường được sinh bởi trình biên dịch mỗi khi một hàm thành viên được gọi tới; nó không được chứa trong đối tượng và không chiếm bộ nhớ. Con trỏ this là kết nối duy nhất giữa các đối tượng và các hàm thành viên bình thường của nó.

Với các hàm ảo, công việc có vẻ phức tạp hơn đôi chút. Khi một lớp dẫn xuất với các hàm ảo được chỉ định, trình biên dịch sẽ tạo ra một bảng – một mảng – các địa chỉ hàm được gọi là bảng ảo. Trong ví dụ sau các lớp Teacher và Principal đều có các bảng hàm ảo của riêng chúng. Có một entry (lối vào) trong mỗi bảng hàm ảo cho mỗi một hàm ảo của lớp. Các đối tượng của các lớp với các hàm ảo chứa một con trỏ tới bảng hàm ảo của lớp. Các đối tượng này lớn hơn đôi chút so với các đối tượng bình thường.

Trong ví dụ khi một hàm ảo được gọi tới với một đối tượng của lớp Teacher hoặc Principal trình biên dịch thay vì chỉ định hàm nào sẽ được gọi sẽ tạo ra mã trước hết tìm bảng hàm ảo của đối tượng và sau đó sử dụng bảng hàm ảo đó để truy cập vào địa chỉ hàm thành viên thích hợp. Vì thế đối với các hàm ảo đối tượng tự nó quyết định xem hàm nào được gọi thay vì giao công việc này cho trình biên dịch.

Ví dụ: Giả sử các lớp Teacher và Principal chứa hai hàm ảo:

```
class Principal : public Teacher{ // Derived class
tring *SchoolName;
public:
void read(); // Virtual function
```

void print() const; // Virtual function

};

class Teacher{ // Base class

String *name;

int numOfStudents;

public:

virtual void read(); // Virtual function

virtual void print() const; // Virtual function

};

Khi đó ta có các bảng hàm ảo sau:

Bảng hàm ảo của lớp Teacher	Bảng hàm ảo của lớp Principal
&Teacher::read	&Principal::read
&Teacher::print	&Principal::print

Các đối tượng của lớp Teacher và Principal sẽ chứa một con trỏ tới các bảng hàm ảo của chúng.

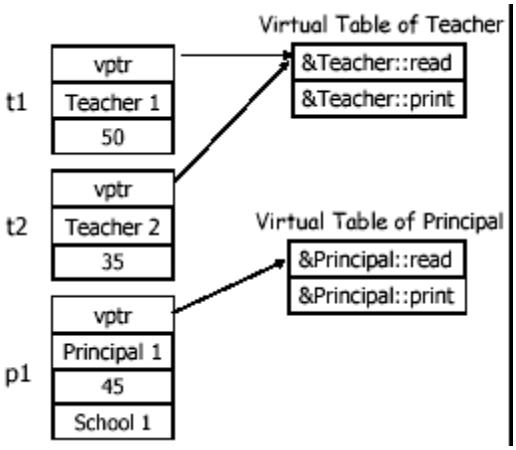
int main(){

Teacher t1("Teacher 1", 50);

Teacher t2("Teacher 2", 35);

Principal p1("Principal 1", 45 , "School 1");

}



Cần ghi nhớ rằng kỹ thuật hàm ảo chỉ làm việc với các con trỏ trỏ tới các đối tượng và với các tham chiếu, chứ không phải bản thân các đối tượng.

int main(){

Teacher t1("Teacher 1",50);

Principal p1("Principal 1",40,"School");

t1.print(); // not polymorphic

```
p1.print(); // not polymorphic
return 0;
}
```

Việc gọi tới các hàm ảo hơi mất thời gian đôi chút vì đó thực chất là việc gọi gián tiếp thông qua bảng hàm ảo. Không nên khai báo các hàm là ảo nếu không cần thiết.

Danh sách liên kết các đối tượng và đa thể

Các cách thức chung nhất để sử dụng các hàm ảo là với một mảng các con trỏ trỏ tới các đối tượng và các danh sách liên kết các đối tượng.

Chúng ta xem xét ví dụ sau đây:

Ví dụ 7.3:

```
class Teacher{   // Base class
  String name;
  int numOfStudents;
  public:
  Teacher(const String & new_name,int nos){ // Constructor of base
                    name=new_name;numOfStudents=nos;
  }
  virtual void print() const;                // print is a virtual function
};
void Teacher::print() const                  // virtual function
{
  cout << "Name: "<< name << endl;
  cout << " Num of Students:"<< numOfStudents << endl;
}
class Principal : public Teacher{          // Derived class
  String SchoolName;
  public:
          Principal(const String & new_name,int nos, const String & sn)
                                    :Teacher(new_name,nos){
                  SchoolName=sn;
          }
          void print() const;
};
void Principal::print() const      // Non-virtual function
{
  Teacher::print();
```

```
  cout << " Name of School:"<< SchoolName << endl;
}
// *** A class to define nodes of the list ***
class List_node{
  friend class List;
  const Teacher * element;
  List_node * next;
  List_node(const Teacher &);  // constructor
};
List_node::List_node(const Teacher & n){
  element = &n;
  next = 0;
}
// *** class to define a linked list of teachers and principals ***
class List{   // linked list for teachers
   List_node *head;
  public:
   List(){head=0;}
   Bool append(const Teacher &);
   void print() const ;
   ~List();
};
// Append a new teacher to the end of the list
// if there is no space returns False, otherwise True
Bool List::append(const Teacher & n){
  List_node *previous, *current;
  if(head)       // if the list is not empty
  {
            previous=head;
            current=head->next;
            while(current)            // searh for the end of the list
            {
                     previous=current;
                     current=current->next;
            }
            previous->next = new List_node(n);
```

```
            if (!(previous->next)) return False;              // If memory is full
  }
  else            // if the list is empty
  {
            head = new List_node(n);       // Memory for new node
            if (!head) return False;           // If memory is full
  }
  return True;
}
// Prints all elements of the list on the screen
void List::print() const{
  List_node *tempPtr;
  if (head)          {
    tempPtr=head;
    while(tempPtr){
                  (tempPtr->element)->print();            // POLYMORPHISM
                  tempPtr=tempPtr->next;
            }
  }
  else       cout << "The list is empty" << endl;
}
// Destructor
// deletes all elements of the list
List::~List(){
  List_node *temp;
  while(head)        // if the list is not empty
  {
            temp=head;
            head=head->next;
            delete temp;
  }
}
// ----- Main Function -----
int main(){
  Teacher t1("Teacher 1",50);
  Principal p1("Principal 1",40,"School1");
```

```
  Teacher t2("Teacher 2",60);
  Principal p2("Principal 2",100,"School2");
  List  theList;
  theList.print(); theList.append(t1); theList.append(p1); theList.append(t2);
  theList.append(p2); theList.print();
  return 0;
}
```

Các lớp trừu tượng

Để viết các hàm đa thể chúng ta cần phải có các lớp dẫn xuất. Nhưng đôi khi chúng ta không cần phải tạo ra bất kỳ một đối tượng thuộc lớp cơ sở nào cả. Lớp cơ sở tồn tại chỉ như là một điểm khởi đầu cho việc kế thừa của các lớp khác. Kiểu lớp cơ sở như thế được gọi là một lớp trừu tượng, có nghĩa là không có một đối tượng thực sự nào của lớp được tạo ra từ lớp đó.

Các lớp trừu tượng làm nảy sinh rất nhiều tình huống mới. Một nhà máy rất nhiều xe thể thao hoặc một xe tải hoặc một xe cứu thương, nhưng nó không thể tạo ra một chiếc xe chung chung nào đó. Nhà máy phải biết loại xe nào mà nó cần tạo ra trước khi thực sự tạo ra nó. Tương tự chúng ta có thể thấy sparrow (chim sẻ), wren (chim hồng tước), robin (chim két cổ đỏ) nhưng chúng ta không thể thấy một con chim chung chung nào đó.

Thực tế một lớp sẽ là một lớp ảo chỉ trong con mắt của con người. Trình biên dịch sẽ bỏ qua các quyết định của chúng ta về việc biến một lớp nào đó thành lớp ảo.

Các hàm ảo thực sự

Sẽ là tốt hơn nếu, đã quyết định tạo ra một lớp trừu tượng cơ sở, chúng ta có thể (hướng dẫn) (instruct) chỉ thị cho trình biên dịch ngăn chặn một cách linh động bất cứ người nào sao cho họ không thể tạo ra bất cứ đối tượng nào của lớp đó. Điều này sẽ cho phép chúng ta tự do hơn trong việc thiết kế lớp cơ sở vì chúng ta sẽ không phải lập kế hoạch cho bất kỳ đối tượng thực sự nào của lớp đó, mà chỉ cần quan tâm tới các dữ liệu và hàm sẽ được sử dụng trong các lớp dẫn xuất. Có một cách để báo cho trình biên dịch biết một lớp là trừu tượng: chúng ta định nghĩa ít nhất một hàm ảo thực sự trong khai báo lớp.

Một hàm ảo thực sự là một hàm ảo không có thân hàm. Thân của hàm ảo trong lớp cơ sở sẽ được loại bỏ và ký pháp =0 sẽ được thêm vào khai báo hàm:

Ví dụ:

```
class generic_shape{ // Abstract base class
protected: int x,y;
public: generic_shape(int x_in,int y_in){ x=x_in; y=y_in;} // Constructor
virtual void draw() const =0; //pure virtual function
};
class Line:public generic_shape{ // Line class
protected: int x2,y2; // End coordinates of line
public:
```

```
Line(int x_in,int y_in,int x2_in,int y2_in):generic_shape(x_in,y_in){
x2=x2_in; y2=y2_in;
}
void draw() const { line(x,y,x2,y2); } // virtual draw function
};
//line là một hàm thư viện vẽ một đường thẳng lên màn hình
class Rectangle:public Line{ // Rectangle class
public:
Rectangle(int x_in,int y_in,int x2_in,int y2_in):Line(x_in,y_in,x2_in,y2_in){}
void draw() const { rectangle(x,y,x2,y2); } // virtual draw
};
class Circle:public generic_shape{ // Circle class
protected: int radius;
public:
Circle(int x_cen,int y_cen,int r):generic_shape(x_cen,y_cen){
radius=r;
}
void draw() const { circle(x,y, radius); } // virtual draw
};
//rectangle và circle là các hàm thư viện vẽ các hình chữ nhật và hình tròn lên màn hình
int main(){
Line Line1(1,1,100,250);
Circle Circle1(100,100,20);
Rectangle Rectangle1(30,50,250,140);
Circle Circle2(300,170,50);
show(Circle1); // show function can take different shapes as argument
show(Line1); show(Circle2); show(Rectangle1);
return 0;
}
// hàm vẽ các hình khác nhau
void show(generic_shape &shape)
{ // Which draw function will be called?
shape.draw(); // It 's unknown at compile-time
}
```

Nếu chúng ta viết một lớp cho một hình mới bằng cách kế thừa nó từ các lớp đã có chúng ta không cần phải thay đổi hàm show. Hàm này có thể thực hiện chức năng với các lớp mới.

Ví dụ:

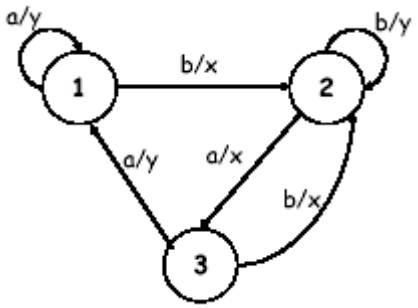
Trong ví dụ này chúng ta sẽ xem xét một "Máy trạng thái hữu hạn" (Finite State Machine) FSM.

Chúng ta có các trạng thái: {1, 2, 3}

Input: {a, b}, x để thoát

Output: {x, y}

Các trạng thái của FSM được định nghĩa bằng cách sử dụng một cấu trúc lợp. Mỗi trạng thái sẽ được kế thừa từ lớp cơ sở.




```
// A Finite State Machine with 3 states
#include<iostream>
// *** Base State (Abstract Class) ***
class State{
 protected:
  State * const next_a, * const next_b; // Pointers to next state
  char output;
 public:
  State( State &  a,   State &  b):next_a(&a),next_b(&b){}
  virtual State* transition(char)=0;
};
// *** State1 ***
class State1:public State{
 public:
  State1( State &  a,   State &  b):State(a,b){}
  State* transition(char);
};
// *** State2 ***
class State2:public State{
```

```
 public:
  State2( State &  a,  State &  b):State(a,b){}
  State* transition(char);
};
/*** State3 ***/
class State3:public State{
 public:
  State3( State &  a,  State &  b):State(a,b){}
  State* transition(char);
};
State* State1::transition(char input)
{
  cout << endl << "Current State: 1";
  switch(input){
  case 'a': output='y';
                              cout << endl << "Output: "<< output;
                              cout << endl << "Next State: 1";
                              return next_a;
  case 'b': output='x';
                              cout << endl << "Output: "<< output;
                              cout << endl << "Next State: 2";
                              return next_b;
  default : cout << endl << "Undefined input";
                              cout << endl << "Next State: Unchanged";
                              return this;
  }
}
State* State2::transition(char input)
{
  cout << endl << "Current State: 2";
  switch(input){
  case 'a': output='x';
                              cout << endl << "Output: "<< output;
                              cout << endl << "Next State: 3";
                              return next_a;
  case 'b': output='y';
```

```
                        cout << endl << "Output: "<< output;
                        cout << endl << "Next State: 2";
                        return next_b;
  default : cout << endl << "Undefined input";
                        cout << endl << "Next State: Unchanged";
                        return this;
 }
}
State* State3::transition(char input)
{
 cout << endl << "Current State: 3";
 switch(input){
 case 'a': output='y';
                    cout << endl << "Output: "<< output;
                    cout << endl << "Next State: State1";
                    return next_a;
 case 'b': output='x';
                    cout << endl << "Output: "<< output;
                    cout << endl << "Next State: 2";
                    return next_b;
 default : cout << endl << "Undefined input";
                    cout << endl << "Next State: Unchanged";
                    return this;
 }
}
// *** Finite State Machine ***
// This class has 3 State objects as members
class FSM{
  State1 s1;
  State2 s2;
  State3 s3;
  State * current;
 public:
  FSM():s1(s1,s2),s2(s3,s2),s3(s1,s2),current(&s1)      {}
  void run();
};
```

```
void FSM::run(){
  char in;
  cout << endl << "The finite state machine starts ...";
  do{
    cout << endl << "Give the input value (a or b; x:EXIT) ";    cin >> in;
    if (in != 'x')
      current = current->transition(in);
    else
            current = 0;                    // EXIT
  }while(current);
  cout << endl << "The finite state machine stops ..." << endl;;
}
int main(){
  FSM machine1;
  machine1.run();
  return 0;
}
```

Hàm transition của mỗi trạng thái xác định hành vi của FSM. Nó nhận giá trị input như là tham số, kiểm tra input sinh ra giá trị output tùy thuộc vào giá trị input và trả về địa chỉ của trạng thái tiếp theo.

Hàm chuyển đổi (transition)của trạng thái hiện tại được gọi. Giá trị trả về của hàm này sẽ xác định trạng thái tiếp theo của FSM.

Cấu tử ảo và hủy tử ảo

Khi chúng ta tạo ra một đối tượng chúng ta thường là đã biết kiểu của đối tượng đang được tạo ra và chúng ta có thể chỉ định điều này cho trình biên dịch. Vì thế chúng ta không cần có các cấu tử ảo.

Cũng như vậy một hàm cấu tử của một đối tượng thiết lập cơ chế ảo của nó (bảng hàm ảo) trước tiên. Chúng ta không nhìn thấy đoạn mã chương trình này, tất nhiên, cũng như chúng ta không nhìn thấy đoạn mã khởi tạo vùng nhớ cho một đối tượng.

Các hàm ảo không thể thậm chí tồn tại trừ khi hàm cấu tử hòan thành công việc của nó vì thế các hàm cấu tử không thể là các hàm ảo.

Hàm hủy tử ảo:

Ví dụ:

```
// non-virtual function used as base class destructor
#include <iostream>
class Base{
public:
```

```
        ~Base() { cout << "Base destructor" << endl; }   // Destructor is not virtual
};
class Derived : public Base{
public:
        ~Derived() { cout << "Derived destructor" << endl; } // Non-virtual
};
int main(){
  Base* pb;               // pb can point to objects of Base ans Derived
pb = new Derived;         // pb points to an oject of Derived
delete pb;
cout << "Program terminates" << endl;
return 0;
}
```

Hãy nhớ lại rằng một đối tượng của lớp dẫn xuất thường chứa dữ liệu từ cả lớp cơ sở và lớp dẫn xuất. Để đảm bảo rằng các dữ liệu này được goodbye một cách hoàn hảo có thể cần có những lời gọi tới hàm hủy tử cho cả lớp cơ sở và lớp dẫn xuất. Nhưng output của ví dụ trên là:

Base Destructor

Program terminates

Trong chương trình này bp là một con trỏ của lớp cơ sở (kiểu Base). Vì thế nó có thể trỏ tới các đối tượng thuộc lớp Base và Derived. Trong ví dụ trên bp trỏ tới một đối tượng thuộc lớp Derived nhưng trong khi xóa con trỏ này chỉ hàm hủy tử của lớp Base là được gọi tới.

Vấn đề tương tự cũng đã được bắt gặp với các hàm bình thường trong phần trước (các hàm không là hàm hủy tử). Nếu như một hàm không phải là hàm ảo chỉ có phiên bản của lớp cơ sở là được gọi tới thậm chí nếu nội dung của con trỏ là một địa chỉ của một đối tượng của lớp dẫn xuất. Vì thế trong ví dụ trên hàm hủy tử của lớp Derived sẽ không bao giờ được gọi tới. Điều này có thể là một tai họa nếu như hàm này thực hiện một vài công việc cao thủ nào đó. Để sửa chữa chúng ta chỉ cần làm cho hàm hủy tử này trở thành hàm ảo và thế là mọi thứ sẽ trở lại bình thường.

Hàm toán tử ảo

Như chúng ta đã thấy khái niệm về ràng buộc động có nghĩa là kiểu động (dynamic type) của đối tượng thực sự sẽ quyết định hàm nào sẽ được gọi thực hiện. Nếu chẳng hạn chúng ta thực hiện lời gọi: p->f(x) trong đó p là một con trỏ x là tham số và f là một hàm ảo, thì chính là kiểu của đối tượng mà p trỏ tới sẽ xác định biến thể (variant) nào của hàm f sẽ được gọi thực hiện. Các toán tử cũng là các hàm thành viên nếu như chúng ta có một biểu thức: a XX b

Trong đó XX là một ký hiệu toán tử thì điều này cũng giống như chúng ta có câu lệnh sau: a.operatorXX(b);

Nếu như hàm toán tử operatorXX được khai báo là một hàm ảo thì chúng ta sẽ thực hiện lời gọi hàm như sau: (*p) XX (*q);

Trong đó p và q là hai con trỏ và khi đó kiểu của đối tượng mà p trỏ tới sẽ xác định biến thể nào của hàm toán tử operatorXX sẽ được gọi tới để thực hiện. Việc con trỏ q trỏ tới đối tượng thuộc lớp nào là không quan trọng, nhưng C++ chỉ cho phép ràng buộc động đối với toán tử đầu tiên. Điều này có thể làm nảy sinh rắc rối nếu chúng ta muốn viết các toán tử đối xứng với hai tham số có cùng kiểu. Chẳng hạn giả sử chúng ta muốn xây dựng toán tử > để so sánh hai đối tượng cùng thuộc kiểu Teacher (có nghĩa là có thể là thuộc lớp Teacher hoặc lớp Principal) trong các ví dụ trước. Nếu chúng ta có hai con trỏ t1 và t2:

Teacher *t1, * t2;

Chúng ta muốn rằng có thể thực hiện so sánh bằng câu lệnh sau:

if(*t1 > *t2)

….

Có nghĩa là chúng ta muốn thực hiện ràng buộc động đối với hàm toán tử > sao cho có thể gọi tới các biến thể khác nhau của nó tùy thuộc vào kiểu của các đối tượng mà hai con trỏ t1 và t2 trỏ tới. Ví dụ nếu chúng cùng trỏ tới các đối tượng thuộc lớp Teacher chúng ta có thể so sánh theo hai điều kiện là tên và số sinh viên quản lý, còn nếu chúng cùng trỏ tới hai đối tượng thuộc lớp Principal chúng ta sẽ so sánh thêm tiêu chí thứ 3 là tên trường.

Để có thể thực hiện điều này theo nguyên tắc của ràng buộc động chúng ta sẽ khai báo hàm toán tử > là hàm ảo ở lớp cơ sở (Teacher):

```
class Teacher{
  ..
  pubic:
        virtual Bool operator > (Teacher & rhs);
};
```

Sau đó trong lớp dẫn xuất chúng ta có khai báo tiếp như sau:

```
class Principal: public Teacher{
  ..
  pubic:
        Bool operator > (Principal & rhs);
};
```

Theo nguyên lý thông thường về các hàm ảo chúng ta mong muốn là toán tử > sẽ hoạt động tốt với các đối tượng thuộc lớp Principal (hay chính xác hơn là các con trỏ trỏ vào các đối tượng thuộc lớp đó). Tuy nhiên thật không may đây lại là một lỗi, định nghĩa lớp như thế sẽ không thể biên dịch được, thâm chí ngay cả trong trường hợp mà có thể biên dịch được thì chúng ta cũng không thể sinh bất cứ đối tượng nào thuộc lớp Principal vì sẽ làm xuất hiện lỗi biên dịch. Lý do là vì trình biên dịch hiểu rằng lớp Principal là một lớp trừu tượng và nguyên nhân là ở tham số trong khai báo của hàm toán tử >. Điều này là vì khai báo hàm toán tử trong lớp Principal không khớp với kiểu tham số trong khai báo của lớp cơ sở Teacher do đó hàm

toán tử > của lớp cơ sở Teacher sẽ không được kế thừa hay bị ẩn đi và điều này có nghĩa là lớp Principal vẫn có một hàm toán tử > là hàm ảo thực sự do đó nó là lớp ảo.

Để sửa chữa lỗi này chúng ta sẽ thực hiện khai báo lại như sau:

```
class Principal: public Teacher{
  ..
  pubic:
        Bool operator > (Teacher & rhs);
};
```

Bây giờ thì trình biên dịch không kêu ca phàn nàn gì nữa và chúng ta xem xét phần cài đặt hàm:

```
Bool Principal::operator > (Teacher & rhs){
  if(name > rhs.name)    return True;
  else if((name == rhs.name) && (numOfStudents > rhs.numOfStudents)) return True;
  else if((name == rhs.name) && (numOfStudents == rhs.numOfStudents) && (schoolName > rhs.schoolName))      return True;
  return False;
};
```

Tuy nhiên ở đây chúng ta lại gặp phải lỗi biên dịch. Trình biên dịch sẽ kêu ca rằng thành viên schoolName không phải là một thành viên của lớp Teacher. Điều này là chính xác và để khắc phục nó chúng ta cần thực hiện một thao tác chuyển đổi kiểu (casting) cho tham số truyền vào của hàm toán tử >:

```
Bool Principal::operator > (Teacher & rhs){
Principal & r = Dynamic_cast<Principal&>(rhs);
if(name > r.name) return True;
else if((name == r.name) && (numOfStudents > r.numOfStudents))  return True;
else if((name == r.name) && (numOfStudents == r.numOfStudents) && (schoolName > r.schoolName)) return True;
  return False;
};
```

Chúng ta cũng thực hiện hoàn toàn tương tự với các toán tử khác hay các đối tượng khác có vai trò tương tự như lớp Principal.

Bài tập

1. Xây dựng lớp Nhân viên, lớp Cán bộ có sử dụng kỹ thuật thừa kế
2. Xây dựng lớp Hình chữ nhật, lớp Tam giác thừa kế từ lớp cơ sở Hình
3. Xây dựng lớp Thí sinh và lớp Thí sinh ưu tiên có sử dụng kỹ thuật thừa kế

CHƯƠNG 6. BẢN MẪU (TEMPLATE)

6.1. Khái niệm bản mẫu

6.1.1. Các bản mẫu lớp

Khi viết các chương trình chúng ta luôn có nhu cầu sử dụng các cấu trúc có khả năng lưu trữ và xử lý một tập các đối tượng nào đó. Các đối tượng này có thể cùng kiểu – khi đó chúng ta có tập các đối tượng đồng nhất, hoặc chúng có thể có kiểu khác nhau khi đó ta có các tập đối tượng không đồng nhất hay hỗn hợp. Để xây dựng lên các cấu trúc đó chúng ta có thể sử dụng mảng hay các cấu trúc dữ liệu chẳng hạn như danh sách, hàng đợi, hoặc là cây. Một lớp có thể được dùng để xây dựng nên các collection object được gọi là một lớp chứa. Các lớp Stack, Queue hoặc Set đều là các ví dụ điển hình về lớp chứa. Vấn đề với các lớp chứa mà chúng ta đã biết này là chúng được xây dựng chỉ để chứa các đối tượng kiểu cơ bản (int, char *, …). Như vậy nếu chúng ta muốn xây dựng một hàng đợi chẳng hạn để chứa các đối tượng thuộc lớp Person chẳng hạn thì lớp Queue này lại không thể sử dụng được, giải pháp là chúng ta lại xây dựng một lớp mới chẳng hạn là Person_Queue. Đây là một phương pháp không hiệu quả và nó đòi hỏi chúng ta phải xây dựng lại hoàn toàn các cấu trúc mới với các kiểu dữ liệu (lớp) mới.

C++ cung cấp một khả năng cho phép chúng ta không phải lặp lại công việc tạo mới này bằng cách tạo ra các lớp chung. Một bản mẫu sẽ được viết cho lớp, và trình biên dịch sẽ tự động sinh ra các lớp khác nhau cần thiết từ bản mẫu này. Lớp chứa cần sử dụng sẽ chỉ phải viết một lần duy nhất. Ví dụ nếu chúng ta có một lớp bản mẫu là List đã được xây dựng xong thì trình biên dịch có thể, bằng cách sử dụng thông tin này, sinh ra lớp List<int> và List<Person>, tương ứng là một danh sách các số nguyên và danh sách các đối tượng của lớp Person.

Chúng ta cũng có thể xây dựng lên các hàm chung. Nếu chẳng hạn chúng ta muốn viết một hàm để sắp xếp một mảng, chúng ta có thể xây dựng một bản mẫu hàm. Trình biên dịch sẽ sử dụng các thông tin này để sinh ra các hàm khác nhau có khả năng sắp xếp các mảng khác nhau.

Các bản mẫu không luôn là một phần của C++. Kế thừa được sử dụng để để xây dựng các object collection. Nếu chúng ta muốn xây dựng một danh sách các đối tượng Person chúng ta sẽ trừu tượng hóa cho lớp Person là một lớp dẫn xuất của một lớp được định nghĩa trước Listable. Lớp Person sau đó sẽ kế thừa khả năng có thể là một phần của danh sách. Nhưng chúng ta se có vấn đề nếu như lớp Person cần một vài thuộc tính của riêng nó. Có thể chúng ta muốn là nó sẽ chứa trong một cấu trúc cây nào đó hoặc có thể xuất các thông tin ra màn hình qua tóan tử << của cout chẳng hạn. Để giải quyết các vấn đề như kiểu trên khái niệm đa kế thừa đã được đưa ra, đây là một khái niệm có thể mang lại rất nhiều khả năng mạnh mẽ nhưng cũng làm nẩy sinh rất nhiều vấn đề rắc rối. Một trong những khó khăn khi sử dụng kế thừa để xây dựng các objetc collection đó là chúng ta cần phải quyết định trước cách thức hoạt động và các tài nguyên của một lớp cụ thể nào đó. Ví dụ như với lớp Person chẳng hạn chúng ta cần phải biết là chúng ta sẽ sử dụng nó trong các danh sách hay các cây, có khả năng io như thế nào…Ngoài ra các lập trình viên cần phải biết rõ chi tiết về cách thức kế thừa của các lớp để có thể biết được các thuộc tính mà nó sẽ có.

Và hóa ra việc sử dụng các lớp chung là một cách tốt, tốt hơn so với việc sử dụng kế thừa trong việc xây dưng các object collection. Viết các chương trình hướng đối tượng không phải có nghĩa là lúc nào chúng ta cũng phải sử dụng kế thừa. Điều này rất rõ ràng nếu chúng ta xem xét tình huống xây dựng một lớp Person và một lớp danh sách cơ sở, lớp Person kế thừa từ lớp danh sách cơ sở đó (như ví dụ trong chương 6 về phần kế thừa) nhưng sẽ là tự nhiên nếu chúng ta xây dựng một lớp danh sách riêng để chứa các đối tượng thuộc lớp Person. Và với khái niệm bản mẫu chúng ta có một lớp List chung, để chứa các đối tượng thuộc lớp Person chúng ta chỉ cần khai báo một đối tượng thuộc lớp List<Person>.

Khái niệm bản mẫu được đưa vào khá muộn trong ngôn ngữ C++. Có rất nhiều lớp chứa trong bản phác thảo chuẩn. Chúng đều là các lớp chung sử dụng các bản mẫu. Các lớp chứa đều không sử dụng khái niệm kế thừa. Ngoài ra còn có một tập các hàm chung trong bản phác thảo chuẩn có thể thực hiện các thuật toán trên các object collection. Chẳng hạn như các hàm tìm kiếm và các hàm sắp xếp.

Trong chương này chúng ta sẽ xem xét cách thức các bản mẫu được sử dụng để xây dựng các lớp chung và các hàm. Thật không may là cú pháp của bản mẫu không được dễ hiểu và đẹp lắm trong C++ mặc dù vậy về bản chất các bản mẫu là những thứ rất có ích.

Trong đặc tả chuẩn của ngôn ngữ C++ có rất nhiều các lớp chứa đã được chuẩn hóa, hay được coi là chuẩn (1 phần cơ bản của ngôn ngữ). Chẳng hạn như các lớp List, Set, Queue, Stack .... Sinh viên có thể xem trong phần Helf của các chương trình dịch.

6.1.2. Các bản mẫu hàm

Chúng ta có thể xây dựng các hàm chồng, là các hàm trùng tên nhưng khác nhau tham số. Tuy nhiên, việc xây dựng các hàm chồng gặp bất lợi là ta phải xây dựng quá nhiều hàm khác nhau cho cùng một mục đích nào đó.

Ví dụ: Xây dựng hàm tìm phần tử lớn nhất

int Max(int a, int b); //tìm phần tử lớn nhất giữa 2 số nguyên

int Max(int a, int b, int c); //tìm phần tử lớn nhất giữa 3 số nguyên

float Max(float a, float b); //tìm phần tử lớn nhất giữa 2 số thực

float Max(float a, float b, float c); //tìm phần tử lớn nhất giữa 3 số thực

float Max(float a[], int n); //tìm phần tử lớn nhất của dãy số thực

int Max(int a[], int n); //tìm phần tử lớn nhất của dãy số nguyên

…..

Giải quyết vấn đề này ta có thể các bản mẫu hàm.

6.2. Các bản mẫu hàm

Các định nghĩa và thể nghiệm

Các lớp không phải là những thứ duy nhất có thể làm bản mẫu. Các hàm cũng có thể là các bản mẫu, khi đó chúng được gọi là các hàm bản mẫu.

Ví dụ 1: Trong các chương trình chúng ta thường hay phải thực hiện việc so sánh hai biến thuộc cùng một kiểu xem biến nào lớn hơn, thay vì thực hiện một câu lệnh if chúng ta có thể viết một hàm trả về phần tử nhỏ hơn ví dụ:

int min(int a, int b){ return (a<b)?a:b; }

Việc dùng hàm min này không có gì đáng nói nhưng nếu chúng ta muốn sử dụng hàm min với hai biến không phải kiểu int thì rắc rối to. Và trong trường hợp đó chúng ta muốn sử dụng bản mẫu hàm:

template <class T>

const T & min(const T & a, const T & b){ return (a<b)?a:b; }

cũng giống như các lớp bản mẫu trong các hàm bản mẫu cũng có sự thay thế kiểu khi sử dụng chúng với các tình huống cụ thể:

int a, y;

long int m, n;

...

cout << min(x,y);

cout << min(m,n);

tương ứng với dòng thứ nhất trình biên dịch sẽ sinh ra một thể nghiệm của hàm và thay thế T bởi kiểu int, dòng thứ hai là một thể nghiệm với T được thay bằng long int.

Một thể nghiệm của một bản mẫu hàm không nhất thiết phải được tạo ra một cách tường minh. Nó sẽ được tự động sinh ra khi có một lời gọi hàm. Trình biên dịch sẽ cố gắng khớp các tham số thực sự của hàm với các tham số hình thức và xác định kiểu mà các tham số kiểu chung sẽ nhận. Nếu chúng có thể khớp, trình biên dịch sẽ kiểm tra xem đã có một thể nghiệm nào cho các tham số thực sự chưa. Nếu chưa có thì mới sinh ra một thể nghiệm mới.

Để điều này có thể thực hiện được tất cả các tham số kiểu chung đều phải xuất hiện trong đặc tả kiểu ít nhất 1 lần của các tham số thường của hàm. Ví dụ trong hàm min T xuất hiện cả trong đặc tả kiểu cho tham số a và tham số b. Khi các thể nghiệm của các bản mẫu hàm được tạo ra sẽ không có nhiều chuyển kiểu tự động như so với các hàm bình thường mà sự khớp giữa các tham số thực sự với các tham số hình thức còn đòi hỏi ngặt nghèo hơn ví dụ việc gọi hàm min như sau:

cout << min(m,y) sẽ lập tức sinh lỗi.

Trên thực tế trong một lời gọi hàm có thể chỉ định một cách tường minh kiểu mà các tham số kiểu chung có thể nhận và cú pháp thực hiện điều này cũng giống như với các thể nghiệm của các lớp bản mẫu:

cout << min<double> (m, y);

Việc chuyển đổi kiểu sẽ được thực hiện vì min<double> là một hàm cụ thể, có nghĩa là không có thao tác kiểm tra khớp kiểu ở đây, đoạn mã tương ứng với min<double> sẽ được sinh ra.

Hàm thứ hai mà chúng ta xem xét là một hàm hoán vị giá trị cho hai biến kiểu bất kỳ:

template <class T>

void swap(T &a, T &b){ T temp = a; a = b; b = temp; }

Điều cần chú ý với hàm swap này là các kiểu sử dụng với chúng cần có hàm toán tử gán.

Các bản mẫu hàm đặc biệt hay được sử dụng với các thao tác hay dùng với các mảng ví dụ:

```
template <class T>
T & min_element(T f[], int n){
  int m = 0;
  for(int i=1;i<n;i++)
        if(f[i] < f[m]    m = i;
  return f[m];
}
```

Hàm sắp xếp cũng là một hàm hay dùng vì thế sẽ rất tiện lợi nếu chúng ta có một hàm có khả năng sắp xếp các mảng có kiểu bất kỳ:

```
template <class T>
void sort(T f[], int n){
  for(int i=0;i<n;i++)
        swap(f[i],min_element(&f[i], n-k);
}
```

Khi một thể nghiệm của hàm sort được tạo ra với một kiểu dữ liệu cụ thể T, các thể nghiệm của các hàm swap và min_element cũng tự động được sinh ra với cùng kiểu.

Tổng kết: các bản mẫu hàm

```
template <class T1, class T2>
return_type f(parameters)
{...}
```

trong đó T1 và T2, ... là các tham số và được sử dụng như các kiểu bình thường khác trong thân hàm. Tất cả các tham số kiểu phải xuất hiện trong đặc tả trong danh sách tham số.

Các thể nghiệm của bản mẫu hàm được gọi tới một cách tự động khi có câu lệnh:

f(các tham số thực sự);

trình biên dịch sẽ khớp các tham số thực sự với các tham số hình thức và xác định kiểu nào sẽ thay thế cho T.

Chúng ta có thể chỉ định tường minh kiểu của T:

f<type>(các tham số thực sự);

Một thể nghiệm đặc biệt đối với một kiểu cụ thể V có thể được tạo ra. Thân hàm sẽ được viết lại và T sẽ được thay thế bằng V.

Cũng giống như các bản mẫu lớp chúng ta có thể tạo ra một thể nghiệm đặc biệt của bản mẫu hàm với một kiểu đặc biệt nào đó. Hàm min_element sẽ không làm việc chẳng hạn trong trường hợp chúng ta có một mảng các xâu, khi đó chúng ta cần có một phiên bản đặc biệt của hàm này để làm việc với các mảng mà các phần tử mảng có kiêu char *:

```
char * & min_element(char * f[], int n){
```

```
  int m = 0;
  for(int i=1;i<m;i++)
        if(strcmp(f[i],f[m])<0) m = i;
  return f[m];
}
```

Mọi vị trí của T bây giờ được thay bằng char *. Khi gặp một lời gọi tới thể nghiệm hàm có kiểu là char * thay vì sinh ra một thể nghiệm từ bản mẫu trình biên dịch sẽ trực tiếp biên dịch đoạn mã được viết riêng để làm việc với kiểu này.

Tất nhiên cũng giống như các bản mẫu lớp chúng ta có thể sử dụng nhiều tham số bản mẫu với bản mẫu hàm, đồng thời có thể dùng của các tham số bản mẫu kiểu và tham số bản mẫu giá trị.

Các hàm chuẩn chung – thư viện thuật toán

Cũng như các lớp bản mẫu có rất nhiều hàm bản mẫu trong đặc tả chuẩn của ngôn ngữ C++. Chúng có khả năng thực hiện một số các thao tác khác nhau trên các lớp chứa và các data collection khác. Chẳng hạn chúng ta có thể thực hiện tìm kiếm, sắp xếp và thực hiện các thay đổi dữ liệu khác. Các bản mẫu hàm này được định nghĩa trong file header algorithm.

Tất cả các hàm bản mẫu này đều sử dụng các iterator để quản lý dữ liệu.

Thư viện STL là thư viện bản mẫu chuẩn đang được sử dụng rộng rãi nhất hiện nay.

6.3. Lớp bản mẫu

6.3.1. Các bản mẫu và thể nghiệm

Chúng ta xem xét lớp Matrix thể hiện các ma trận không sử dụng bản mẫu như sau:

```
class Matrix{
  public:
        Matrix(int i=0, int j=0):r(i), c(j), a(new double[r*c]){}
        Matrix(const Matrix &m):a(0){*this = m;}
        ~Matrix(){delete []a;}
        int num_row() {return r;} // cho biết số hàng của ma trận
        int num_col() {return c;}  // cho biết số cột của ma trận
        Matrix & operator = (const Matrix &); // toán tử gán
        double & operator()(int i, int j); // toán tử chỉ số
  private:
        int r, c;
        double *a;
};
```

Như chúng ta đã biết ma trận là một bảng các hàng và các cột. Mỗi phần tử riêng biệt của ma trận trong lớp Matrix trên có kiểu là double. Để biểu diễn các phần tử của 1 ma trận có r hàng và c cột chúng ta sử dụng một mảng một chiều có (r * c) phần tử và một con trỏ a để

quản lý mảng một chiều này. Các phần tử của ma trận được lưu trong mảng một chiều này theo nguyên tắc hàng 1 trước, sau đó đến hàng 2 ... Lớp Matrix có hai cấu tử, 1 cấu tử với hai tham số khởi tạo hàng và cột, phần dữ liệu không có gì, một cấu tử copy bình thường. Còn lại các phương thức khác có lẽ không cần giải thích nhiều.

```
Matrix & Matrix::operator = (const Matrix & m){
  if(this != &m){
        r = m.r;
        c = m.c;
        delete [] a;
        a = new double[r*c];
        for(int i=0; i< r*c; i++)        a[i] = m.a[i];
  }
  return *this;
}
double & Matrix::operator () (int i, int j){
  if(i<1 || i>r || j<1 || j>c)        return 0;
  return a[(i-1)*c + j-1];
}
```

Lớp Matrix này tất nhiên có thể hoạt động tốt nhưng chỉ vơi các ma trận mà các phần tử có kiểu double. Vì thế chúng ta có thể viết lại lớp Matrix để nó trở thành một bản mẫu:

```
template <class T>
class Matrix{
  public:
        Matrix(int i=0, int j=0):r(i), c(j), a(new T[r*c]){}
        Matrix(const Matrix &m):a(0){*this = m;}
        ~Matrix(){delete []a;}
        int num_row() {return r;} // cho biết số hàng của ma trận
        int num_col() {return c;}  // cho biết số cột của ma trận
        Matrix & operator = (const Matrix &); // toán tử gán
        T & operator()(int i, int j); // toán tử chỉ số
  private:
        int r,c;
        T *a;
};
```

Có hai thay đổi trong cài đặt của chúng ta so với lớp Matrix trước. Đầu tiên là dòng trước khai báo lớp Matrix. Từ khóa template báo cho trình biên dịch biết rằng lớp Matrix là một bản mẫu và không phải là một lớp bình thường. Các tham số bản mẫu (template

parameters) hay các tham số chung được đặt giữa hai dấu < và >. Trong trường hợp này chỉ có một tham số bản mẫu và nó được đặt tên là T. Từ khóa class cũng chỉ ra rằng T là một kiểu dữ liệu bất kỳ nào đó. (chú ý là T cũng có thể là một kiểu dữ liệu built-in chẳng hạn như int hoặc char * mặc dù từ khóa class vẫn được sử dụng).

Thay đổi thứ hai là chúng ta thay mọi chỗ nào có từ khóa double bằng T. Ví dụ biến thành viên a sẽ có kiểu là T * thay vì double *.

Việc sử dụng lớp Matrix cũng hơi khác chút ít:

Matrix<int> mi(2,3); // khai báo một ma trận các số kiểu int có kích thước là 2x3

Matrix<Person> mp(10,3); // khai báo một ma trận các số kiểu int có kích thước là 10x3

Trình biên dịch đã sử dụng bản mẫu Matrix để sinh ra hai lớp thường chứa các kiểu dữ liêu khác nhau. Chúng ta gọi đó là hai thể nghiệm của lớp Matrix (có thể dùng thuật ngữ các lớp sinh hoặc là các đặt tả). Trong trường hợp thứ nhất dường như kiểu tham số T sẽ được thay bằng kiểu int ở bất kỳ vị trí nào nó xuất hiện, và trong trường hợp thứ hai là kiểu Person. Cách dễ nhất để giải thích là trình biên dịch sẽ tự động sinh ra các đoạn mã tương ứng thay thế kiểu tham số lớp và biên dịch như một lớp thường (thực tế thì công việc này phức tạp hơn nhiều nhưng có lẽ là những gì mà trình biên dịch tự mình thực hiện). Mỗi một khai báo Matrix<X>, trong đó X là một kiểu nào đó sẽ tương ứng với lớp mới được trình biên dịch sinh ra. Tên lớp này được sử dụng bình thường như các tên lớp khác và các tham số hàm cũng như các biến của lớp có thể được khai báo một cách bình thường.

Trong lớp Matrix chúng ta có hai hàm thành viên operator = và operator () được định nghĩa riêng rẽ. Do lớp Matrix là một bản mẫu nên các hàm thành viên của nó cũng là các bản mẫu.

```
template <class T>
Matrix<T> & Matrix<T>::operator = (const Matrix & m){
  if(this != &m){
        r = m.r;
        c = m.c;
        delete [] a;
        a = new T[r*c];
        for(int i=0; i< r*c; i++)        a[i] = m.a[i];
  }
  return *this;
}
template <class T>
T & Matrix<T>::operator () (int i, int j){
  if(i<1 || i>r || j<1 || j>c)        return 0; //return a[0];
  return a[(i-1)*c + j-1];
}
```

Chúng ta thấy rằng cả hai khai báo này đều bắt đầu bằng dòng <template class T> giống như khai báo lớp Matrix ban đầu. Vì tất cả các thể nghiệm của lớp bản mẫu đều có các thể nghiệm riêng của các hàm thành viên của nó nên chúng ta buộc phải viết là Matrix<T>:: trước tên hàm khi chúng ta muốn chỉ ra tên đầy đủ của một hàm thành viên. Nếu chúng ta muốn chỉ rõ tên của bản mẫu lớp trong định nghĩa của nó (lớp) chúng ta không cần viết cả <T> và tên lớp mà chỉ cần tên lớp là đủ. Có thể thấy rõ chú ý này với tham số của hàm toán tử = chúng ta chỉ cần viết là Matrix & chứ không cần Matrix<T>, tuy nhiên với phiên bản 3.0 chúng ta cần chỉ rõ tham số là Matrix<T> thì mới biên dịch được.

Kết luận:

```
template <class T>
class C{
  return_type f(parameters);
  ....
};
template <class T>
return_type C<T>::f(parameters){
  ...
}
```

T là một tham số kiểu và được dụng như một kiểu bình thường trong lớp.

Cả định nghĩa lớp và định nghĩa hàm của một lớp bản mẫu nên đặt trong một file header của lớp đó.

Các thể nghiệm của một lớp bản mẫu được tạo ra khi khai báo:

C<type_name> khi đó T sẽ được thay bằng type_name.

Biểu thức C<type_name> được dùng như một tên lớp bình thường.

Với một lớp bất kỳ chúng ta nên có hai file tương ứng: một file khai báo (.h) (header) và một file định nghĩa (.cpp). Chúng ta cũng được học rằng định nghĩa lớp nên đặt trong file header và định nghĩa hàm nên đặt trong file .cpp. Với các lớp bản mẫu chúng ta có đôi chút thay đổi trong khuyến cáo này. Vì trình biên dịch phải tự động sinh ra một lớp nào đó từ lớp bản mẫu nên nó phải truy cập tới toàn bộ định nghĩa và cài đặt của lớp bản mẫu. Chính vì thế chúng ta đặt tất cả cài đặt của một lớp trong file header. Điều này có nghĩa là không có file định nghĩa .cpp. Thực chất các trình biên dịch khác nhau giải quyết vấn đề này theo các cách khác nhau nhưng khuyến cáo này có thể làm việc với mọi trình biên dịch.

Giả sử chúng ta cần có một hàm có khả năng cộng hai ma trận trong trường hợp chúng chứa các số nguyên chẳng hạn. Để giải quyết vấn đề này với lớp bản mẫu Matrix chúng ta có thể cài đặt một hàm add như sau:

```
Matrix<int> add(Matrix<int> a, Matrix<int> b){
  if((a.num_row()!=b.num_row()) || (a.num_col()!=b.num_col()))
        return Matrix<int>(0);
  Matrix<int> c(a);
```

```
  for(int i=1;i<=c.num_row();i++)
        for(int j=1;j<=c.num_col();j++)
              c(i,j) += b(i,j);
  return c;
}
```

Và đây là một đoạn chương trình hoàn chỉnh đọc vào một ma trận kiểu int và nhân đôi nó.

```
int main(){
  int x, y;
  cout << "Number of rows:"; cin >> x;
  cout << "Number of columns:"; cin >> y;
  Matrix<int> mi(x,y);
  for(int i=1;i<=mi.num_row();i++)
        for(int j=1;j<=mi.num_col();j++) cin >> mi(i,j);
  mi = add(mi,mi);
  for(i=1;i<=mi.num_row();i++){
        for(j=1;j<=mi.num_col();j++)        cout << setw(4) << mi(i,j) ;
        cout << endl;
  }
  return 0;
}
```

6.3.2. Các thành phần tĩnh

Cũng như các lớp thường khác một lớp bản mẫu cũng có các thành viên dữ liệu và hàm thành viên tĩnh. Chẳng hạn chúng ta có thể thêm vào lớp bản mẫu Matrix một biến thành viên tĩnh mn để kiểm soát số các ma trận có các phần tử thuộc một kiểu nào đó và một hàm thành viên tĩnh để đọc giá trị này. (Bình thường các hàm cấu tử và hủy tử sẽ thay đổi để chúng có thể đếm và thay đổi giá trị của biến thành viên tĩnh này nhưng chúng ta sẽ không xem xét cụ thể ở đây, tất cả các cấu tử được sử dụng cũng như các hủy tử).

```
template <class T>
class Matrix{
  public:
        ...
        static int num_matrices();
  private:
        ...
        static int mn;
};
```

Đối với các lớp thường chúng ta biết rằng các thành viên tĩnh là chung đối với các đối tượng của một lớp, nó chỉ có một bản (edition) duy nhất. Đối với các lớp bản mẫu, mỗi lớp thể nghiệm của lớp bản mẫu sẽ có một bản riêng của biến thành viên tĩnh. Chẳng hạn nếu chúng ta có 3 lớp thể nghiệm của lớp Matrix là Matrix<int>, Matrix<double>, Matrix<float> chúng ta sẽ có 3 biến tĩnh, mỗi biến cho một lớp thể nghiệm.

Các hàm thành viên tĩnh được định nghĩa tương tự như với các hàm không tĩnh ví dụ:

```
template <class T>
int Matrix<T>::num_matrices(){
  return mn;
}
```

Các biến thành viên tĩnh phải được định nghĩa bên ngoài lớp, điều này cũng đúng với các biến thành viên tĩnh của một lớp bản mẫu. Vì sẽ có một thể nghiệm của biến thành viên tĩnh cho tất cả các thể nghiệm của lớp bản mẫu nên định nghĩa của biến thành viên tĩnh cũng phải là một bản mẫu:

```
template <class T>
int Matrix<T>::mn = 0;
```

Chú ý về các thành viên tĩnh của các lớp bản mẫu:

Tồn tại dưới dạng một bản đối với mỗi thể nghiệm của lớp

Các hàm thành viên tĩnh được định nghĩa như các hàm bình thường của lớp bản mẫu.

Các biến thành viên tĩnh là các bản mẫu và phải được định nghĩa riêng biệt, ví dụ đối với một thành viên tĩnh v của lớp bản mẫu C sẽ được định nghĩa như sau:

```
template <class T>
type C<T>::v = giá trị khởi tạo;
```

Nếu có một hàm thành viên tĩnh f của một lớp thường C, thì f có thể được gọi theo 2 cách: x.f() hoặc C::f() trong đó x là một đối tượng thuộc C. Ví dụ với lớp Matrix ta có thể gọi như sau:

```
cout << "Num of int matrices:" << mi.num_matrices();
```

hoặc:

```
cout << "Num of int matrices:" << Matrix<int>::num_matrices();
```

6.3.3. Các lớp bạn và lớp trợ giúp

Nếu chúng ta đã có một lớp non-generic, nó có thể được chuyển thành một lớp bản mẫu bằng một cách tương đối dễ dàng.

Ví dụ: lớp Stack:

```
template <class T>
class Stack{
  public:
        Stack():first(0){};
```

```
        ~Stack();
        void push(T d);
        T pop();
        Bool isEmpty();
        Bool isFull();
    private:
        Element<T> *first;
        Stack(const Stack &){};
        Stack & operator=(const Stack &){};
};
```

Để cài đặt lớp Stack chúng ta sử dụng một lớp trợ giúp là lớp Element, lớp định nghĩa các phần tử riêng biệt trong một danh sách liên kết. Lớp Element cũng là một lớp bản mẫu vì các phần tử của danh sách sẽ chứa các phần tử dữ liệu có kiểu khác nhau đối với mỗi lớp thể nghiệm của lớp Stack. Như vậy trong khai báo của lớp Element chúng ta sẽ sử dụng một tham số bản mẫu khác chẳng hạn U, nhưng do trong khai báo lớp Stack biến first có kiểu là Element<T> nên Stack thể nghiệm tương ứng sẽ vẫn chứa dữ liệu có kiểu T, có nghĩa là kiểu dữ liệu của Stack là do nó quyết định mặc dù Element là một lớp bản mẫu trợ giúp.

```
template <class U>
class Element{
  friend class Stack<U>;
  Element *next;
  U data;
  Element(Element *n, U d):next(n),data(d){}
};
```

Chú ý rằng lớp Stack phải điều khiển được các thành phần dữ liệu trong lớp Element nên nó phải là một lớp bạn của Element. Thứ hai là tương thích về kiểu nên trong khai báo lớp Element Stack sẽ là Stack<U>, nếu không trình biên dịch sẽ không hiểu.

Bây giờ chúng ta sẽ thực hiện cài đặt các phương thức của lớp Stack:

```
template <class T>
void Stack<T>::push(T d){       first = new Element<T>(first,d); }
template <class T>
T Stack<T>::pop(){
Element<T> * p = first;
T d = p->data;
first = first->next;
delete p;
return d;
```

```
}
template <class T>
Stack<T>::~Stack(){
while(!isEmpty()){
Element<T> * p = first;
first = first->next;
delete p;
}
}
template <class T>
Stack<T>::~Stack(){
  while(!isEmpty()){
            Element<T> * p = first;
            first = first->next;
            delete p;
  }
}
```

Và đây là đoạn chương trình chính sử dụng lớp Stack trên:

```
int main(){
  int n;
  cout << "Ngai muon xem dang nhi phan cua so:";     cin >> n;
  Stack<int> stkint;
  while(n){        stkint.push(n%2);       n/=2;   }
  cout <<"Ket qua:";
  while(!stkint.isEmpty())         cout << stkint.pop();
  /* cần include “mstring.h”
Stack<String> stks;
  stks.push("ajg ajgd");
  stks.push(" a 128");
  while(!stks.isEmpty())           cout << stks.pop() << endl;
  */
  return 0;
}
```

6.3.4. Các tham số bản mẫu

Các lớp bản mẫu mà chúng ta xem xét từ đầu tới giờ chỉ có một tham số bản mẫu nhưng trên thực tế một lớp bản mẫu hoặc một hàm bản mẫu có thể có số tham số bản mẫu bất kỳ.

Chúng ta chia các tham số bản mẫu ra làm hai phần: các kiểu tham số bản mẫu và các giá trị tham số bản mẫu. Các kiểu tham số bản mẫu được chỉ ra bởi từ khóa class đứng trước và có thể là bất kỳ kiểu nào còn các giá trị tham số giống như các tham số hàm bình thường nhưng có thể không phải là kiểu dấu phẩy động.

Chúng ta sẽ xem xét ví dụ sau: cài đặt một stack bằng mảng, có hai tham số bản mẫu được dùng, ngoại trừ T còn có một tham số int để chỉ định số phần tử lớn nhất mà stack có thể chứa:

```
template <class T, int size>
class Stack{
  public:
        Stack():n(0){};
        ~Stack();
        void push(const T & d){s[n++] = d};
        T pop(){return s[n--];};
        Bool isEmpty(){return (n<=0)?True:False;};
        Bool isFull();
  private:
        T s[size];
};
```

Khi chúng ta tạo ra một thể nghiệm của lớp bản mẫu này chúng ta cần cung cấp các giá trị cho cả hai tham số bản mẫu:

Stack<int, 200) x; // ngăn xếp có tối đa 200 phần tử

Khi giá trị được gán cho tham số giá trị cần phải có một biểu thức hằng chỉ ra rằng có 1 giá trị có thể tính tại thời điểm biên dịch chương trình.

Chú ý: Các tham số bản mẫu

template <param 1, param 2, param 3, ...>

Trong đó param N có thể có dạng:

class T hoặc

type_name V

Một tham số giá trị có thể không phải là kiểu số dấu phẩy động

Có thể sử dụng các tham số mặc định.

Giống như các tham số hàm bình thường các tham số bản mẫu cũng có thể nhận các giá trị mặc định ví dụ chúng ta có thể chuyển khai báo Stack trên thành:

```
template <class T=double, int size = 100>
class Stack{
....
};
```

Nếu một tham số bản mẫu có giá trị mặc định thì các tham số bản mẫu sau đó cũng bắt buộc phải có giá trị mặc định. Nếu lớp bản mẫu có tham số bản mẫu mặc định ta có thể bỏ qua các tham số đó khi tạo ra các thể nghiệm lớp chẳng hạn:

Stack<Person> sp;// một ngăn xếp Person có kích thước max là 100

Stack<> sf; //ngăn xếp double có max phần tử là 100.

6.3.5. Các lớp thuộc tính

Khi chúng ta tạo ra một thể nghiệm của một lớp bản mẫu nào đó có một tham số kiểu T, kiểu T sẽ được thay thếở tất cả các vị trí trong lớp.Việc thay thế này có thể làm việc tốt với tất cả các kiểu đối với một lớp bản mẫu nào đó nhưng không phải tất cả các lớp bản mẫu đều như vậy. Để minh họa chúng ta sẽ lấy ví dụ về lớp bản mẫu Tree:

```
#include <iostream>
#include <bool.h>
template <class T>
class Node{
  friend class Tree<T>;
  T data;
  Tree<T> * left, * right;
  Node(T d):data(d), left(new Tree<T>), right(new Tree<T>){};
  ~Node(){delete left; delete right;};
};
template <class D>
class Tree{
  public:
        Tree():root(0){};
        Tree(D d){ root = new Node<D>(d);};
        Tree(const Tree & t){copy(t);};
        ~Tree(){delete root;};
        Bool empty()const{return (root==0)?True:False;};
        D & value() const{return root->data;};
        Tree & l_child() const {return *root->left;};
        Tree & r_child() const {return *root->right;};
        Tree & operator =(const Tree & t);
        Bool operator==(const Tree & t) const;
        void inorder() const;
        void put_in(D d);
        D* search(D d);
  private:
```

```
        Node<D> *root;
        void copy(const Tree & t);
};
template <class D>
void Tree<D>::copy(const Tree<D> & t){
  if(t.empty())    root = 0;
  else{
        root = new Node<D>(t.value());
        l_child().copy(t.l_child());
        r_child().copy(t.r_child());
  }
}
template <class D>
Tree<D> & Tree<D>::operator=(const Tree<D> & t){
  if(root!=t.root){
        delete root;
        copy(t);
  }
  return *this;
}
template <class D>
Bool Tree<D>::operator==(const Tree<D> & t)const{
  if((empty()&&t.empty())||
(!empty()&&!t.empty()&&l_child()==t.l_child()&&r_child()==t.r_child()))
        return True;
  else
        return False;
}
template <class D>
void Tree<D>::inorder() const{
  if(!empty()){
        l_child().inorder();
        cout << value() << " ";
        r_child().inorder();
  }
}
```

```
int main(){
  Tree<int> t;
  t = Tree<int>(10);
  t.l_child() = Tree<int>(20);
  t.l_child().r_child() = Tree<int>(12);
  t.r_child() = Tree<int>(30);
  t.inorder();
  return 0;
}
```

Tuy nhiên nếu chúng ta có thêm một lớp Person chẳng hạn được khai báo như sau:

```
class Person{
  private:        char pno[10];  char name[30];
  public:         char * getPno() const; char * getName() const;
};
```

Và trong chương trình chính chúng ta khai báo:

```
Tree<Person> tp;
```

Thì lập tức trình biên dịch sẽ báo lỗi, lý do là do trong các hàm của lớp bản mẫu Tree chúng ta đã dùng một số hàm mặc định có với kiểu int nhưng chưa có với lớp Person:

```
Person(char * PNO, char *n){strcpy(pno,PNO);strcpy(name,n);};
Bool operator>(const Person &);
Bool operator<(const Person &);
Bool operator==(const Person &);
friend ostream & operator <<(ostream & out, const Person &);
```

Sau khi cài đặt các hàm này mọi việc lại trở lại như xưa.

Ở đây điều cần chú ý là: bình thường với các kiểu dữ liệu built-in một số hàm (chẳng hạn các hàm toán tử, cấu tử copy) mặc định có nên việc chương trình dùng chúng là không sao, nhưng với các kiểu dữ liệu do người dùng định nghĩa không có các hàm này nên cần phải cài đặt chúng nếu chương trình có dùng đến.

Bài tập

1. Xây dựng bản mẫu cho lớp Hàng hóa
2. Xây dựng bản mẫu cho lớp Matrix
3. Xây dựng bản mẫu cho lớp List
4. Xây dựng hàm bản mẫu tìm phần tử nhỏ nhất

TÀI LIỆU THAM KHẢO

1. Phạm Văn Ất, Kỹ thuật lập trình hướng đối tượng, NXB KHKT, 2004
2. Nguyễn Thanh Thủy, Bài tập Lập trình HĐT với C++, NXB KHKT, 2004.

3. Đặng Xuân Hường, Lập trình hướng đối tượng với C++, NXB Thống kê, 2004