CHƯƠNG 2

NHỮNG MỞ RỘNG CỦA C++

Mục đích chương này:

1. Giới thiệu những điểm khác biệt chủ yếu giữa C và C++
2. Các điểm mới của C++ so với C (những vấn đề cơ bản nhất)

1. CÁC ĐIỂM KHÔNG TƯƠNG THÍCH GIỮA C++ VÀ ANSI C

1.1 Định nghĩa hàm

Trong định nghĩa hàm, ANSI C cho phép hai kiểu khai báo dòng tiêu đề của hàm, trong khi đó C++ chỉ chấp nhận một cách:

```
/*C++ không có khai báo kiểu này*/
double fexple(u,v)
int u;
double v;
```

```
/*cả C và C++ cho phép*/
double fexple(int u,double v)
int u;
double v;
```

1.2 Khai báo hàm nguyên mẫu

Trong ANSI C, khi sử dụng một hàm chưa được định nghĩa trước đó trong cùng một tệp, ta có thể:

3. không cần khai báo (khi đó ngầm định giá trị trả về của hàm là int)
4. chỉ cần khai báo tên hàm và giá trị trả về, không cần danh sách kiểu của các tham số.
5. khai báo hàm nguyên mẫu.

Với C++, chỉ có phương pháp thứ 3 là chấp nhận được. Nói cách khác, một lời gọi hàm chỉ được chấp nhận khi trình biên dịch biết được kiểu của các tham số, kiểu của giá trị trả về. Mỗi khi trình biên dịch gặp một lời gọi hàm, nó sẽ so sánh các kiểu của các đối số được truyền với các tham số hình thức tương ứng. Trong trường hợp có sự khác nhau, có thể thực hiện một số chuyển kiểu tự động để cho hàm nhận được có danh sách các tham số đúng với kiểu đã được khai báo của hàm. Tuy nhiên phải tuân theo nguyên tắc chuyển kiểu tự động sau đây:

char-->int-->longint-->float-->double

Ví dụ 2.1

```
double fexple (int, double)   /*khai báo hàm fexple*/
    ....
```

```
main(){
int n;
char c;
double z,res1,res2,res3;
....
res1 = fexple(n,z); /* không có chuyển đổi kiểu*/
res2 = fexple(c,z); /* có chuyển đổi kiểu, từ char (c) thành int*/
res3 = fexple(z,n); /* có chuyển đổi kiểu, từ double(z) thành int và từ int(n)
                       thành double*/
....
}
```

Trong C++ bắt buộc phải có từ khoá **void** trước tên của hàm trong phần khai báo để chỉ rằng hàm không trả về giá trị. Trường hợp không có, trình biên dịch ngầm hiểu kiểu của giá trị trả về là **int** và như thế trong thân hàm bắt buộc phải có câu lệnh **return**. Điều này hoàn toàn không cần thiết đối với mô tả trong ngôn ngữ C.

Thực ra, các khả năng vừa mô tả không hoàn toàn là điểm không tương thích giữa C và C++ mà đó chỉ là sự "gạn lọc" các điểm yếu và hoàn thiện các mặt còn chưa hoàn chỉnh của C.

1.3 Sự tương thích giữa con trỏ *void* và các con trỏ khác

Trong ANSI C, kiểu **void *** tương thích với các kiểu trỏ khác cả hai chiều. Chẳng hạn với các khai báo sau :

```
void *gen;
int  *adj;
```

hai phép gán sau đây là hợp lệ trong ANSI C:

```
gen = adj;
adj = gen;
```

Thực ra, hai câu lệnh trên đã kèm theo các phép "chuyển kiểu ngầm định":

int* --->**void*** đối với câu lệnh thứ nhất, và

void* --->**int*** đối với câu lệnh thứ hai.

Trong C++, chỉ có chuyển đổi kiểu ngầm định từ một kiểu trỏ tuỳ ý thành **void*** là chấp nhận được, còn muốn chuyển đổi ngược lại, ta phải thực hiện chuyển kiểu tường minh như cách viết sau đây:

```
gen = adj;
adj = (int *)gen;
```

2. CÁC KHẢ NĂNG VÀO/RA MỚI CỦA C++

Các tiện ích vào/ra (hàm hoặc macro) của thư viện C chuẩn đều có thể sử dụng trong C++. Để sử dụng các hàm này chúng ta chỉ cần khai báo tệp tiêu đề trong đó có chứa khai báo hàm nguyên mẫu của các tiện ích này.

Bên cạnh đó, C++ còn cài đặt thêm các khả năng vào/ra mới dựa trên hai toán tử "<<"(xuất) và ">>" (nhập) với các đặc tính sau đây:

6. đơn giản trong sử dụng
7. có khả năng mở rộng đối với các kiểu mới theo nhu cầu của người lập trình.

Trong tệp tiêu đề `iostream.h` người ta định nghĩa hai đối tượng **cout** và **cin** tương ứng với hai thiết bị chuẩn ra/vào được sử dụng cùng với "<<" và ">>". Thông thường ta hiểu **cout** là màn hình còn **cin** là bàn phím.

2.1 Ghi dữ liệu lên thiết bị ra chuẩn (màn hình) *cout*

Trong phần này ta xem xét một số ví dụ minh hoạ cách sử dụng **cout** và "<<" để đưa thông tin ra màn hình.

Ví dụ 2.2

Chương trình sau minh hoạ cách sử dụng **cout** để đưa ra màn hình một xâu ký tự.

```
#include <iostream.h> /*phải khai báo khi muốn sử dụng cout*/
main()
   {
    cout << "Welcome C++";
   }
```

```
Welcome C++
```

"<<" là một toán tử hai ngôi, toán hạng ở bên trái mô tả nơi kết xuất thông tin (có thể là một thiết bị ngoại vi chuẩn hay là một tập tin), toán hạng bên phải của "<<" là một biểu thức nào đó. Trong chương trình trên, câu lệnh `cout <<"Welcome C++"` đưa ra màn hình xâu ký tự "`Welcome C++`".

Ví dụ 2.3

Sử dụng **cout** và "<<" đưa ra các giá trị khác nhau:

```
#include <iostream.h> /*phải khai báo khi muốn sử dụng cout*/
void main() {
    int n = 25;
    cout << "Value : ";
    cout << n;
   }
```

```
Value : 25
```

Trong ví dụ này chúng ta đã sử dụng toán tử "<<" để in ra màn hình đầu tiên là một xâu ký tự, sau đó là một số nguyên. Chức năng của toán tử "<<" rõ ràng là khác nhau trong hai lần kết xuất dữ liệu: với câu lệnh thứ nhất, chỉ đưa ra màn hình một dãy các ký tự, ở câu lệnh sau, đã sử dụng một khuôn mẫu để chuyển đổi một giá trị nhị phân thành một chuỗi các ký tự chữ số. Việc một toán tử có nhiều vai trò khác nhau liên quan đến một khái niệm mới trong C++, đó là "*Định nghĩa chồng toán tử*". Điều này sẽ được đề cập đến trong chương 4.

Ví dụ 2.4

Trong ví dụ này ta gộp cả hai câu lệnh kết xuất trong ví dụ 2.3 thành một câu lệnh phức tạp hơn, tuy kết quả không khác trước:

```
#include <iostream.h> /*phải khai báo khi muốn sử dụng cout*/
void main() {
   int n = 25;
    cout << "Value : " << n;
   }
```

```
Value : 25
```

2.2 Các khả năng viết ra trên *cout*

Chúng ta vừa xem xét một vài ví dụ viết một xâu ký tự, một số nguyên. Một cách tổng quát, chúng ta có thể sử dụng toán tử "<<" cùng với **cout** để đưa ra màn hình giá trị của một biểu thức có các kiểu sau:

8. kiểu dữ liệu cơ sở (**char**, **int**, **float**, **double**),
9. xâu ký tự: (**char** *),
10. con trỏ (trừ con trỏ **char** *)

Trong trường hợp muốn đưa ra địa chỉ biến xâu ký tự phải thực hiện phép chuyển kiểu tường minh, chẳng hạn (**char** *) --> (**void** *).

Xét ví dụ sau đây:

Ví dụ 2.4

```
#include <iostream.h>
void void main(){
int n = 25;
long p = 250000;
unsigned q = 63000;
char c = 'a';
float x = 12.3456789;
double y = 12.3456789e16;
char * ch = "Welcome C++";
int *ad = &n;
cout <<"Value of n : " << n <<"\n";
cout <<"Value of p : " << p <<"\n";
cout <<"Value of c : " << c <<"\n";
cout <<"Value of q : " << q <<"\n";
cout <<"Value of x : " << x <<"\n";
cout <<"Value of y : " << y <<"\n";
cout <<"Value of ch : " << ch <<"\n";
cout <<"Addrese of n : " << ad <<"\n";
cout <<"Addrese de ch : " << (void *)ch <<"\n";
}
```

```
Value of n : 25
Value of p : 250000
Value of c : a
Value of q : 63000
Value of x : 12.345679
Value of y : 1.234567e+17
Value of ch : Welcome C++
Addrese of n : 0xfff2
Addrese de ch : 0x00b2
```

2.3 Đọc dữ liệu từ thiết bị vào chuẩn (bàn phím) *cin*

Nếu như **cout** dùng để chỉ thiết bị ra chuẩn, thì **cin** được dùng để chỉ một thiết bị vào chuẩn. Một cách tương tự, toán tử ">>" được dùng kèm với **cin** để nhập vào các giá trị; hai câu lệnh

```
int n;
cin >> n;
```

yêu cầu đọc các ký tự trên bàn phím và chuyển chúng thành một số nguyên và gán cho biến n.

Giống như **cout** và "<<", có thể nhập nhiều giá trị cùng kiểu hay khác kiểu bằng cách viết liên tiếp tên các biến cần nhập giá trị cùng với ">>" ngay sau **cin**. Chẳng hạn:

```
int n;
float p;
char c;
cin >> c >> n >> p;
```

Có thể sử dụng toán tử ">>" để nhập dữ liệu cho các biến có kiểu **char**, **int**, **float**, **double** và **char** *.

Giống với hàm `scanf()`, **cin** tuân theo một số qui ước dùng trong việc phân tích các ký tự:

(i) Các giá trị số được phân cách bởi: SPACE, TAB, CR, LF. Khi gặp một ký tự "không hợp lệ" (dấu "." đối với số nguyên, chữ cái đối với số, ...) sẽ kết thúc việc đọc từ **cin**; ký tự không hợp lệ này sẽ được xem xét trong lần đọc sau.

(ii) Đối với giá trị xâu ký tự, dấu phân cách cũng là SPACE, TAB,CR, còn đối với giá trị ký tự, dấu phân cách là ký tự CR. Trong hai trường hợp này không có khái niệm "ký tự không hợp lệ". Mã sinh ra do bấm phím Enter của lần nhập trước vẫn được xét trong lần nhập xâu/ký tự tiếp theo và do vậy sẽ có "nguy cơ " không nhập được đúng giá trị mong muốn khi đưa ra lệnh nhập xâu ký tự hoặc ký tự ngay sau các lệnh nhập các giá trị khác. Giải pháp khắc phục vấn đề này để đảm bảo công việc diễn ra đúng theo ý là trước mỗi lần gọi lệnh nhập dữ liệu cho xâu/ký tự ta sử dụng một trong hai chỉ thị sau đây:

```
fflush(stdin); //khai báo trong stdio.h
cin.clear(); //hàm thành phần của lớp định nghĩa đối tượng cin
```

Ta tham khảo chương trình sau:

Ví dụ 2.6

```
#include <iostream.h>
#include <conio.h>
void main() {
 int n;
 float x;
 char t[81];
 clrscr();
 do {
 cout << "Nhap vao mot so nguyen, mot xau, mot so thuc : ";
 cin >> n >> t >> x;
 cout << "Da nhap " << n << "," << t << " va " << x << "\n";
 } while (n);
 }
```

```
Nhap vao mot so nguyen, mot xau, mot so thuc :  3 long 3.4
Da nhap 3,long va 3.4
Nhap vao mot so nguyen, mot xau, mot so thuc :  5 hung 5.6
Da nhap 5,hung and 5.6
Nhap vao mot so nguyen, mot xau, mot so thuc :  0 4
3
Da nhap 0,4 va 3
```

3. NHỮNG TIỆN ÍCH CHO NGƯỜI LẬP TRÌNH

3.1 Chú thích cuối dòng

Mọi ký hiệu đi sau "//" cho đến hết dòng được coi là chú thích, được chương trình dịch bỏ qua khi biên dịch chương trình.

Xét ví dụ sau:

```
cout << "Xin chao\n"; //lời chào hỏi
```

Thường ta sử dụng chú thích cuối dòng khi muốn giải thích ý nghĩa của một câu lệnh gì đó. Đối với một đoạn chương trình kiểu chú thích giới hạn bởi "/*" và "*/" cho phép mô tả được nhiều thông tin hơn.

3.2 Khai báo mọi nơi

Trong C++ không nhất thiết phải nhóm lên đầu các khai báo đặt bên trong một hàm hay một khối lệnh, mà có thể đặt xen kẽ với các lệnh xử lý. Ví dụ.

```
{
int n;
n=23;
cout <<n<<"\n";
...
int *p=&n;
cout <<p<<"\n";
...
}
```

Giá trị khởi đầu cho các biến có thể thay đổi tại các thời điểm chạy chương trình khác nhau.

Một ví dụ minh hoạ cho khả năng định nghĩa khắp mọi nơi là có thể khai báo các biến điều khiển ngay bên trong các vòng lặp nếu biến đó chưa được khai báo trước đó trong cùng khối lệnh cũng như không được khai báo lại ở phần sau. Xem đoạn chương trình sau:

```
{
        ... //chưa có i
for(int i=0;...;...)
        ... //không được khai báo i
}
```

3.3 Toán tử phạm vi "*::*"

Bình thường, biến cục bộ che lấp biến toàn cục cùng tên. Chẳng hạn:

```
#include <iostream.h>
int x;
main() {
   int x = 10; //x cục bộ
   cout<<x<<"\n";//x cục bộ
}
```

Trong những trường hợp cần thiết, khi muốn truy xuất tới biến toàn cục phải sử dụng toán tử "**::**" trước tên biến:

```
#include <iostream.h>
int x;
main() {
   int x = 10; //x cục bộ
   ::x = 10; //x toàn cục
   cout<<x<<"\n";//x cục bộ
  cout<<::x<<"\n";//x toàn cục
}
```

4. HÀM inline

Trong C++ có thể định nghĩa các hàm được thay thế trực tiếp thành mã lệnh máy tại chỗ gọi (inline) mỗi lần được tham chiếu. Điểm này rất giống với cách hoạt động của các macro có tham số trong C. Ưu điểm của các hàm **inline** là chúng không đòi hỏi các thủ tục bổ sung khi gọi hàm và trả giá trị về. Do vậy hàm **inline** được thực hiện nhanh hơn so với các hàm thông thường.

Một hàm **inline** được định nghĩa và được sử dụng giống như bình thường. Điểm khác nhau duy nhất là phải đặt mô tả **inline** trước khai báo hàm.

Xét ví dụ sau đây:

Ví dụ 2.7

```
#include <iostream.h>
#include <math.h>
#include <conio.h>
inline double norme(double vec[3]); // khai báo hàm inline
void main(){
clrscr();
double v1[3],v2[3];
int i;
for(i=0;i<3;i++) {
  v1[i]=i;v2[i]=2*i-1;}
  cout <<"norme cua v1 : "<<norme(v1) <<" - norme cua v2 : "<<norme(v2);
```

```
  getch();
  }
/*Định nghĩa hàm inline*/
inline double norme(double vec[3])  {
   int i;double s=0;
   for(i=0;i<3;i++)
        s+=vec[i]*vec[i];
   return(sqrt(s));
  }
```

```
norme cua v1 : 2.236068 - norme cua v2 : 3.316625
```

Hàm `norme()` nhằm mục đích tính chuẩn của `vector` với ba thành phần.

Từ khoá **inline** yêu cầu chương trình biên dịch xử lý hàm norme khác với các hàm thông thường. Cụ thể là, mỗi lần gọi `norme()`, trình biên dịch ghép trực tiếp các chỉ thị tương ứng của hàm vào trong chương trình (ở dạng ngôn ngữ máy). Do đó cơ chế quản lý lời gọi và trở về không cần nữa (không cần lưu ngữ cảnh, sao chép các thông số...), nhờ vậy tiết kiệm thời gian thực hiện. Bên cạnh đó các chỉ thị tương ứng sẽ được sinh ra mỗi khi gọi hàm do đó chi phí lưu trữ tăng lên khi hàm được gọi nhiều lần.

Điểm bất lợi khi sử dụng các hàm **inline** là nếu chúng quá lớn và được gọi thường xuyên thì kích thước chương trình sẽ tăng lên rất nhanh. Vì lý do này, chỉ những hàm đơn giản, không chứa các cấu trúc lặp mới được khai báo là hàm **inline**.

Việc sử dụng hàm **inline** so với các macro có tham số có hai điểm lợi. Trước hết hàm **inline** cung cấp một cách có cấu trúc hơn khi mở rộng các hàm đơn giản thành hàm **inline**. Thực tế cho thấy khi tạo một macro có tham số thường hay quên các dấu đóng ngoặc, rất cần đến để đảm bảo sự mở rộng nội tuyến riêng trong mỗi trường hợp. Với macro có thể gây ra hiệu ứng phụ hoặc bị hạn chế khả năng sử dụng. Chẳng hạn với macro:

```
#define square(x) {x++*x++}
```

Với lời gọi

`square(a)`

với `a` là biến sẽ sản sinh ra biểu thức `a++*a++` và kết quả là làm thay đổi giá trị của biến a tới hai lần.

Còn lời gọi

```
square (3)
```

sẽ gây lỗi biên dịch vì ta không thể thực hiện các phép toán tăng giảm trên các toán hạng là hằng số.

Việc sử dụng hàm **inline** như một giải pháp thay thế sẽ tránh được các tình huống như thế.

Ngoài ra, các hàm **inline** có thể được tối ưu bởi chương trình biên dịch.

Điều quan trọng là đặc tả **inline** chỉ là một yêu cầu, chứ không phải là một chỉ thị đối với trình biên dịch. Nếu vì một lý do nào đó trình biên dịch không thể đáp ứng được yêu cầu (chẳng hạn khi bên trong định nghĩa hàm **inline** có các cấu trúc lặp) thì hàm sẽ được biên dịch như một hàm bình thường và yêu cầu **inline** sẽ bị bỏ qua.

Hàm **inline** phải được khai báo bên trong tệp tin nguồn chứa các hàm sử dụng nó. Không thể dịch tách biệt các hàm **inline**. [1]

5. THAM CHIẾU

Ngôn ngữ C++ giới thiệu một khái niệm mới "reference" tạm dịch là "tham chiếu". Về bản chất, tham chiếu là "bí danh" của một vùng nhớ được cấp phát cho một biến nào đó.

Một tham chiếu có thể là một biến, tham số hình thức của hàm hay dùng làm giá trị trả về của một hàm. Các phần tiếp sau lần lượt giới thiệu các khả năng của tham chiếu được sử dụng trong chương trình viết bằng ngôn ngữ C++.

5.1 Tham chiếu tới một biến

Xét hai chỉ thị:

```
int n;
int &p = n;
```

Trong chỉ thị thứ hai, dấu "`&`" để xác định p là một biến tham chiếu còn dấu "=" và tên biến `n` để xác định vùng nhớ mà `p` tham chiếu tới. Lúc này cả hai định danh `p` và `n` cùng xác định vùng nhớ được cấp phát cho biến `n`. Như vậy các chỉ thị sau:

```
n =3;
cout  <<p;
```

---

[1] "Biên dịch tách biệt" cho phép khai báo hàm trong một tệp tiêu đề, còn định nghĩa hàm đó lại ở trong tập tin chương trình sử dụng hàm.

cho kết quả 3 trên thiết bị hiển thị.

Xét về bản chất tham chiếu và tham trỏ giống nhau vì cùng chỉ đến các đối tượng có địa chỉ, cùng được cấp phát địa chỉ khi khai báo. Nhưng cách sử dụng chúng thì khác nhau. Khi nói đến tham chiếu "&p" ta phải gắn nó với một biến nào đó đã khai báo qua "&p=n", trong khi đó khai báo con trỏ "*p" không nhất thiết phải khởi tạo giá trị cho nó. Trong chương trình biến trỏ có thể tham chiếu đến nhiều biến khác nhau còn biến tham chiếu thì "ván đã đóng thuyền" từ khi khai báo, có nghĩa là sau khi khởi tạo cho tham chiếu gắn với một biến nào đó rồi thì ta không thể thay đổi để gắn tam chiếu với một biến khác.

Xét các chỉ thị sau:

Ví dụ 2.8

```
int n=3,m=4;
int *p;
p=&n; //p chỉ đến n
*p=4; //gán giá trị của m cho n thông qua con trỏ p
...
p=&m; //cho p chỉ đến m
int &q=n; //khai báo tham chiếu q chỉ đến n
q=4; //gán cho biến n giá trị 4
...
q=m; //gán giá trị của biến m cho biến n.
```

Nói một cách đơn giản, tham chiếu của một biến giống như bí danh của một con người nào đó. Có nghĩa là để chỉ đến một con người cụ thể nào đó, ta có thể đồng thời sử dụng tên của anh ta hoặc bí danh. Do vậy, để truy nhập đến vùng nhớ tương ứng với một biến, chúng ta có thể sử dụng hoặc là tên biến hoặc là tên tham chiếu tương ứng. Đối với con người, bí danh bao giờ cũng nhằm nói đến một người đã tồn tại, và như vậy tham chiếu cũng phải được khai báo và khởi tạo sau khi biến được khai báo. Chương trình sau đây sẽ gây lỗi biên dịch do tham chiếu y chưa được khởi tạo.

Ví dụ 2.9

Chương trình sai

```
#include <iostream.h>
void main() {
  int x=3, &y;//lỗi vì y phải được khởi tạo
```

```
  cout <<" x = "<<x<<"\n"
       <<" y = "<<y<<"\n";
  y = 7;
  cout <<" x = "<<x<<"\n"
       <<" y = "<<y<<"\n";
  }
```

Chương trình đúng như sau:

```
#include <iostream.h>
void main() {
  int x=3, &y=x;//y bây giờ là một "bí danh" của x
  cout <<" x = "<<x<<"\n"
       <<" y = "<<y<<"\n";
  y = 7;
  cout <<" x = "<<x<<"\n"
       <<" y = "<<y<<"\n";
  }
```

```
x = 3
y = 3
x = 7
y = 7
```

Lưu ý cuối cùng là không thể gắn một tham chiếu với một hằng số trừ trường hợp có từ khoá const đứng trước khai báo tham chiếu. Dễ dàng kiểm tra các nhận xét sau:

```
int &p =3; //không hợp lệ
const int & p =3; //hợp lệ
```

5.2 Truyền tham số cho hàm bằng tham chiếu

Trong C, các tham số và giá trị trả về của một hàm được truyền bằng giá trị. Để giả lập cơ chế truyền tham biến ta phải sử dụng con trỏ.

Trong C++, việc dùng khái niệm tham chiếu trong khai báo tham số hình thức của hàm sẽ yêu cầu chương trình biên dịch truyền địa chỉ của biến cho hàm và hàm sẽ thao tác trực tiếp trên các biến đó. Chương trình sau đưa ra ba cách viết khác nhau của hàm thực hiện việc hoán đổi nội dung của hai biến.

Ví dụ 2.10

```
/*swap.cpp*/
#include <conio.h>
#include <iostream.h>
/*Hàm swap1 được gọi với các tham số được truyền theo tham trị*/
void swap1(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}
/*Hàm swap2 thực hiện việc truyền tham số bằng tham trỏ*/
void swap2(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}
/*Hàm swap3 thực hiện việc truyền tham số bằng tham chiếu*/
void swap3(int &x, int &y) {
  int temp = x;
  x = y;
  y = temp;
}
void main() {
  int a=3, b=4;
  clrscr();
  cout <<"Truoc khi goi swap1:\n";
  cout<<"a = "<<a<<" b = "<<b<<"\n";
  swap1(a,b);
  cout <<"Sau khi goi swap:\n";
  cout<<"a = "<<a<<" b = "<<b<<"\n";
```

```
  a=3; b =4;
  cout <<"Truoc khi goi swap2:\n";
  cout<<"a = "<<a<<" b = "<<b<<"\n";
  swap2(&a,&b);
  cout <<"Sau khi goi swap2:\n";
  cout<<"a = "<<a<<" b = "<<b<<"\n";
  a=3;b=4;
  cout <<"Truoc khi goi swap3:\n";
  cout<<"a = "<<a<<" b = "<<b<<"\n";
  swap3(a,b);
  cout <<"Sau khi goi swap3:\n";
  cout<<"a = "<<a<<" b = "<<b<<"\n";
  getch();
}
```

```
Truoc khi goi swap1:
a = 3 b = 4
Sau khi goi swap1:
a = 3 b = 4
Truoc khi goi swap2
a = 3 b = 4
Sau khi goi swap2:
a = 4 b = 3
Truoc khi goi swap3:
a = 3 b = 4
Sau khi goi swap3:
a = 4 b = 3
```

Trong chương trình trên, ta truyền tham số `a,b` cho hàm `swap1()` theo tham trị cho nên giá trị của chúng không thay đổi trước và sau khi gọi hàm.

Giải pháp đưa ra trong hàm `swap2()` là thay vì truyền trực tiếp giá trị hai biến a và b người ta truyền địa chỉ của chúng rồi thông qua các địa chỉ này để xác định giá trị biến. Bằng cách đó giá trị của hai biến a và b sẽ hoán đổi cho nhau sau lời gọi hàm. Khác với `swap2()`, hàm `swap3()` đưa ra giải pháp sử dụng tham chiếu. Các tham số hình thức của hàm `swap3()` bây giờ là các tham chiếu đến các tham số thực được truyền cho hàm. Nhờ vậy mà giá trị của hai tham số thực a và b có thể hoán đổi được cho nhau.

Có một vấn đề mà chắc rằng bạn đọc sẽ phân vân: "làm thế nào để khởi tạo các tham số hình thức là tham chiếu". Xin thưa rằng điều đó được thực hiện tự động trong mỗi lời gọi hàm chứ không phải trong định nghĩa. Từ đó nảy sinh thêm một nhận xét quan trọng: "tham số ứng với tham số hình thức là tham chiếu phải là biến trừ trường hợp có từ khoá const đứng trước khai báo tham số hình thức". Chẳng hạn không thể thực hiện lời gọi hàm sau:

```
swap3(2,3);
```

Bạn đọc có thể xem thêm phần "Hàm thiết lập sao chép lại" để thấy được ích lợi to lớn của việc truyền tham số cho hàm bằng tham chiếu.

Chú ý: khi muốn truyền bằng tham biến một biến trỏ thì viết như sau:

```
int* & adr; //adr là một tham chiếu tới một biến trỏ chỉ đến một biến nguyên.
```

5.3 Giá trị trả về của hàm là tham chiếu

Định nghĩa của hàm có dạng như sau:

```
<type> & fct( ...) {
...
return <bien co pham vi toan cuc>;
}
```

Trong trường hợp này biểu thức được trả lại trong câu lệnh **return** phải là tên của một biến xác định từ bên ngoài hàm, bởi vì chỉ khi đó mới có thể sử dụng được giá trị của hàm. Khi ta trả về một tham chiếu đến một biến cục bộ khai báo bên trong hàm, biến cục bộ này sẽ bị mất đi khi kết thúc thực hiện hàm và do vậy, tham chiếu của hàm cũng không còn có ý nghĩa nữa.

Khi giá trị trả về của hàm là tham chiếu, ta có thể gặp các câu lệnh gán "kỳ dị" trong đó vế trái là một lời gọi hàm chứ không phải là tên của một biến. Điều này hoàn toàn hợp lý, bởi lẽ bản thân hàm đó có giá trị trả về là một tham chiếu. Nói cách khác, vế trái của lệnh gán (biểu thức gán) có thể là lời gọi đến một hàm có giá trị trả về là một tham chiếu.

Xét ví dụ sau đây:

Ví dụ 2.11

```
/*fr.cpp*/
#include <iostream.h>
#include <conio.h>
int a[5];
```

```
int &fr(int *d,int i);
void main() {
  clrscr();
  cout<<"Nhap gia tri cho mang a:\n";
  for(int i=0;i<5;i++) {
    cout<<"a["<<i<<"]= ";
    cin>>fr(a,i);
    }
  cout<<"Mang a sau khi nhap\n";
  for(i=0;i<5;i++)
    cout<<a[i]<<" ";
  cout<<"\n";
  getch();
  }

int &fr(int *d,int i) {
  return d[i];
  }
```

```
Nhap gia tri cho mang a:
a[0]= 6
a[1]= 4
a[2]= 3
a[3]= 5
a[4]= 6
Mang a sau khi nhap
6 4 3 5 6
```

Bạn đọc có thể xem thêm phần "Định nghĩa chồng toán tử" để thấy được lợi ích của vấn đề trả về tham chiếu cho hàm.

6. ĐỊNH NGHĨA CHỒNG HÀM (Overloading functions)

C++ cho phép sử dụng một tên cho nhiều hàm khác nhau ta gọi đó là sự "chồng hàm". Trong trường hợp đó, các hàm sẽ khác nhau ở giá trị trả về và danh sách kiểu các tham số. Chẳng hạn chúng ta muốn định nghĩa các hàm trả về số nhỏ nhất trong:

11. hai số nguyên
12. hai số thực
13. hai ký tự
14. ba số nguyên
15. một dãy các số nguyên ...

Dĩ nhiên có thể tìm cho mỗi hàm như vậy một tên phân. Lợi dụng khả năng "định nghĩa chồng hàm" của C++, chúng ta có thể viết các hàm như sau:

Ví dụ 2.12

```
#include <iostream.h>
//Hàm nguyên mẫu
int min(int, int); //Hàm 1
double min(double, double);//Hàm 2
char min(char, char);//Hàm 3
int min(int, int, int);//Hàm 4
int min(int, int *);//Hàm 5
main() {
int n=10, p =12, q = -12;
double x = 2.3, y = -1.2;
char c = 'A', d= 'Q';
int td[7] = {1,3,4,-2,0,23,9};
cout<<"min (n,p) : "<<min(n,p)<<"\m"; //Hàm 1
cout<<"min (n,p,q) : "<<min(n,p,q)<<"\m"; //Hàm 4
cout<<"min (c,d) : "<<min(c,d)<<"\m"; //Hàm 3
cout<<"min (x,y) : "<<min(n,p)<<"\m"; //Hàm 2
cout<<"min (td) : "<<min(7,td)<<"\m"; //Hàm 5
cout<<"min (n,x) : "<<min(n,x)<<"\m"; //Hàm 2
//cout<<"min (n,p,x) : "<<min(n,p,x)<<"\m"; //Lỗi
}
int min(int a, int b) {
  return (a> b? a: b);
}
int min(int a, int b, int c) {
```

```
  return (min(min(a,b),c));
}
double min(double a, double b) {
  return (a> b? a: b);
}
char min(char a, char b) {
  return (a> b? a: b);
}
int min(int n, int *t) {
int res = t[0];
for (int i=1; i<n; i++)
   res = min(res,t[i]);
return res;
}
```

Nhận xét

16. Một hàm có thể gọi đến hàm cùng tên với nó (ví dụ như hàm 4,5 gọi hàm 1).

17. Trong trường hợp có các hàm trùng tên trong chương trình, việc xác định hàm nào được gọi do chương trình dịch đảm nhiệm và tuân theo các nguyên tắc sau:

Trường hợp các hàm có một tham số

Chương trình dịch tìm kiếm "sự tương ứng nhiều nhất" có thể được; có các mức độ tương ứng như sau (theo độ ưu tiên giảm dần):

a) Tương ứng thật sự: ta phân biệt các kiểu dữ liệu cơ sở khác nhau đồng thời lưu ý đến cả dấu.

b) Tương ứng dữ liệu số nhưng có sự chuyển đổi kiểu dữ liệu tự động ("numeric promotion"): **char** và **short** -->**int**; **float** -->**int.**

c) Các chuyển đổi kiểu chuẩn được C và C++ chấp nhận.

d) Các chuyển đổi kiểu do người sử dụng định nghĩa.

Quá trình tìm kiếm bắt đầu từ mức cao nhất và dừng lại ở mức đầu tiên cho phép tìm thấy sự phù hợp. Nếu có nhiều hàm phù hợp ở cùng một mức, chương trình dịch đưa ra thông báo lỗi do không biết chọn hàm nào giữa các hàm phù hợp.

Ý tưởng chung là phải tìm một hàm phù hợp nhất so với tất cả những hàm còn lại. Để đạt mục đích này, chương trình dịch chọn cho mỗi tham số các hàm phù hợp (ở tất cả các mức độ). Trong số các hàm được lựa chọn, chương trình dịch chọn ra (nếu tồn tại và tồn tại duy nhất) hàm sao cho đối với mỗi đối số nó đạt được sự phù hợp hơn cả so với các hàm khác.

Trong trường hợp vẫn có nhiều hàm thoả mãn, lỗi biên dịch xảy ra do chương trình dịch không biết chọn hàm nào trong số các hàm thỏa mãn. Đặc biệt lưu ý khi sử dụng định nghĩa chồng hàm cùng với việc khai báo các hàm với tham số có giá trị ngầm định sẽ được trình bày trong mục tiếp theo.

7. THAM SỐ NGẦM ĐỊNH TRONG LỜI GỌI HÀM

Ta xét ví dụ sau:

Ví dụ 2.13

```
#include <iostream.h>
void main() {
int n=10,p=20;
void fct(int, int = 12) ;//khai báo hàm với một giá trị ngầm định
fct(n,p); //lời gọi thông thường, có hai tham số
fct(n); //lời gọi chỉ với một tham số
//fct() sẽ không được chấp nhận
}
//khai báo bình thường
void fct(int a, int b)  {
cout <<"tham so thu nhat : " <<a <<"\n";
cout<<"tham so thu hai    : "<<b<<"\n";
}
```

```
tham so thu nhat : 10
tham so thu hai   : 20
tham so thu nhat : 10
tham so thu hai   : 12
```

Trong khai báo của `fct()` bên trong hàm `main()` :

```
void fct(int,int =12);
```

khai báo

```
int = 12
```

chỉ ra rằng trong trường hợp vắng mặt tham số thứ hai ở lời gọi hàm fct() thì tham số hình thức tương ứng sẽ được gán giá trị ngầm định 12.

Lời gọi

```
fct();
```

không được chấp nhận bởi vì không có giá trị ngầm định cho tham số thứ nhất.

Ví dụ 2.14

```
#include <iostream.h>
void main(){
int n=10,p=20;
void fct(int = 0, int = 12);//khai báo hàm với hai tham số có giá trị ngầm định
fct(n,p); //lời gọi thông thường, có hai tham số
fct(n);    //lời gọi chỉ với một tham số
fct() ;    //fct() đã được chấp nhận
}
void fct(int a, int b)   //khai báo bình thường
{
cout<<"tham so thu nhat : " <<a <<"\n";
cout<<"tham so thu hai    : "<<b<<"\n";
}
```

```
tham so thu nhat : 10
tham so thu hai   : 20
tham so thu nhat : 10
tham so thu hai   : 12
tham so thu nhat : 0
tham so thu hai   : 12
```

Chú ý

18. Các tham số với giá trị ngầm định phải được đặt ở cuối trong danh sách các tham số của hàm để tránh nhầm lẫn các giá trị.
19. Các giá trị ngầm định của tham số được khai báo khi sử dụng chứ không phải trong phần định nghĩa hàm. Ví dụ sau đây gây ra lỗi biên dịch:

Ví dụ 2.15

```
#include <conio.h>
#include <iostream.h>
void f();
void main() {
  clrscr();
  int n=10,p=20;
  void fct(int =0,int =12);
  cout<<"Goi fct trong main\n";
  fct(n,p);
  fct(n);
  fct();
  getch();
  }
void fct(int a=10,int b=100) {
  cout<<"Tham so thu nhat : "<<a<<"\n";
  cout<<"Tham so thu hai  : "<<b<<"\n";
  }
```

20. Nếu muốn khai báo giá trị ngầm định cho một tham số biến trỏ, thì phải chú ý viết * và = cách xa nhau ít nhất một dấu cách.
21. Các giá trị ngầm định có thể là một biểu thức bất kỳ (không nhất thiết chỉ là biểu thức hằng), có giá trị được tính tại thời điểm khai báo:
    ```
    float x;
    int n;
    void fct(float = n*2+1.5);
    ```
22. Chồng hàm và gọi hàm với tham số có giá trị ngầm định có thể sẽ dẫn đến lỗi biên dịch khi chương trình dịch không xác định được hàm phù hợp.

Xét ví dụ sau:

Ví dụ 2.16

```
#include <iostream.h>
void fct(int, int=10);
void fct(int);
void main() {
```

```
int n=10, p=20;
fct(n,p);//OK
fct(n);//ERROR
}
```

8. BỔ SUNG THÊM CÁC TOÁN TỬ QUẢN LÝ BỘ NHỚ ĐỘNG: new và delete

8.1 Toán tử cấp phát bộ nhớ động new

Với khai báo

```
int * adr;
```

chỉ thị

```
ad = new int;
```

cho phép cấp phát một vùng nhớ cần thiết cho một phần tử có kiểu int và gán cho adr địa chỉ tương ứng. Lệnh tương đương trong C:

```
ad =(int *) malloc(sizeof(int));
```

Với khai báo

```
char *adc;
```

chỉ thị

```
adc =new char [100];
```

cho phép cấp phát vùng nhớ đủ cho một bảng chứa 100 ký tự và đặt địa chỉ đầu của vùng nhớ cho biến adc. Lệnh tương ứng trong C như sau:

```
adc =(char *)malloc(100);
```

Hai cách sử dụng **new** như sau:

Dạng 1

```
new type;
```

giá trị trả về là

(i) một con trỏ đến vị trí tương ứng khi cấp phát thành công

(ii) NULL trong trường hợp trái lại.

Dạng 2

```
new type[n];
```

trong đó n là một biểu thức nguyên không âm nào đó, khi đó toán tử **new** xin cấp phát vùng nhớ đủ để chứa n thành phần kiểu `type` và trả lại con trỏ đến đầu vùng nhớ đó nếu như cấp phát thành công.

8.2 Toán tử giải phóng vùng nhớ động *delete*

Một vùng nhớ động được cấp phát bởi **new** phải được giải phóng bằng **delete** mà không thể dùng `free` được, chẳng hạn:

```
delete adr;
delete adc;
```

Ví dụ 2.17

Cấp phát bộ nhớ động cho mảng hai chiều

```
#include<iostream.h>
void Nhap(int **mat);//nhập ma trận hai chiều
void In(int **mat);//In ma trận hai chiều
void main() {
  int **mat;
  int i;
  /*cấp phát mảng 10 con trỏ nguyên*/
  mat = new int *[10];
  for(i=0; i<10; i++)
    /*mỗi con trỏ nguyên xác định vùng nhớ 10 số nguyên*/
    mat[i] = new int [10];
  /*Nhập số liệu cho mảng vừa được cấp phát*/
  cout<<"Nhap so lieu cho matran 10*10\n";
  Nhap(mat);
  /*In ma trận*/
  cout<<"Ma tran vua nhap \n";
  In(mat);
  /*Giải phóng bộ nhớ*/
  for(i=0;i<10;i++)
```

```
    delete mat[i];
  delete mat;
 }
void Nhap(int ** mat) {
 int i,j;
 for(i=0; i<10;i++)
   for(j=0; j<10;j++) {
    cout<<"Thanh phan thu ["<<i<<"]["<<j<<"]= ";
    cin>>mat[i][j];
    }
 }
void In(int ** mat) {
 int i,j;
 for(i=0; i<10;i++) {
   for(j=0; j<10;j++)
    cout<<mat[i][j]<<" ";
   cout<<"\n";
    }
 }
```

Ví dụ 2.18

Quản lý tràn bộ nhớ `set_new_handler`

```
#include <iostream>
main()
{
void outof();
set_new_handler(&outof);
long taille;
int *adr;
int nbloc;
cout<<"Kich thuoc can nhap? ";
```

```
cin >>taille;
for(nbloc=1;;nbloc++)
  {
   adr =new int [taille];
   cout <<"Cap phat bloc so : "<<nbloc<<"\n";
  }
}
void outof()//hàm được gọi khi thiếu bộ nhớ
{
cout <<"Het bo nho -Ket thuc \n";
exit(1);
}
```

9. TÓM TẮT

9.1 Ghi nhớ

C++ là một sự mở rộng của C(superset), do đó có thể sử dụng một chương trình biên dịch C++ để dịch và thực hiện các chương trình nguồn viết bằng C.

C yêu cầu các chú thích nằm giữa /* và */. C++ còn cho phép tạo một chú thích bắt đầu bằng "//" cho đến hết dòng.

C++ cho phép khai báo khá tuỳ ý. Thậm chí có thể khai báo biến trong phần khởi tạo của câu lệnh lặp for.

C++ cho phép truyền tham số cho hàm bằng tham chiếu. Điều này tương tự như truyền tham biến cho chương trình con trong ngôn ngữ PASCAL. Trong lời gọi hàm ta dùng tên biến và biến đó sẽ được truyền cho hàm qua tham chiếu. Điều đó cho phép thao tác trực tiếp trên biến được truyền chứ không phải gián tiếp qua biến trỏ.

Toán tử **new** và **delete** trong C++ được dùng để quản lý bộ nhớ động thay vì các hàm cấp phát động của C.

C++ cho phép người viết chương trình mô tả các giá trị ngầm định cho các tham số của hàm, nhờ đó hàm có thể được gọi với một danh sách các tham số không đầy đủ.

Toán tử "**::**" cho phép truy nhập biến toàn cục khi đồng thời sử dụng biến cục bộ và toàn cục trùng tên.

Có thể định nghĩa các hàm cùng tên với các tham số khác nhau. Hai hàm cùng tên sẽ được phân biệt nhờ giá trị trả về và danh sách kiểu các tham số.

9.2 Các lỗi thường gặp

Quên đóng */ cho các chú thích.

Khai báo biến sau khi biến được sử dụng.

Sử dụng lệnh **return** để trả về giá trị nhưng khi định nghĩa lại mô tả hàm kiểu **void** hoặc ngược lại, quên câu lệnh này trong trường hợp hàm yêu cầu có giá trị trả về.

Không có hàm nguyên mẫu cho các hàm.

Bỏ qua việc khởi tạo cho các tham chiếu.

Thay đổi giá trị của biến hằng.

Tạo các hàm cùng tên, cùng tham số.

9.3 Một số thói quen lập trình tốt

Sử dụng "//" để tránh lỗi không đóng */ khi chú thích nằm gọn trong một dòng.

Sử dụng các khả năng vào ra mới của C++ để chương trình dễ đọc hơn.

Đặt các khai báo biến lên đầu các khối lệnh.

Chỉ dùng từ khoá inline với các hàm "nhỏ", "không phức tạp"

Sử dụng con trỏ để truyền tham số cho hàm khi cần thay đổi giá trị tham số, còn tham chiếu dùng để truyền các tham số có kích thước lớn mà không có nhu cầu thay đổi nội dung.

Tránh sử dụng biến cùng tên cho nhiều mục đích khác nhau trong chương trình.

10. BÀI TẬP

Bài tập 2.1

Sử dụng cin và cout viết chương trình nhập vào một dãy số nguyên rồi sắp xếp dãy số đó tăng dần.

Bài tập 2.2

Sử dụng **new** và **delete** thực hiện các thao tác cấp phát bộ nhớ cho một mảng động hai chiều. Hoàn thiện chương trình bằng cách thực hiện các thao liên quan đến ma trận vuông.

Bài tập 2.3

Lập chương trình mô phỏng các hoạt động trên một ngăn xếp chứa các số nguyên.