CHƯƠNG 6

KHUÔN HÌNH
(TEMPLATE)

Mục đích chương này:

1. Hiểu được lợi ích của việc sử dụng khuôn hình hàm và khuôn hình lớp để viết chương trình.
2. Biết cách tạo và sử dụng một khuôn hình hàm và khuôn hình lớp.
3. Khái niệm các tham số kiểu và các tham số biểu thức trong khuôn hình hàm, khuôn hình lớp.
4. Định nghĩa chồng khuôn hình hàm.
5. Cụ thể hoá một khuôn hình hàm, một hàm thành phần của khuôn hình lớp.
6. Thuật toán sản sinh một thể hiện hàm (hàm thể hiện) của một khuôn hình hàm
7. Các vấn đề khác của lập trình hướng đối tượng liên quan đến khuôn hình lớp.

KHUÔN HÌNH HÀM

Khuôn hình hàm là gì?

Ta đã biết định nghĩa chồng hàm cho phép dùng một tên duy nhất cho nhiều hàm thực hiện các công việc khác nhau. Khái niệm khuôn hình hàm cũng cho phép sử dụng cùng một tên duy nhất để thực hiện các công việc khác nhau, tuy nhiên so với định nghĩa chồng hàm, nó có phần mạnh hơn và chặt chẽ hơn; mạnh hơn vì chỉ cần viết định nghĩa khuôn hình hàm một lần, rồi sau đó chương trình biên dịch làm cho nó thích ứng với các kiểu dữ liệu khác nhau; chặt chẽ hơn bởi vì dựa theo khuôn hình hàm, tất cả các hàm thể hiện được sinh ra bởi trình biên dịch sẽ tương ứng với cùng một định nghĩa và như vậy sẽ có cùng một giải thuật.

Tạo một khuôn hình hàm

Giả thiết rằng chúng ta cần viết một hàm `min` đưa ra giá trị nhỏ nhất trong hai giá trị có cùng kiểu. Ta có thể viết một định nghĩa như thế đối với kiểu `int` như sau:

```
int min (int a, int b) {
  if (a < b) return a;
```

```
    else return b;
}
```

Giả sử, ta lại phải viết định nghĩa hàm `min()` cho kiểu **double, float, char, char*** ...

```
float min(float a, float b) {
  if (a < b) return a;
    else b; }
```

Nếu tiếp tục như vậy, sẽ có khuynh hướng phải viết rất nhiều định nghĩa hàm hoàn toàn tương tự nhau; chỉ có kiểu dữ liệu các tham số là thay đổi. Các chương trình biên dịch C++ hiện có cho phép giải quyết đơn giản vấn đề trên bằng cách định nghĩa một khuôn hình hàm duy nhất theo cách như sau:

```
#include <iostream.h>
//tạo một khuôn hình hàm
template <class T> T min(T a, T b) {
  if (a < b) return a;
    else return b;
  }
```

So sánh với định nghĩa hàm thông thường, ta thấy chỉ có dòng đầu tiên bị thay đổi:

```
template <class T> T min (T a, T b)
```

trong đó

`template<class T>`

xác định rằng đó là một khuôn hình với một tham số kiểu T;

Phần còn lại

`T min(T a, T b)`

nói rằng, `min()` là một hàm với hai tham số hình thức kiểu `T` và có giá trị trả về cũng là kiểu `T`.

Sử dụng khuôn hình hàm

Khuôn hình hàm cho kiểu dữ liệu cơ sở

Để sử dụng khuôn hình hàm `min()` vừa tạo ra, chỉ cần sử dụng hàm `min()` trong những điều kiện phù hợp (ở đây có nghĩa là hai tham số của hàm có cùng kiểu dữ liệu). Như vậy, nếu trong một chương trình có hai tham số nguyên `n` và `p`, với lời gọi `min(n,p)` chương trình biên dịch sẽ tự động sản sinh ra hàm `min()` (ta gọi là một hàm thể hiện) tương ứng với hai tham số kiểu nguyên **int**. Nếu chúng ta gọi `min()` với hai tham số kiểu **float**, chương trình biên dịch cũng sẽ tự động sản sinh một hàm thể hiện `min` khác tương ứng với các tham số kiểu **float** và cứ thế. Sau đây là một ví dụ hoàn chỉnh:

Ví dụ 6.1

```
/*template1.cpp*/
#include <iostream.h>
#include <conio.h>
//tạo một khuôn hình hàm
template <class T> T min(T a, T b) {
  if ( a < b) return a;
     else return b;
  }
//ví dụ sử dụng khuôn hình hàm min
void main() {
 clrscr();
 int n = 4, p = 12;
 float x = 2.5, y= 3.25;
 cout<<"min (n, p) = "<<min (n, p)<<"\n";//int min(int, int)
 cout<<"min (x, y) = "<<min (x, y)<<"\n";//float min(float, float)
 getch();
 }
```

```
min(n, p) = 4
min(x, y) = 2.5
```

Khuôn hình hàm min cho kiểu char *

```
/*template2.cpp*/
#include <iostream.h>
#include <conio.h>
template <class T> T min (T a, T b) {
 if (a < b) return a;
   else return b;
 }
void main() {
 clrscr();
 char * adr1 = "DHBK";
 char * adr2 = "CDSD";
 cout << "min (adr1, adr2) ="<<min (adr1, adr2);
 getch();
 }
```

```
min (adr1, adr2) = DHBK
```

Kết quả khá thú vị vì ta hy vọng hàm `min()` trả về xâu `"CDSD"`. Thực tế, với biểu thức `min(adr1, adr2)` , chương trình biên dịch đã sinh ra hàm thể hiện sau đây:

```
char * min(char * a, char * b) {
 if (a < b) return a;
   else return b;
 }
```

Việc so sánh `a < b` thực hiện trên các giá trị biến trỏ (ở đây trong các khuôn hình máy PC ta luôn luôn có `a < b`). Ngược lại việc hiển thị thực hiện bởi toán tử << sẽ đưa ra xâu ký tự trỏ bởi con trỏ ký tự.

Khuôn hình hàm min với kiểu dữ liệu lớp

Để áp dụng khuôn hình hàm `min()` ở trên với kiểu lớp, cần phải định nghĩa lớp sao cho có thể áp dụng phép toán so sánh "<" với các đối tượng của lớp này, nghĩa là ta phải định nghĩa một hàm toán tử `operator <` cho lớp. Sau đây là một ví dụ minh hoạ:

Ví dụ 6.2

```
/*template3.cpp*/
#include <iostream.h>
#include <conio.h>
//khuôn hình hàm min
template <class T> T min( T a, T b) {
  if (a < b) return a;
     else return b;
  }
//lớp vect
class vect {
   int x, y;
 public:
   vect(int abs =0, int ord = 0) { x= abs, y= ord;}
   void display() { cout <<x<<" "<<y<<"\n"; }
   friend int operator < (vect , vect);
 };
int operator < (vect a, vect b) {
 return a.x*a.x + a.y*a.y < b.x*b.x + b.y*b.y;
 }
void main() {
 clrscr();
 vect u(3,2),v(4,1);
 cout<<"min (u, v) = ";
 min(u,v).display();
 getch();
 }
```

```
min (u, v) = 3 2
```

Nếu ta áp dụng khuôn hình hàm `min()` đối với một lớp mà chưa định nghĩa toán tử "<", chương trình biên dịch sẽ đưa ra một thông báo lỗi tương tự như việc định nghĩa một hàm `min()` cho kiểu lớp đó.

Các tham số kiểu của khuôn hình hàm

Phần này trình bày cách đưa vào các tham số kiểu trong một khuôn hình hàm, để chương trình biên dịch sản sinh một hàm thể hiện.

Các tham số kiểu trong định nghĩa khuôn hình hàm

Một cách tổng quát, khuôn hình hàm có thể có một hay nhiều tham số kiểu, với mỗi tham số này có từ khoá **class** đi liền trước, chẳng hạn như:

```
template <class T, class U> int fct (T a, T *b, U c) {... }
```

Các tham số này có thể để ở bất kỳ đâu trong định nghĩa của khuôn hình hàm, nghĩa là:

Trong dòng tiêu đề (như đã chỉ ra trong ví dụ trên).

Trong các khai báo các biến cục bộ.

(i) Trong các chỉ thị thực hiện.

Chẳng hạn:

```
template <class T, class U> int fct (T a, T *b, U c) {
T x; //biến cục bộ x kiểu T
U *adr; //biến cục bộ adr kiểu U *
...
adr = new T [10];//cấp phát một mảng 10 thành phần kiểu T ... n = sizeof (T);
}
```

Ta xem chương trình sau:

Ví dụ 6.3

```
/*templat4.cpp*/
#include <iostream.h>
#include <conio.h>
template <class T, class U> T fct(T x, U y, T z) {
 return x + y + z;
 }
void main() {
 clrscr();
 int n= 1, p = 2, q = 3;
```

```
 float x =2.5, y = 5.0;
 cout <<fct( n, x, p)<<"\n";// (int) 5
 cout <<fct(x, n, y)<<"\n"; // (float)8.5
 cout <<fct(n, p, q)<<"\n"; // (int) 6
    //cout <<fct(n, p, x)<<"\n"; // lỗi
 getch();
 }
```

Trong mọi trường hợp, mỗi tham số kiểu phải xuất hiện ít nhất một lần trong khai báo danh sách các tham số hình thức của khuôn hình hàm. Điều đó hoàn toàn logic bởi vì nhờ các tham số này, chương trình dịch mới có thể sản sinh ra hàm thể hiện cần thiết. Điều gì sẽ xảy ra nếu trong danh sách các tham số của khuôn hình hàm không có đủ các tham số kiểu? Hiển nhiên khi đó chương trình dịch không thể xác định các tham số kiểu dữ liệu thực ứng với các tham số kiểu hình thức trong `template<...>`.

Khuôn hình hàm sau đây thực hiện trao đổi nội dung của hai biến.

Ví dụ 6.4

```
/*templat5.cpp*/
#include <iostream.h>
#include <conio.h>
//định nghĩa khuôn hình hàm đổi chỗ nội dung hai biến với kiểu bất kỳ
template <class X> void swap(X &a, X &b) {
 X temp;
 temp=a;
 a=b;
 b=temp;
 }
void main() {
 clrscr();
 int i=10, j=20;
 float x=10.1, y=23.1;
 cout<<"I J ban dau: "<<i<<" "<<j<<endl;
 cout<<"X Y ban dau: "<<x<<" "<<y<<endl;
 swap(i,j);//đổi chỗ hai số nguyên
```

```
 swap(x,y); //đổi chỗ hai số nguyên
 cout<<"I J sau khi doi cho: "<<i<<" "<<j<<endl;
 cout<<"X Y sau khi doi cho: "<<x<<" "<<y<<endl;
 getch();
 }
```

```
I J ban dau: 10 20
X Y ban dau: 10.1 23.1
I J sau khi doi cho: 20 10
X Y sau khi doi cho: 23.1 10.1
```

Giải thuật sản sinh một hàm thể hiện

Trở lại khuôn hình hàm `min()`:

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

Với các khai báo:

```
int n;
char c;
```

câu hỏi đặt ra là: chương trình dịch sẽ làm gì khi gặp lời gọi kiểu như là `min(n,c)`? Câu trả lời dựa trên hai nguyên tắc sau đây:

(ii) C++ quy định phải có một sự tương ứng chính xác giữa kiểu của tham số hình thức và kiểu tham số thực sự được truyền cho hàm, tắc là ta chỉ có thể sử dụng khuôn hình hàm `min()` trong các lời gọi với hai tham số có cùng kiểu. Lời gọi `min(n, c)` không được chấp nhận và sẽ gây ra lỗi biên dịch.

(iii) C++ thậm chí còn không cho phép các chuyển kiểu thông thường như là: `T` thành const `T` hay `T[]` thành `T *`, những trường hợp hoàn toàn được phép trong định nghĩa chồng hàm.

Ta tham khảo đoạn chương trình sau đây:

```
int n;
char c;
unsigned int q;
```

```
const int r = 10;
int t[10];
int *adi;
...
min (n, c) //lỗi
min (n, q) //lỗi
min (n, r) //lỗi
min (t, adi) //lỗi
```

Khởi tạo các biến có kiểu dữ liệu chuẩn

Trong khuôn hình hàm, tham số kiểu có thể tương ứng khi thì một kiểu dữ liệu chuẩn, khi thì một kiểu dữ liệu lớp. Sẽ làm gì khi ta cần phải khai báo bên trong khuôn hình hàm một đối tượng và truyền một hay nhiều tham số cho hàm thiết lập của lớp. Xem ví dụ sau đây:

```
template <class T> fct(T a) {
  T x(3);//x là một đối tượng cục bộ kiểu T mà chúng ta xây dựng bằng cách
         //truyền giá trị 3 cho hàm thiết lập ...
  }
```

Khi sử dụng hàm `fct()` cho một kiểu dữ liệu lớp, mọi việc đều tốt đẹp. Ngược lại, nếu chúng ta cố gắng áp dụng cho một kiểu dữ liệu chuẩn, chẳng hạn như **int**, khi đó chương trình dịch sản sinh ra hàm sau đây:

```
fct( int a) { int x(3); ... }
```

Để cho chỉ thị

```
int x(3) ;
```

không gây ra lỗi, C++ đã ngầm hiểu câu lệnh đó như là phép khởi tạo biến `x` với giá trị `3`, nghĩa là:

```
int x = 3;
```

Một cách tương tự:

```
double x(3.5); //thay vì double x = 3.5;
char c('e'); //thay vì char c = 'e';
```

Các hạn chế của khuôn hình hàm

Về nguyên tắc, khi định nghĩa một khuôn hình hàm, một tham số kiểu có thể tương ứng với bất kỳ kiểu dữ liệu nào, cho dù đó là một kiểu chuẩn hay một kiểu lớp do người dùng định nghĩa. Do vậy không thể hạn chế việc thể hiện đối với một số kiểu dữ liệu cụ thể nào đó. Chẳng hạn, nếu một khuôn hình hàm có dòng đầu tiên:

```
template <class T> void fct(T)
```

chúng ta có thể gọi `fct()` với một tham số với kiểu bất kỳ: **int**, **float**, **int** *, **int** **, ..., *t** (`t` là một kiểu dữ liệu nào đấy)

Tuy nhiên, chính định nghĩa bên trong khuôn hình hàm lại chứa một số yếu tố có thể làm cho việc sản sinh hàm thể hiện không đúng như mong muốn. Ta gọi đó là các hạn chế của các khuôn hình hàm.

Đầu tiên, chúng ta có thể cho rằng một tham số kiểu có thể tương ứng với một con trỏ. Do đó, với dòng tiêu đề:

```
template <class T> void fct(T *)
```

ta chỉ có thể gọi `fct()` với một con trỏ đến một kiểu nào đó: **int***, **int** **, `t *`, `t **`.

Trong các trường hợp khác, sẽ gây ra các lỗi biên dịch. Ngoài ra, trong định nghĩa của một khuôn hình hàm, có thể có các chỉ thị không thích hợp đối với một số kiểu dữ liệu nhất định. Chẳng hạn, khuôn hình hàm:

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

không thể dùng được nếu `T` tương ứng với một kiểu lớp trong đó phép toán "<" không được định nghĩa chồng. Một cách tương tự với một khuôn hình hàm kiểu:

```
template <class T> void fct(T) {
  ... T x(2, 5); /*đối tượng cục bộ được khởi tạo bằng một hàm thiết lập với
                                                    hai tham số*/
  }
```

không thể áp dụng cho các kiểu dữ liệu lớp không có hàm thiết lập với hai tham số.

Tóm lại, mặc dù không tồn tại một cơ chế hình thức để hạn chế khả năng áp dụng của các khuôn hình hàm, nhưng bên trong mỗi một khuôn hình hàm đều có

chứa những nhân tố để người ta có thể biết được khuôn hình hàm đó có thể được áp dụng đến mức nào.

Các tham số biểu thức của một khuôn hình hàm

Trong định nghĩa của một khuôn hình hàm có thể khai báo các tham số hình thức với kiểu xác định. Ta gọi chúng là các tham số biểu thức. Chương trình `templat6.cpp` sau đây định nghĩa một khuôn hình hàm cho phép đếm số lượng các phần tử nul (0 đối với các giá trị số hoặc NULL nếu là con trỏ) trong một bảng với kiểu bất kỳ và kích thước nào đó:

Ví dụ 6.5

```
/*templat6.cpp*/
#include <iostream.h>
#include <conio.h>
template <class T> int compte(T * tab, int n) {
 int i, nz = 0;
 for (i=0; i<n; i++)
  if (!tab[i])
   nz++;
 return nz;
 }
void main() {
 clrscr();
 int t[5] = {5, 2, 0, 2, 0};
 char c[6] = { 0, 12, 0, 0, 0};
 cout<<" compte (t) = "<<compte(t, 5)<<"\n";
 cout<<" compte (c) = "<<compte(c,6)<<"\n";
 getch();
 }
```

```
compte (t) = 2
compte (c) = 4
```

Ta có thể nói rằng khuôn hình hàm `compte` định nghĩa một họ các hàm `compte` trong đó kiểu của tham số đầu tiên là tuỳ ý (được xác định bởi lời gọi), còn kiểu của tham số thứ hai đã xác định (kiểu **int**).

Định nghĩa chồng các khuôn hình hàm

Giống như việc định nghĩa chồng các hàm thông thường, C++ cho phép định nghĩa chồng các khuôn hình hàm, tức là có thể định nghĩa một hay nhiều khuôn hình hàm có cùng tên nhưng với các tham số khác nhau. Điều đó sẽ tạo ra nhiều họ các hàm (mỗi khuôn hình hàm tương ứng với một họ các hàm). Ví dụ có ba họ hàm `min`:

(iv) Họ thứ nhất bao gồm các hàm tìm giá trị nhỏ nhất trong hai giá trị,

(v) Họ thứ hai tìm số nhỏ nhất trong ba số,

(vi) Họ thứ ba tìm số nhỏ nhất trong một mảng.

Ví dụ 6.6

```
/*templat7.cpp*/
#include <iostream.h>
#include <conio.h>
//khuôn hình 1
template <class T> T min(T a, T b) {
 if (a < b) return a;
   else return b;
 }
//khuôn hình 2
template <class T> T min(T a, T b, T c) {
 return min (min (a, b), c);
 }
//khuôn hình 3
template <class T> T min (T *t, int n) {
 T res = t[0];
 for(int i = 1; i < n; i++)
  if (res > t[i])
   res = t[i];
 return res;
 }
void main() {
 clrscr();
 int n = 12, p = 15, q = 2;
```

```
float x = 3.5, y = 4.25, z = 0.25;
int t[6] = {2, 3, 4,-1, 21};
char c[4] = {'w', 'q', 'a', 'Q'};
cout<<"min(n,p) = "<<min(n,p)<<"\n"; //khuôn hình 1 int min(int,int)
cout<<"min(n,p,q) = "<<min(n,p,q)<<"\n";//khuôn hình 2 int min(int,int,int)
cout<<"min(x,y) = "<<min(x,y)<<"\n"; //khuôn hình 1 float min(float,float)
cout<<"min(x,y,z) = "<<min(x,y,z)<<"\n";//khuôn hình 2 float min(float,float,float)
cout<<"min(t,6) = "<<min(t,6)<<"\n"; // khuôn hình 3 int min(int *, int)
cout<<"min(c,4) = "<<min(c,4)<<"\n"; //khuôn hình 3 char min(char *,int)
getch();
}
```

```
min(n,p) = 12
min(n,p,q) = 2
min(x,y) = 3.5
min(x,y,z) = 0.25
min(t,6) = -1
min(c,4) = Q
```

Nhận xét

Cũng giống như định nghĩa chồng các hàm, việc định nghĩa chồng các khuôn hình hàm có thể gây ra sự nhập nhằng trong việc sản sinh các hàm thể hiện. Chẳng hạn với bốn họ hàm sau đây:

```
template <class T> T fct(T, T) {...}
template <class T> T fct(T *, T) {...}
template <class T> T fct(T, T*) {...}
template <claas T> T fct(T *, T*) {...}
```

Xét các câu lệnh sau đây:

```
int x;
int y;
```

Lời gọi

```
fct(&x, &y)
```

có thể tương ứng với khuôn hình hàm 1 hay khuôn hình hàm 4.

Cụ thể hoá các hàm thể hiện

Một khuôn hình hàm định nghĩa một họ các hàm dựa trên một định nghĩa chung, nói cách khác chúng thực hiện theo cùng một giải thuật. Trong một số trường hợp, sự tổng quát này có thể chịu "rủi ro", chẳng hạn như trong trường hợp áp dụng khuôn hình hàm `min` cho kiểu **char*** như đã nói ở trên. Khái niệm cụ thể hoá, đưa ra một giải pháp khắc phục các "rủi ro" kiểu như trên. C++ cho phép ta cung cấp, ngoài định nghĩa của một khuôn hình hàm, định nghĩa của một số các hàm cho một số kiểu dữ liệu của tham số. Ta xét chương trình ví dụ sau đây:

Ví dụ 6.7

```
/*templat8.cpp*/
#include <iostream.h>
#include <string.h>
#include <conio.h>
//khuôn hình hàm min
template <class T> T min (T a, T b) {
  if (a < b) return a;
    else return b;
  }
//hàm min cho kiểu xâu ký tự
char * min (char *cha, char *chb) {
   if (strcmp(cha, chb) <0) return cha;
     else return chb;
  }
void main() {
 clrscr();
 int n= 12, p = 15;
 char *adr1= "DHBK", *adr2 ="CD2D";
 cout<<"min(n, p) = "<<min(n, p)<<"\n"; //khuôn hình hàm
 cout<<"min(adr1,adr2) = "<<min(adr1,adr2)<<endl; //hàm char * min (char *,
```

```
//char *)
 getch();
 }
```

```
min(n, p) = 12
min(adr1,adr2) = CD2D
```

Như vậy, bản chất của cụ thể hoá khuôn hình hàm là định nghĩa các hàm thông thường có cùng tên với khuôn hình hàm để giải quyết một số trường hợp rủi ro khi ta áp dụng khuôn hình hàm cho một số kiểu dữ liệu đặc biệt nào đó.

Tổng kết về các khuôn hình hàm

Một cách tổng quát, ta có thể định nghĩa một hay nhiều khuôn hình cùng tên, mỗi khuôn hình có các tham số kiểu cũng như là các tham số biểu thức riêng. Hơn nữa, có thể cung cấp các hàm thông thường với cùng tên với một khuôn hình hàm; trong trường hợp này ta nói đó là sự cụ thể hoá một hàm thể hiện.

Trong trường hợp tổng quát khi có đồng thời cả hàm định nghĩa chồng và khuôn hình hàm, chương trình dịch lựa chọn hàm tương ứng với một lời gọi hàm dựa trên các nguyên tắc sau đây:

(vii) Đầu tiên, kiểm tra tất cả các hàm thông thường cùng tên và chú ý đến sự tương ứng chính xác; nếu chỉ có một hàm phù hợp, hàm đó được chọn; còn nếu có nhiều hàm cùng thoả mãn (có sự nhập nhằng) sẽ tạo ra một lỗi biên dịch và quá trình tìm kiếm bị gián đoạn.

(viii) Nếu không có hàm thông thường nào tương ứng chính xác với lời gọi, khi đó ta kiểm tra tất cả các khuôn hình hàm có cùng tên với lời gọi; nếu chỉ có một tương ứng chính xác được tìm thấy, hàm thể hiện tương ứng được sản sinh và vấn đề được giải quyết; còn nếu có nhiều hơn một khuôn hình hàm(có sự nhập nhằng) điều đó sẽ gây ra lỗi biên dịch và quá trình tìm kiếm bị ngắt.

(ix) Cuối cùng, nếu không có khuôn hình hàm phù hợp, ta kiểm tra một lần nữa tất cả các hàm thông thường cùng tên với lời goi. Trong trường hợp này chúng ta phải tìm kiếm sự tương ứng dựa vào cả các chuyển kiểu cho phép trong C/C++.

KHUÔN HÌNH LỚP

Khuôn hình lớp là gì?

Bên cạnh khái niệm khuôn hình hàm, C++ còn cho phép định nghĩa khuôn hình lớp. Cũng giống như khuôn hình hàm, ở đây ta chỉ cần viết định nghĩa các khuôn hình lớp một lần rồi sau đó có thể áp dụng chúng với các kiểu dữ liệu khác nhau để được các lớp thể hiện khác nhau.

Tạo một khuôn hình lớp

Ta thường tạo ra lớp `point` theo kiểu (ở đây ta bỏ qua định nghĩa của các hàm thành phần):

```
class point {
  int x, y;
 public:
  point (int abs =0, int ord =0);
  void display();
  //...
  };
```

Trong ví dụ này, ta định nghĩa một lớp các điểm có toạ độ nguyên. Nếu muốn toạ độ điểm có kiểu dữ liệu khác (**float, double, long, unsigned int**) ta phải định nghĩa một lớp khác bằng cách thay thế, trong định nghĩa lớp `point`, từ khoá **int** bằng từ khoá tương ứng với kiểu dữ liệu mong muốn.

Để tránh sự trùng lặp trong các tình huống như trên, chương trình dịch C++ cho phép định nghĩa một khuôn hình lớp và sau đó, áp dụng khuôn hình lớp này với các kiểu dữ liệu khác nhau để thu được các lớp thể hiện như mong muốn:

```
template <class T> class point {
  T x; T y;
 public:
  point (T abs=0, T ord=0);
  void display();
 };
```

Cũng giống như các khuôn hình hàm, tập hợp `template<class T>` xác định rằng đó là một khuôn hình trong đó có một tham số kiểu `T`; Cũng cần phải nhắc lại rằng, C++ sử dụng từ khoá **class** chỉ để nói rằng `T` đại diện cho một kiểu dữ liệu nào đó. Tiếp theo đây ta bàn đến việc định nghĩa các hàm thành phần của khuôn hình lớp. Người ta phân biệt hai trường hợp: (i) Khi hàm thành phần được định nghĩa bên trong định nghĩa lớp trường hợp này không có gì thay đổi. Xét định nghĩa hàm thiết lập sau đây:

```
template <class T> class point {
  T x; T y;
 public:
  point(T abs=0, T ord=0) {
```

```
    x = abs; y = ord;
    }
  ...
};
```

(ii) Ngược lại, khi định nghĩa của hàm thành phần nằm ngoài định nghĩa lớp, khi đó cần phải "nhắc lại" cho chương trình dịch biết: các tham số kiểu của khuôn hình lớp, có nghĩa là phải nhắc lại: `template <class T>` trước định nghĩa hàm, còn tên của khuôn hình lớp được viết như là `point<T>`

Tóm lại, dòng tiêu đề đầy đủ cho hàm thành phần `display()` của khuôn hình hàm `point` như sau:

```
template <class T> void point<T>::display()
```

Sau đây là định nghĩa đầy đủ của khuôn hình lớp `point`:

```
#include <iostream.h>
//tạo khuôn hình hàm
template <class T> class point { T x, y;
 public:
  // định nghĩa hàm thành phần ở bên trong khuôn hình lớp
  point(T abs = 0, T ord = 0) {
    x = abs; y = ord;
    }
  void display();
 };
// định nghĩa hàm thành phần ở bên ngoài khuôn hình lớp
template <class T> void point<T>::display() {
 cout<<"Toa do: "<<x<<" "<<y<<"\n";
 }
```

Sử dụng khuôn hình lớp

Một khi khuôn hình lớp `point` đã được định nghĩa, một khai báo như :

```
point<int> ai;
```

khai báo một đối tượng `ai` có hai thành phần toạ độ là kiểu nguyên (**int**) . Điều đó có nghĩa là `point<int>` có vai trò như một kiểu dữ liệu lớp; người ta gọi nó là một lớp thể hiện của khuôn hình lớp `point`. Một cách tổng quát, khi áp dụng một kiểu dữ liệu nào đó với khuôn hình lớp `point` ta sẽ có được một lớp thể hiện tương ứng với kiểu dữ liệu. Như vậy:

```
point<double> ad;
```

định nghĩa một đối tượng `ad` có các toạ độ là số thực; còn với `point<double>` đóng vai trò một lớp và được gọi là một lớp thể hiện của khuôn hình lớp `point`.

Trong trường hợp cần phải truyền các tham số cho các hàm thiết lập, ta làm bình thường. Ví dụ:

```
point<int> ai(3,5);
point<double> ad(2.5,4.4);
```

Ví dụ sử dụng khuôn hình lớp

Ta xét ví dụ sau:

Ví dụ 6.8

```
/*templat9.cpp*/
#include <iostream.h>
#include <conio.h>
//tạo một khuôn hình lớp
template <class T> class point {
  T x, y;
 public:
  point(T abs = 0, T ord = 0) {
    x = abs; y = ord;
    }
  void display() {
    cout<<"Toa do: "<<x<<" "<<y<<"\n";
    }
 };
void main() {
 clrscr();
```

```
point<int> ai(3,5); ai.display();
point<char> ac('d','y'); ac.display();
point<double> ad(3.5, 2.3); ad.display();
getch();
}
```

```
Toa do: 3 5
Toa do: d y
Toa do: 3.5 2.3
```

Các tham số trong khuôn hình lớp

Hoàn toàn giống như khuôn hình hàm, các khuôn hình lớp có thể có các tham số kiểu và tham số biểu thức. Trong phần này ta bàn về các tham số kiểu; còn các tham số biểu thức sẽ được nói trong phần sau. Tuy có nhiều điểm giống nhau giữa khuôn hình hàm và khuôn hình lớp, nhưng các ràng buộc đối với các kiểu tham số lại không như nhau.

Số lượng các tham số kiểu trong một khuôn hình lớp

Xét ví dụ khai báo sau:

```
template <class T, class U, class V> //danh sách ba tham số kiểu
class try {
 T x;
 U t[5];
 ...
 V fm1 (int, U);
 ...
 };
```

Sản sinh một lớp thể hiện

Một lớp thể hiện được khai báo bằng cách liệt kê đằng sau tên khuôn hình lớp các tham số thực (là tên các kiểu dữ liệu) với số lượng bằng với số các tham số trong danh sách (`template<...>`) của khuôn hình lớp try:

```
try <int, float, int> // lớp thể hiện với ba tham số int, float, int
try <int,int *, double>// lớp thể hiện với ba tham số int, int *, double
```

```
try <char *, int, obj> // lớp thể hiện với ba tham số char *, int, obj
```

Trong dòng cuối ta cuối giả định `obj` là một kiểu dữ liệu đã được định nghĩa trước đó. Thậm chí có thể sử dụng các lớp thể hiện để làm tham số thực cho các lớp thể hiện khác, chẳng hạn:

```
try <float, point<int>, double>
try <point<int>,point<float>, char *>
```

Cần chú ý rằng, vấn đề tương ứng chính xác được nói tới trong các khuôn hình hàm không còn hiệu lực với các khuôn hình lớp. Với các khuôn hình hàm, việc sản sinh một thể hiện không chỉ dựa vào danh sách các tham số có trong `template<...>` mà còn dựa vào danh sách các tham số hình thức trong tiêu đề của hàm.

Một tham số hình thức của một khuôn hình hàm có thể có kiểu, là một lớp thể hiện nào đó, chẳng hạn:

```
template <class T> void fct(point<T>) { ... }
```

Việc khởi tạo mới các kiểu dữ liệu mới vẫn áp dụng được trong các khuôn hình lớp. Một khuôn hình lớp có thể có các thành phần(dữ liệu hoặc hàm) **static**. Trong trường hợp này, cần phải biết rằng, mỗi thể hiện của lớp có một tập hợp các thành phần **static** của riêng mình:

Các tham số biểu thức trong khuôn hình lớp

Một khuôn hình lớp có thể chứa các tham số biểu thức. So với khuôn hình hàm, khái niệm tham số biểu thức trong khuôn hình lớp có một số điểm khác biệt: tham số thực tế tương ứng với tham số biểu thức phải là một hằng số.

Giả sử rằng ta muốn định nghĩa một lớp `table` để thao tác trên các bảng chứa các đối tượng có kiểu bất kỳ. Một cách tự nhiên ta nghĩ ngay đến việc tạo một khuôn hình lớp với một tham số kiểu. Đồng thời còn có thể dùng một tham số thứ hai để xác định số thành phần của mảng. Trong trường hợp này, định nghĩa của khuôn hình lớp có dạng như sau:

```
template <class T, int n> class table {
  T tab[n];
public:
  ...
};
```

Danh sách các tham số (`template<...>`) chứa hai tham số với đặc điểm khác nhau hoàn toàn: một tham số kiểu được xác đinh bởi từ khoá **class**, một tham số

biểu thức kiểu int. Chúng ta sẽ phải chỉ rõ giá trị của chúng trong khai báo các lớp thể hiện. Chẳng hạn, lớp thể hiện:

```
table <int ,4>
```

tương ứng với khai báo như sau:

```
class table<int,4> {
  int tab[4];
 public:
  ...
 };
```

Sau đây là một ví dụ hoàn chỉnh:

Ví dụ 6.9

```
/*templat10.cpp*/
#include <iostream.h>
#include <conio.h>
template <class T, int n> class table {
  T tab[n];
 public:
  table() { cout<<"Tao bang\n";}
  T & operator[](int i)
     { return tab[i];
       }
 };
class point {
  int x, y;
 public:
  point (int abs = 1, int ord = 1) {
   x = abs; y = ord;
    cout<<"Tao diem "<<x<<" "<<y<<"\n";
    }
  void display() {
    cout<<"Toa do: "<<x<<" "<<y<<"\n";
```

```
    }
  };
void main() {
 clrscr();
 table<int, 4> ti;
 for(int i = 0; i < 4; i++)
  ti[i] = i;
 cout<<"ti: ";
 for(i = 0; i < 4; i++)
  cout <<ti[i]<<" ";
 cout<<"\n";
 table <point, 3> tp;
 for(i = 0; i < 3; i++)
  tp[i].display();
 getch();
 }
```

```
Tao bang
ti: 0 1 2 3
Tao diem 1 1
Tao diem 1 1
Tao diem 1 1
Tao bang
Toa do: 1 1
Toa do: 1 1
Toa do: 1 1
```

Tổng quát về khuôn hình lớp

Ta có thể khai báo một số tuỳ ý các tham số biểu thức trong danh sách các tham số của khuôn hình hàm. Các tham số này có thể xuất hiện ở bất kỳ nơi nào trong định nghĩa của khuôn hình lớp. Khi sản sinh một lớp có các tham số biểu thức, các tham số thực tế tương ứng phải là các biểu thức hằng phù hợp với kiểu dữ liệu đã khai báo trong danh sách các tham số hình thức của khuôn hình lớp.

Cụ thể hoá khuôn hình lớp

Khả năng cụ thể hoá khuôn hình lớp có đôi chút khác biệt so với khuôn hình hàm.

Khuôn hình lớp định nghĩa họ các lớp trong đó mỗi lớp chứa đồng thời định nghĩa của chính nó và các hàm thành phần. Như vậy, tất cả các hàm thành phần cùng tên sẽ được thực hiện theo cùng một giải thuật. Nếu ta muốn cho một hàm thành phần thích ứng với một tình huống cụ thể cụ thể nào đó, có thể viết một định nghĩa khác cho nó. Sau đây là một ví dụ cải tiến khuôn hình lớp `point`. Ở đây chúng ta đã cụ thể hoá hàm hiển thị `display()` cho trường hợp kiểu dữ liệu **char**:

Ví dụ 6.10

```
/*templat11.cpp*/
#include <iostream.h>
#include <conio.h>
//tạo một khuôn hình lớp
template <class T> class point {
  T x, y;
 public:
  point(T abs = 0, T ord = 0) {
   x = abs; y = ord;
   }
  void display();
 };
template <class T> void point<T>::display() {
 cout<<"Toa do: "<<x<<" "<<y<<"\n";
 }
//Thêm một hàm display cụ thể hoá trong trường hợp các ký tự
void point<char>::display() {
 cout<<"Toa do: "<<(int)x<<" "<<(int)y<<"\n";
 }
void main() {
 clrscr();
 point <int> ai(3,5);
 ai.display();
```

```
point <char> ac('d','y');
ac.display();
point <double> ad(3.5, 2.3);
ad.display();
getch();
}
```

```
Toa do: 3 5
Toa do: 100 121
Toa do: 3.5 2.3
```

Ta chú ý dòng tiêu đề trong khai báo một thành phần được cụ thể hoá:

```
void point<char>::display()
```

Khai báo này nhắc chương trình dịch sử dùng hàm này thay thế hàm `display()` của khuôn hình lớp `point` (trong trường hợp giá trị thực tế cho tham số kiểu là **char**).

Nhận xét

(x) Có thể cụ thể hoá giá trị của tất cả các tham số. Xét khuôn hình lớp sau đây:

```
template <class T, int n> class table {
  T tab[n];
public:
  table() {cout<<" Tao bang\n"; }
  ...
};
```

Khi đó, chúng ta có thể viết một định nghĩa cụ thể hoá cho hàm thiết lập cho các bảng 10 phần tử kiểu `point` như sau:

```
table<point,10>::table(...) {...}
```

(xi) Có thể cụ thể hoá một hàm thành phần hay một lớp. Trong ví dụ 6.10 chương trình `template11.cpp` đã cụ thể hoá một hàm thành phần của khuôn hình lớp. Nói chung có thể: cụ thể hoá một hay nhiều hàm thành phần, hoặc không cần thay đổi định nghĩa của bản thân lớp (thực tế cần phải làm như vậy) mà cụ thể hoá bản thân lớp bằng cách đưa thêm định nghĩa. Khả năng thứ hai này có dẫn tới việc phải cụ thể hoá một số hàm thành phần. Chẳng hạn, sau khi định nghĩa khuôn hình `template<class T> class point`, ta có thể định

nghĩa một phiên bản cụ thể cho kiểu dữ liệu `point` thích hợp với thể hiện `point<char>`. Ta làm như sau:

```
class point<char> {
//định nghĩa mới  };
```

Sự giống nhau của các lớp thể hiện

Xét câu lệnh gán giữa hai đối tượng. Như chúng ta đã biết, chỉ có thể thực hiện được phép gán khi hai đối tượng có cùng kiểu (với trường hợp thừa kế vấn đề đặc biệt hơn một chút nhưng thực chất chỉ là chuyển kiểu ngầm định đối với biểu thức vế phải của lệnh gán). Trước đây, ta biết rằng hai đối tượng có cùng kiểu nếu chúng được khai báo với cùng một tên lớp. Trong trường hợp khuôn hình lớp, nên hiểu sự cùng kiểu ở đây như thế nào? Thực tế, hai lớp thể hiện tương ứng với cùng một kiểu nếu các tham số kiểu tương ứng nhau một cách chính xác và các tham số biểu thức có cùng giá trị. Như vậy (giả thiết rằng chúng ta đã định nghĩa khuôn hình lớp `table` trong mục 2.6) với các khai báo:

```
table <int, 12> t1;
table <float,12> t2;
```

ta không có quyền viết:

`t2 = t1;` *//không tương thích giữa hai tham số đầu*

Cũng vậy, với các khai báo:

```
table <int, 12> ta;
table <int, 20> tb;
```

cũng không được quyền viết

`ta = tb;`*//giá trị thực của hai tham số sau khác nhau.*

Những qui tắc chặt chẽ trên nhằm làm cho phép gán ngầm định được thực hiện chính xác. Trường hợp có các định nghĩa chồng toán tử gán, có thể không nhất thiết phải tuân theo qui tắc đã nêu trên. Chẳng hạn hoàn toàn có thể thực hiện được phép gán

```
t2 = t1;
```

nếu ta có định nghĩa hai lớp thể hiện `table<int, m>` và `table<float,n>` và trong lớp thứ hai có định nghĩa chồng phép gán bằng.

Các lớp thể hiện và các khai báo bạn bè

Các khuôn hình lớp cũng cho phép khai báo bạn bè. Bên trong một khuôn hình lớp, ta có thể thực hiện ba kiểu khai báo bạn bè như sau.

Khai báo các lớp bạn hoặc các hàm bạn thông thường

Giả sử `A` là một lớp thông thường và `fct()` là một hàm thông thường. Xét khai báo sau đây trong đó khai báo `A` là lớp bạn và `fct()` là hàm bạn của tất cả các lớp thể hiện của khuôn hình lớp:

```
template <class T> class try {
int x; public:
friend class A;
friend int fct(float);
...
};
```

Khai báo bạn bè của một thể hiện của khuôn hình hàm, khuôn hình lớp

Xét hai ví dụ khai báo sau đây. Giả sử chúng ta có khuôn hình lớp và khuôn hình hàm sau:

```
template <class T> class point {...};
template <class T> int fct (T) {...};
```

Ta định nghĩa hai khuôn hình lớp như sau:

```
template <class T, class U> class try1 {
int x; public:
friend class point<int>;
friend int fct(double);
...
};
```

Khai báo này xác định hai thể hiện rất cụ thể của khuôn hình hàm `fct` và khuôn hình lớp `point` là bạn của khuôn hình lớp `try1`.

```
template <class T, class U> class try2 {
int x; public:
friend class point<T>;
friend int fct(U);
...
};
```

So với `try1`, trong `try2` người ta không xác định rõ các thể hiện của `fct()` và `point` là bạn của một thể hiện của `try2`. Các thể hiện này sẽ được cụ thể tại thời điểm chúng ta tạo ra một lớp thể hiện của `try2`. Ví dụ, với lớp thể hiện `try2<double, long>` ta có lớp thể hiện bạn là `point<double>` và hàm thể hiện bạn là `fct<long>`

Khai báo bạn bè của khuôn hình hàm, khuôn hình lớp

Xét ví dụ sau đây:

```
template <class T, class U> class try3 {
int x; public:
template <class X> friend class point<X>;
template <class X> friend int fct(X);
...
};
```

Lần này, tất cả các thể hiện của khuôn hình lớp `point` đều là bạn của các thể hiện nào của khuôn hình lớp `try3`. Tương tự như vậy tất cả các thể hiện của khuôn hình hàm `fct()` đều là bạn của các thể hiện của khuôn hình lớp `try3`.

Ví dụ về lớp bảng có hai chỉ số

Trước đây, để định nghĩa một lớp table có hai chỉ số ta phải sử dụng một lớp vector trung gian. Ở đây, ta xét một cách làm khác nhờ sử dụng khuôn hình lớp.

Với định nghĩa sau:

```
template <class T, int n> class table {
  T tab[n];
 public:
  T &operator [](int i) {
    return tab[i];
    }
};
```

ta có thể khai báo:

```
table <table<int,2>,3> t2d;
```

Trong trường hợp này, thực chất ta có một bảng ba phần tử, mỗi phần tử có kiểu `table<int,2>`. Nói cách khác, mỗi phần tử lại là một bảng chứa hai số

nguyên. Ký hiệu `t2d[1][2]` biểu thị một tham chiếu đến thành phần thứ ba của `t2d[1]`, bảng `t2d[1]` là phần tử thứ hai của bảng hai chiều các số nguyên `t2d`. Sau đây là một chương trình hoàn chỉnh:

Ví dụ 6.10

```
#include <iostream.h>
#include <conio.h>
template <class T, int n> class table {
  T tab[n];
 public:
  table() //hàm thiết lập
   {
   cout<<"Tao bang co "<<n<<"phan tu\n";
   }
  T & operator[] (int i) //hàm toán tử []
   {
   return tab[i];
   }
 };
void main() {
 clrscr();
 table <table<int,2>,3> t2d;
 t2d[1][2] = 15;
 cout <<"t2d [1] [2] ="<<t2d[1][2]<<"\n";
 int i, j;
 for (i = 0; i < 2; i++)
  for(j = 0; j < 3; j++)
   t2d[i][j] = i*3+j;
 for(i =0; i < 2; i++) {
  for(j = 0; j < 3; j++)
   cout <<t2d[i][j]<<" ";
  cout<<"\n";
  }
```

```
}
```

```
Tao bang co 2 phan tu
Tao bang co 2 phan tu
Tao bang co 2 phan tu
Tao bang co 3 phan tu
t2d [1] [2] = 15
0 1 2 3 4 5 6 7 8
```

Chú ý

Chương trình ví dụ trên đây còn có một số điểm yếu: chưa quản lý vấn đề tràn chỉ số bản, không khởi tạo giá trị cho các thành phần của bảng. Để giải quyết cần có một hàm thiết lập với một tham số có kiểu **int**, và một hàm thiết lập có nhiệm vụ chuyển kiểu int thành kiểu T bất kỳ. Các yêu cầu này coi như là bài tập.

Ví dụ 6.11

```
/*templa13.cpp*/
#include <iostream.h>
#include <conio.h>
template <class T, int n> class table {
  T tab[n];
  int limit;
 public:
  table (int init = 0);
   T & operator[] (int i) {
       if (i < 0 || i >limit)
         cout<<"Tran chi so "<<i<<"\n";
       else return tab[i];
    }
};
template <class T, int n> table<T,n>::table(int init = 0) {
  int i;
  for (i = 0; i < n; i++) tab[i] = init;
       limit = n - 1;
  cout<<"Tao bang kich thuoc "<<n<<" init = "<<init<<"\n";
```

```
}
void main() {
 clrscr();
 table <table<int,3>,2>ti; //khởi tạo ngầm định
 table <table<float,4>,2> td(10); //khởi tao bằng 10
 ti[1][6] = 15;
 ti[8][-1] = 20;
 cout<<ti[1][2]<<"\n"; cout<<td[1][0]<<"\n";
 getch();
 }
```

```
Tao bang kich thuoc 3 init = 0
Tao bang kich thuoc 3 init = 0
Tao bang kich thuoc 3 init = 0
Tao bang kich thuoc 3 init = 0
Tao bang kich thuoc 2 init = 0
Tao bang kich thuoc 4 init = 0
Tao bang kich thuoc 4 init = 0
Tao bang kich thuoc 4 init = 10
Tao bang kich thuoc 4 init = 10
Tao bang kich thuoc 2 init = 10
Tran chi so 6
Tran chi so 8
Tran chi so -1 0 10
```

Chú ý

Nếu để ý các thông báo tạo các bảng ta thấy rằng ta thu được nhiều hơn hai lần so với dự kiến với các bảng có hai chỉ số. Lời giải thích nằm ở trong câu lệnh gán tb[i]=ti của hàm thiết lập table. Khi khai báo mảng các đối tượng T chương trình dịch gọi tới các hàm thiết lập ngầm định với các tham số bằng 0. Để thực hiện lệnh gán tb[i]=ti chương trình dịch sẽ thực hiện chuyển đổi từ kiểu số nguyên sang một đối tượng tạm thời kiểu T. Điều này cũng sẽ gọi tới hàm thiết lập `table(int)`. Chẳng hạn trong khai báo của bảng `td`, ta thấy, chương trình dịch tạo ra hai bảng một chiều các số thực với kích thước là 4 và `init =0`, đồng thời cũng có hai lần tạo các bảng trung gian với kích thước bằng 4 và `init=10`.

TÓM TẮT

Ghi nhớ

Khuôn hình lớp/hàm là phương tiện mô tả ý nghĩa của một lớp/hàm tổng quát còn lớp/hàm thể hiện là một bản sao của khuôn hình tổng quát với các kiểu dữ liệu cụ thể.

Các khuôn hình lớp/hàm thường được tham số hoá, tuy nhiên vẫn có thể sử dụng các kiểu cụ thể trong các khuôn hình lớp/hàm nếu cần.

BÀI TẬP

Bài 6.1. Viết chương trình khai báo khuôn hình lớp để mô phỏng hoạt động của hàng đợi hoặc ngăn xếp trên các kiểu đối tượng khác nhau.

1. Khuôn hình hàm.....233
1.1 Khuôn hình hàm là gì?.....233
1.2 Tạo một khuôn hình hàm.....233
1.3 Sử dụng khuôn hình hàm.....234
1.3.1.....Khuôn hình hàm cho kiểu dữ liệu cơ sở 234
1.3.2..... Khuôn hình hàm min cho kiểu char * 235
1.3.3.....Khuôn hình hàm min với kiểu dữ liệu lớp 236
1.4 Các tham số kiểu của khuôn hình hàm.....237
1.4.1.....Các tham số kiểu trong định nghĩa khuôn hình hàm 237
1.5 Giải thuật sản sinh một hàm thể hiện.....240
1.6 Khởi tạo các biến có kiểu dữ liệu chuẩn..... 241
1.7 Các hạn chế của khuôn hình hàm.....241
1.8 Các tham số biểu thức của một khuôn hình hàm.....242
1.9 Định nghĩa chồng các khuôn hình hàm.....244
1.10 Cụ thể hoá các hàm thể hiện.....246
1.11 Tổng kết về các khuôn hình hàm.....247
2. KHUôN hình lớp.....247
2.1 Khuôn hình lớp là gì?.....247
2.2 Tạo một khuôn hình lớp.....248
2.3 Sử dụng khuôn hình lớp.....249
2.4 Ví dụ sử dụng khuôn hình lớp.....250
2.5 Các tham số trong khuôn hình lớp.....251

2.5.1..... Số lượng các tham số kiểu trong một khuôn hình lớp 251

2.5.2..... Sản sinh một lớp thể hiện 251

2.6 Các tham số biểu thức trong khuôn hình lớp.....252

2.7 Tổng quát về khuôn hình lớp.....254

2.8 Cụ thể hoá khuôn hình lớp..... 255

2.9 Sự giống nhau của các lớp thể hiện..... 257

2.10 Các lớp thể hiện và các khai báo bạn bè.....258

2.10.1 Khai báo các lớp bạn hoặc các hàm bạn thông thường. 258

2.10.2 Khai báo bạn bè của một thể hiện của khuôn hình hàm, khuôn hình lớp.....258

2.10.3 Khai báo bạn bè của khuôn hình hàm, khuôn hình lớp. 259

2.11 Ví dụ về lớp bảng có hai chỉ số..... 259

3. Tóm tắt..... 263

3.1 Ghi nhớ.....263

4. Bài tập.....263