CHƯƠNG 5

KỸ THUẬT THỪA KẾ
(INHERITANCE)

Mục đích chương này:

1. Cài đặt sự thừa kế.
2. Sử dụng các thành phần của lớp cơ sở.
3. Định nghĩa lại các hàm thành phần.
4. Truyền thông tin giữa các hàm thiết lập của lớp dẫn xuất và lớp cơ sở.
5. Các loại dẫn xuất khác nhau và sự thay đổi trạng thái của các thành phần lớp cơ sở.
6. Sự tương thích giữa các đối tượng của lớp dẫn xuất và lớp cơ sở.
7. Toán tử gán và thừa kế.
8. Hàm ảo và tính đa hình
9. Hàm ảo thuần tuý và lớp cơ sở trừu tượng
10. Đa thừa kế và các vấn đề liên quan.

GIỚI THIỆU CHUNG

Thừa kế là một trong bốn nguyên tắc cơ sở của phương pháp lập trình hướng đối tượng. Đặc biệt đây là cơ sở cho việc nâng cao khả năng sử dụng lại các bộ phận của chương trình. Thừa kế cho phép ta định nghĩa một lớp mới, gọi là lớp dẫn xuất, từ một lớp đã có, gọi là lớp cơ sở. Lớp dẫn xuất sẽ thừa kế các thành phần (dữ liệu, hàm) của lớp cơ sở, đồng thời thêm vào các thành phần mới, bao hàm cả việc làm "tốt hơn" hoặc làm lại những công việc mà trong lớp cơ sở chưa làm tốt hoặc không còn phù hợp với lớp dẫn xuất. Chẳng hạn có thể định nghĩa lớp "mặt hàng nhập khẩu" dựa trên lớp "mặt hàng", bằng cách bổ sung thêm thuộc tính "thuế". Khi đó cách tính chênh lệch giá bán, mua cũ trong lớp "mặt hàng" sẽ không phù hợp nữa nên cần phải sửa lại cho phù hợp. Lớp điểm có màu được định nghĩa dựa trên lớp điểm không màu bằng cách bổ sung thêm thuộc tính màu, hàm `display()` lúc này ngoài việc hiển thị hai thành phần toạ độ còn phải cho biết màu của đối tượng điểm. Trong cả hai ví dụ đưa ra, trong lớp dẫn xuất đều có sự bổ sung và thay đổi thích hợp với tình hình mới.

Thừa kế cho phép không cần phải biên dịch lại các thành phần chương trình vốn đã có trong các lớp cơ sở và hơn thế nữa không cần phải có chương trình nguồn tương ứng. Kỹ thuật này cho phép chúng ta phát triển các công cụ mới dựa trên

những gì đã có được. Người sử dụng Borland C hay Turbo Pascal 6.0/7.0 rất thích sử dụng Turbo Vision - một thư viện cung cấp các lớp, đối tượng là cơ sở để xây dựng các giao diện ứng dụng hết sức thân thiện đối với người sử dụng. Tất cả các lớp này đều được cung cấp dưới dạng các tập tin *.obj, *.lib nghĩa là người sử dụng hoàn toàn không thể và không cần phải biết rõ phần chương trình nguồn tương ứng. Tuy nhiên điều đó không quan trọng khi người lập trình được phép thừa kế các lớp định nghĩa trước đó.

Thừa kế cũng cho phép nhiều lớp có thể dẫn xuất từ cùng một lớp cơ sở, nhưng không chỉ giới hạn ở một mức: một lớp dẫn xuất có thể là lớp cơ sở cho các lớp dẫn xuất khác. Ở đây ta thấy rằng khái niệm thừa kế giống như công cụ cho phép mô tả cụ thể hoá các khái niệm theo nghĩa: lớp dẫn xuất là một cụ thể hoá hơn nữa của lớp cơ sở và nếu bỏ đi các dị biệt trong các lớp dẫn xuất sẽ chỉ còn các đặc điểm chung nằm trong lớp cơ sở.

Hình 5.1 mô tả một sơ đồ thừa kế của các lớp, có cung đi từ lớp này sang lớp kia nếu chúng có quan hệ thừa kế. Ta gọi đó là đồ thị thừa kế. Sau đây là một số mô tả cho các lớp xuất hiện trong đồ thị thừa kế ở trên.

1. *Lớp* mặt hàng

các thuộc tính

tên

số lượng trong kho

giá mua

giá bán

các phương thức

hàm chênh lệch giá bán mua

{giá bán - giá mua}

thủ tục mua(q)

{Thêm vào trong kho q đơn vị mặt hàng}

thủ tục bán(q)

{Bớt đi q đơn vị mặt hàng có trong kho}

2. *Lớp* mặt hàng nhập khẩu *thừa kế từ* mặt hàng

các thuộc tính

thuế nhập khẩu

các phương thức

hàm chênh lệch giá bán -mua

{giá bán - giá mua* thuế nhập khẩu}

3. *Lớp* xe gắn máy *thừa kế từ* mặt hàng nhập khẩu

các thuộc tính

dung tích xy lanh

các phương thức

4. *Lớp* hàng điện tử dân dụng *thừa kế từ* mặt hàng

các thuộc tính

điện áp

thời hạn bảo hành

các phương thức

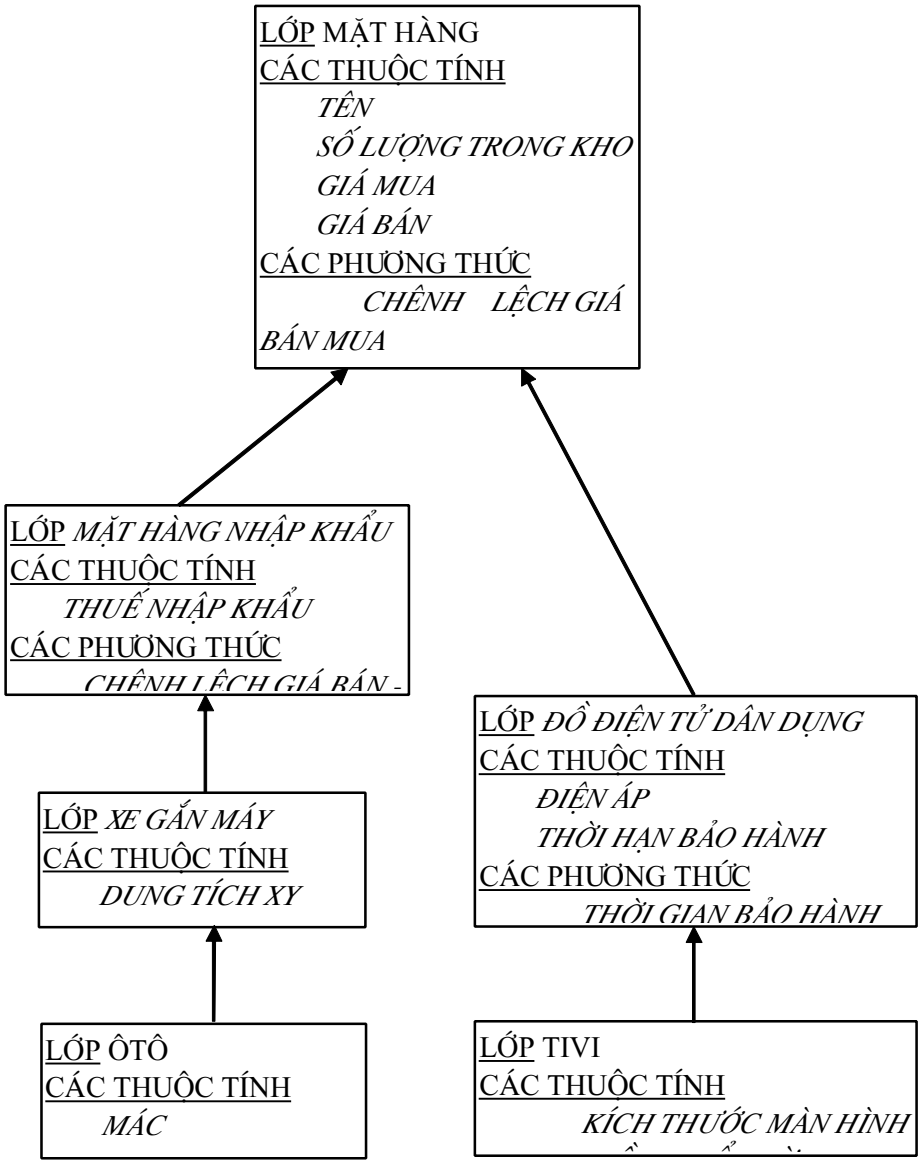
hàm thời gian bảo hành thực tế

...

Tính đa hình cũng là một trong các điểm lý thú trong lập trình hướng đối tượng, được thiết lập trên cơ sở thừa kế trong đó đối tượng có thể có biểu hiện khác nhau tuỳ thuộc vào tình huống cụ thể. Tính đa hình ấy có thể xảy ra ở một hành vi của đối tượng hay trong toàn bộ đối tượng. Ví dụ trực quan thể hiện tính đa hình là một ti vi có thể vừa là đối tượng của mặt hàng vừa là đối tượng của lớp mặt hàng điện tử dân dụng. Các đối tượng hình học như hình vuông, hình tròn, hình chữ nhật đều có cùng cách vẽ như nhau: xác định hai điểm đầu và cuối, nối hai điểm này. Do vậy thuật toán tuy giống nhau đối với tất cả các đối tượng hình, nhưng cách vẽ thì phụ thuộc vào từng lớp đối tượng cụ thể. Ta nói phương thức nối điểm của các đối tượng hình học có tính đa hình. Tính đa hình còn được thể hiện trong cách thức hiển thị thông tin trong các đối tượng điểm màu/không màu.

Các ngôn ngữ lập trình hướng đối tượng đều cho phép đa thừa kế, theo đó một lớp có thể là dẫn xuất của nhiều lớp khác. Do vậy dẫn tới khả năng một lớp cơ sở có thể được thừa kế nhiều lần trong một lớp dẫn xuất khác, ta gọi đó là sự xung đột thừa kế. Điều này hoàn toàn không hay, cần phải tránh. Từng ngôn ngữ sẽ có những giải pháp của riêng mình, C++ đưa ra khái niệm thừa kế ảo.

Trong chương này ta sẽ đề cập tới các khả năng của C++ để thể hiện nguyên tắc thừa kế khi viết chương trình.



HÌNH 5.1 VÍ DỤ VỀ SƠ ĐỒ THỪA KẾ

ĐƠN THỪA KẾ

1.1 Ví dụ minh hoạ

Chương trình `inheri1.cpp` sau đây là một ví dụ thể hiện tính thừa kế đơn của lớp `coloredpoint`, mô tả các điểm màu trên mặt phẳng, từ lớp các điểm không màu nói chung `point`. Chương trình này đề cập đến khá nhiều khía cạnh, liên quan đến kỹ thuật cài đặt tính thừa kế trong C++, đó là:

(i) Truy nhập các thành phần lớp cơ sở từ lớp dẫn xuất,

Định nghĩa lại (đè) các thành phần lớp cơ sở trong lớp dẫn xuất

Truyền thông tin giữa các hàm thiết lập

Ví dụ 5.1

```
/*inheri1.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  point() {x = 0; y = 0;}
  point(float ox, float oy) {x = ox; y = oy; }
  point(point &p) {x = p.x; y = p.y;}
  void display() {
    cout<<"Goi ham point::display() \n";
    cout<<"Toa do :"<<x<<" "<<y<<endl;
   }
  void move(float dx, float dy) {
   x += dx;
   y += dy;
   }
 };
/*lop coloredpoint thừa kế từ point*/
class coloredpoint : public point {
  unsigned int color;
 public:
```

```
 coloredpoint():point() {
  color =0;
  }
 coloredpoint(float ox, float oy, unsigned int c):point(ox,oy) {
  color = c;
  }
 coloredpoint(coloredpoint &b):point((point &)b) {
  color = b.color;
  }
 void display() {
  cout<<"Ham coloredpoint::display()\n";
  point::display();/*gọi tới hàm cùng tên trong lớp cơ sở*/
  cout<<"Mau "<<color<<endl;
  }
};
void main() {
 clrscr();
 coloredpoint m;
 cout<<"Diem m \n";
 m.display();
 cout<<"Chi hien thi toa do cua m\n";
 m.point::display();/*gọi phương thức display trong lớp point*/
 coloredpoint n(2,3,6);
 cout<<"Diem n \n";
 n.display();
 cout<<"Chi hien thi toa do cua n\n";
 n.point::display();
 coloredpoint p =n;
 cout<<"Diem p \n";
 p.display();
 cout<<"Chi hien thi toa do cua p\n";
 p.point::display();
```

```
getch();
}
```

```
Diem m
Ham coloredpoint::display()
Goi ham point::display()
Toa do :0 0
Mau 0
Chi hien thi toa do cua m
Goi ham point::display()
Toa do :0 0
Diem n
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 6
Chi hien thi toa do cua n
Goi ham point::display()
Toa do :2 3
Diem p
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 6
Chi hien thi toa do cua p
Goi ham point::display()
Toa do :2 3
```

1.2 Truy nhập các thành phần của lớp cơ sở từ lớp dẫn xuất

Các thành phần **private** trong lớp cơ sở không thể truy nhập được từ các lớp dẫn xuất. Chẳng hạn các thành phần **private** x và y trong lớp cơ sở `point` không được dùng trong định nghĩa các hàm thành phần của lớp dẫn xuất `coloredpoint`. Tức là "*Phạm vi lớp*" *chỉ mở rộng cho bạn bè, mà không được mở rộng đến các lớp con cháu.*

Trong khi đó, trong định nghĩa của các hàm thành phần trong lớp dẫn xuất được phép truy nhập đến các thành phần **protected** và **public** trong lớp dẫn xuất, chẳng hạn có thể gọi tới hàm thành phần `point::dislay()` của lớp cơ sở bên trong định nghĩa hàm thành phần `coloredpoint::display()` trong lớp dẫn xuất.

1.3 Định nghĩa lại các thành phần của lớp cơ sở trong lớp dẫn xuất

Chương trình inheri1.cpp cũng có một ví dụ minh hoạ cho điều này: hàm `display()` tuy đã có trong lớp `point`, được định nghĩa lại trong lớp `coloredpoint`. Lúc này, thực tế là có hai phiên bản khác nhau của `display()` cùng tồn tại trong lớp `coloredpoint`, một được định nghĩa trong lớp `point`, được xác định bởi `point::display()` và một được định nghĩa trong lớp `coloredpoint` và được xác định bởi `coloredpoint::display()`. Trong phạm vi của lớp dẫn xuất hàm thứ hai "che lấp" hàm thứ nhất, nghĩa là tên gọi `display()` trong định nghĩa của hàm thành phần của lớp `coloredpoint` hoặc trong lời gọi hàm thành phần `display()` từ một đối tượng lớp `coloredpoint` phải tham chiếu đến `coloredpoint::display()`. Nếu muốn gọi tới hàm `display()` của lớp `point` ta phải viết đầy đủ tên của nó, nghĩa là `point::display()`. Trong định nghĩa hàm `coloredpoint::display()` nếu ta thay thế `point::display()` bởi `display()` thì sẽ tạo ra lời gọi đệ quy vô hạn lần.

Cần chú ý rằng việc định nghĩa lại một hàm thành phần khác với định nghĩa chồng hàm thành phần, bởi vì khái niệm định nghĩa lại chỉ được xét tới khi ta nói đến sự thừa kế. Hàm định nghĩa lại và hàm bị định nghĩa lại giống hệt nhau về tên, tham số và giá trị trả về, chúng chỉ khác nhau ở vị trí, một hàm đặt trong lớp dẫn xuất và hàm kia thì có mặt trong lớp cơ sở. Trong khi đó, các hàm chồng chỉ có cùng tên, thường khác nhau về danh sách tham số và tất cả chúng thuộc về cùng một lớp. Định nghĩa lại hàm thành phần chính là cơ sở cho việc cài đặt tính đa hình của các phương thức của đối tượng.

C++ còn cho phép khai báo bên trong lớp dẫn xuất các thành phần dữ liệu cùng tên với các thành phần dữ liệu đã có trong lớp cơ sở. Hai thành phần cùng tên này có thể cùng kiểu hay khác kiểu. Lúc này bên trong một đối tượng của lớp dẫn xuất có tới hai thành phần khác nhau có cùng tên, nhưng trong phạm vi lớp dẫn xuất tên chung đó nhằm chỉ định thành phần được khai báo lại trong lớp dẫn xuất. Khi muốn chỉ định thành phần cùng tên trong lớp cơ sở, phải sử dụng tên lớp cơ sở và toán tử phạm vi "::" đặt trước tên thành phần đó.

1.4 Tính thừa kế trong lớp dẫn xuất

1.4.1 Sự tương thích của đối tượng thuộc lớp dẫn xuất với đối tượng thuộc lớp cơ sở

Một cách tổng quát, trong lập trình hướng đối tượng, một đối tượng của lớp dẫn xuất có thể "thay thế" một đối tượng của lớp cơ sở. Nghĩa là: tất cả những thành phần dữ liệu có trong lớp cơ sở đều tìm thấy trong lớp dẫn xuất; tất cả các

hành động thực hiện được trên lớp cơ sở luôn luôn có thể làm được trên các lớp dẫn xuất. Trong chương trình `inheri1.cpp` ta thấy rằng: đối tượng `m` của lớp `coloredpoint` có đồng thời hai thành phần toạ độ x , y và thêm thành phần dữ liệu bổ sung `color`; ngoài ra có thể gọi các hàm thành phần `point::display()` và `point::move(...)` thông qua đối tượng m.

Tính tương thích giữa một đối tượng của lớp dẫn xuất và đối tượng lớp cơ sở được thể hiện ở chỗ có thể chuyển kiểu ngầm định từ một đối tượng của lớp dẫn xuất sang một đối tượng của lớp cơ sở. Xét các chỉ thị sau:

```
point p;
p.display();
coloredpointcol pc(2,3,5);
```

câu lệnh

```
p=pc;
p.display();
pc.display();
```

cho kết quả

```
Diem p khong mau
Goi ham point::display()
Toa do :0 0
Diem pc co mau
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
p =pc
Goi ham point::display()
Toa do :2 3
```

Tuy nhiên nhận xét trên đây không hoàn toàn đúng xét theo chiều ngược lại. Câu lệnh sau đây không đúng nếu không có định nghĩa chồng toán tử gán gữa hai đối tượng với các kiểu dữ liệu `coloredpoint` và `point`.

```
pc=p;
```

Chú ý

Từ `pc` chỉ có thể gọi đến các thành phần hàm **public** trong lớp `point` (xem thêm phần sau để hiểu rõ hơn).

1.4.2 Tương thích giữa con trỏ lớp dẫn xuất và con trỏ lớp cơ sở

Tương tự như vấn đề trình bày trong phần trênmột con trỏ đối tượng lớp cơ sở có thể chỉ đến một đối tượng lớp dẫn xuất, còn một con trỏ lớp dẫn xuất không thể nhận địa chỉ của đối tượng lớp cơ sở, trừ trường hợp ép kiểu. Ta xét chương trình ví dụ sau:

Ví dụ 5.2

```
/*inheri2.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  point() {x = 0; y = 0;}
  point(float ox, float oy) {x = ox; y = oy; }
  point(point &p) {x = p.x; y = p.y;}
  void display() {
   cout<<"Goi ham point::display() \n";
   cout<<"Toa do :"<<x<<" "<<y<<endl;
   }
 void move(float dx, float dy) {
  x += dx;
  y += dy;
  }
 };
/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
  unsigned int color;
 public:
  coloredpoint():point() {
```

```
  color =0;
  }
 coloredpoint(float ox, float oy, unsigned int c):point(ox,oy) {
  color = c;
  }
 coloredpoint(coloredpoint &b):point((point &)b) {
  color = b.color;
  }
void display() {
  cout<<"Ham coloredpoint::display()\n";
  point::display();/*gọi tới hàm cùng tên trong lớp cơ sở*/
  cout<<"Mau "<<color<<endl;
  }
};
void main() {
clrscr();
point *adp;
coloredpoint pc(2,3,5);
pc.display();
cout<<"adp = &pc \n";
adp=&pc;
adp->move(2,3);
cout<<"adp->move(2,3)\n";
pc.display();
adp->display();
getch();
}
```

```
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
```

```
adp = &pc
adp->move(2,3)
Ham coloredpoint::display()
Goi ham point::display()
Toa do :4 6
Mau 5
Goi ham point::display()
Toa do :4 6
```

Nhận xét

Chú ý kết quả thực hiện chỉ thị :

```
adp->display();
```

```
Goi ham point::display()
Toa do :4 6
```

Như vậy, mặc dù `adp` chứa địa chỉ của đối tượng `coloredpoint` là `pc`, nhưng

`apd->display()`

vẫn là

`point::display()`

chứ không phải `coloredpoint::display()`.

Hiện tượng này được giải thích là do `adp` được khai báo là con trỏ kiểu `point` và vì các hàm trong `point` đều được khai báo như bình thường nên `adp` chỉ có thể gọi được các hàm thành phần có trong `point` chứ không phụ thuộc vào đối tượng mà `adp` chứa địa chỉ. Muốn có được tính đa hình cho `display()` nghĩa là lời gọi tới `display()` phụ thuộc vào kiểu đối tượng có địa chỉ chứa trong `adp`, ta phải khai báo hàm thành phần `display()` trong `point` như là một hàm ảo. Phần sau chúng ta sẽ đề cập vấn đề này,

1.4.3 Tương thích giữa tham chiếu lớp dẫn xuất và tham chiếu lớp cơ sở

Vấn đề được xét trong phần 2.4.2 cũng hoàn toàn tương tự đối với tham chiếu. Ta xét chương trình sau:

Ví dụ 5.3

```
/*inheri3.cpp*/
```

```
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  point() {x = 0; y = 0;}
  point(float ox, float oy) {x = ox; y = oy; }
  point(point &p) {x = p.x; y = p.y;}
  void display() {
    cout<<"Goi ham point::display() \n";
    cout<<"Toa do :"<<x<<" "<<y<<endl;
   }
  void move(float dx, float dy) {
   x += dx;
   y += dy;
   }
 };
/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
  unsigned int color;
 public:
  coloredpoint():point() {
   color =0;
   }
  coloredpoint(float ox, float oy, unsigned int c):point(ox,oy) {
   color = c;
   }
  coloredpoint(coloredpoint &b):point((point &)b) {
   color = b.color;
   }
 void display() {
  cout<<"Ham coloredpoint::display()\n";
```

```
  point::display();
  cout<<"Mau "<<color<<endl;
  }
};
void main() {
clrscr();
coloredpoint pc(2,3,5);
pc.display();
cout<<"point &rp = pc \n";
point &rp=pc;
rp.move(2,3);
cout<<"rp.move(2,3)\n";
pc.display();
rp.display();
getch();
}
```

```
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
point &rp = pc
rp.move(2,3)
Ham coloredpoint::display()
Goi ham point::display()
Toa do :4 6
Mau 5
Goi ham point::display()
Toa do :4 6
```

1.5 Hàm thiết lập trong lớp dẫn xuất

1.5.1 Hàm thiết lập trong lớp

Nếu lớp có khai báo tường minh ít nhất một hàm thiết lập thì khi tạo ra một đối tượng sẽ có một lời gọi đến một trong các hàm thiết lập được định nghĩa. Việc chọn lựa hàm thiết lập dựa theo các tham số được cung cấp kèm theo. Trường hợp không có hàm thiết lập nào phù hợp sẽ sinh ra một lỗi biên dịch. Như vậy không thể tạo ra một đối tượng nếu không dùng đến một trong các hàm thiết lập đã được định nghĩa.

Trong trường hợp thật sự không có hàm thiết lập tường minh ta không thể mô tả tường tận các dữ liệu của đối tượng liên quan. Xét chương trình sau:

Ví dụ 5.4

```
#include <iostream.h>
#include <conio.h>
/*khai báo lớp point mà không có hàm thiết lập tường minh*/
class point {
  float x,y;
 public:
  void display() {
   cout<<"Goi ham point::display() \n";
   cout<<"Toa do :"<<x<<" "<<y<<endl;
   }
 void move(float dx, float dy) {
  x += dx;
  y += dy;
  }
};
void main() {
 clrscr();
 cout<<"Diem p \n";
 point p;
 p.display();
 getch();
}
```

```
Diem p
Goi ham point::display()
Toa do :8.187236e-34 2.637535e-11
```

1.5.2 Phân cấp lời gọi

Một đối tượng lớp dẫn xuất về thực chất có thể coi là một đối tượng của lớp cơ sở, vì vậy việc gọi hàm thiết lập lớp dẫn xuất để tạo đối tượng lớp dẫn xuất sẽ kéo theo việc gọi đến một hàm thiết lập trong lớp cơ sở.

Về nguyên tắc, những phần của đối tượng lớp dẫn xuất thuộc về lớp cơ sở sẽ được tạo ra trước khi các thông tin mới được xác lập. Như vậy, thứ tự thực hiện của các hàm thiết lập sẽ là: hàm thiết lập cho lớp cơ sở, rồi đến hàm thiết lập cho lớp dẫn xuất nhằm bổ sung những thông tin còn thiếu.

Cơ chế trên đây được thực hiện một cách ngầm định, không cần phải gọi tường minh hàm thiết lập lớp cơ sở trong hàm thiết lập lớp dẫn xuất (thực tế là cũng không thể thực hiện được điều đó vì không thể gọi hàm thiết lập của bất kỳ lớp nào một cách tường minh).

Giải pháp của C++ (cũng được nhiều ngôn ngữ lập trình hướng đối tượng khác chấp nhận) là: trong định nghĩa của hàm thiết lập lớp dẫn xuất, ta mô tả luôn một lời gọi tới một trong các hàm thiết lập lớp cơ sở. Hãy xem lại định nghĩa của các hàm thiết lập lớp `coloredpoint` trong chương trình inheri1.cpp:

Một số nhận xét quan trọng

(ii) Nếu muốn sử dụng hàm thiết lập ngầm định của lớp cơ sở thì có thể không cần mô tả cùng với định nghĩa của hàm thiết lập lớp dẫn xuất, nghĩa là định nghĩa của hàm `coloredpoint::coloredpoint()` có thể viết lại như sau:

```
coloredpoint() { /*để tạo phạn đối tượng point sử dụng hàm point::point()* /
   color =0;
   }
```

(iii) Các tham số mà hàm thiết lập lớp dẫn xuất truyền cho hàm thiết lập lớp cơ sở không nhất thiết lấy nguyên si từ các tham số nó nhận được mà có thể được chế biến đi ít nhiều. Ví dụ, ta có thể viết lại định nghĩa cho `coloredpoint::coloredpoint(float, float,unsigned)` như sau:

```
coloredpoint::coloredpoint
  (float ox, float oy, unsigned c) : point((ox+oy)/2, (ox-oy)/2)
  {
   color = c;
  }
```

1.5.3 Hàm thiết lập sao chép

Nếu trong lớp dẫn xuất không khai báo tường minh hàm thiết lập sao chép, thì công việc này được chương trình biên dịch đảm nhiệm nhờ định nghĩa hàm thiết lập sao chép ngầm định.

Về nguyên tắc, trong định nghĩa của hàm thiết lập sao chép lớp dẫn xuất ta có thể mô tả bất kỳ hàm thiết lập nào có mặt trong lớp cơ sở. Chương trình sau minh hoạ điều đó:

Ví dụ 5.5

```
/*inheri4.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  float getx() {return x;}
  float gety() {return y;}
  point() {x = 0; y = 0;}
  point(float ox, float oy) {x = ox; y = oy; }
  point(point &p) {x = p.x; y = p.y;}
  void display() {
    cout<<"Goi ham point::display() \n";
    cout<<"Toa do :"<<x<<" "<<y<<endl;
    }
  void move(float dx, float dy) {
   x += dx;
   y += dy;
   }
 };
/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
  unsigned int color;
 public:
  coloredpoint():point() {
```

```
   color =0;
   }
  coloredpoint(float ox, float oy, unsigned int c);
  coloredpoint(coloredpoint &b):point(b.getx(),b.gety()) {
   cout<<"Goi ham thiet lap sao chep""
      <<" coloredpoint::coloredpoint(coloredpoint &) \n";
   color = b.color;
   }
 void display() {
  cout<<"Ham coloredpoint::display()\n";
  point::display();
  cout<<"Mau "<<color<<endl;
  }
 };
coloredpoint::coloredpoint
  (float ox, float oy, unsigned c) : point(ox, oy)
 {
  color = c;
 }
void main() {
 clrscr();
 coloredpoint pc(2,3,5);
 cout<<"pc = ";
 pc.display();
 cout<<"coloredpoint qc = pc;\n";
 coloredpoint qc = pc;
 cout<<"qc= ";
 qc.display();
 getch();
 }
```

```
pc = Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
coloredpoint qc = pc;
Goi ham thiet lap sao chep coloredpoint::coloredpoint(coloredpoint &)
qc= Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
```

Trong thực tế, ta thường sử dụng một cách làm khác, gọi hàm thiết lập sao chép của lớp cơ sở trong định nghĩa của hàm thiết lập sao chép lớp dẫn xuất. Cách tiếp cận này yêu cầu phải tách cho được tham chiếu đến đối tượng lớp cơ sở để dùng làm tham số của hàm thiết lập cơ sở.

Ta xét định nghĩa hàm

`coloredpoint::coloredpoint(coloredpoint&)`

trong chương trình sau:

Ví dụ 5.6

```
/*inheri5.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  point() {x = 0; y = 0;}
  point(float ox, float oy) {x = ox; y = oy; }
  point(point &p) {x = p.x; y = p.y;}
  void display() {
    cout<<"Goi ham point::display() \n";
    cout<<"Toa do :"<<x<<" "<<y<<endl;
    }
  void move(float dx, float dy) {
```

```
   x += dx;
   y += dy;
   }
 };
/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
  unsigned int color;
 public:
  coloredpoint():point() {
   color =0;
   }
  coloredpoint(float ox, float oy, unsigned int c):point(ox,oy) {
   color = c;
   }
  coloredpoint(coloredpoint &b):point((point &)b) {
   color = b.color;
   }
 void display() {
  cout<<"Ham coloredpoint::display()\n";
  point::display();
  cout<<"Mau "<<color<<endl;
  }
 };
void main() {
 clrscr();
 coloredpoint m(2,3,5);
 cout<<"Diem m \n";
 m.display();
 cout<<"coloredpoint p =m;\n";
 coloredpoint p =m;
 cout<<"Diem p \n";
 p.display();
```

```
 getch();
 }
```

```
Diem m
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
coloredpoint p =m;
Diem p
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
```

Nếu bạn đọc còn nhớ phần 2.4.3 của chương này thì sẽ thấy rằng định nghĩa của `coloredpoint::coloredpoint(coloredpoint &)` có thể cải tiến thêm một chút như sau:

```
coloredpoint(coloredpoint &b):point(b) {
   color = b.color;
   }
```

Thực tế ở đây có sự chuyển kiểu ngầm định từ `coloredpoint &` sang `point &`.

1.6 Các kiểu dẫn xuất khác nhau.

Tuỳ thuộc vào từ khoá đứng trước lớp cơ sở trong khai báo lớp dẫn xuất, người ta phân biệt ba loại dẫn xuất như sau:

Từ khoá	Kiểu dẫn xuất
public	dẫn xuất `public`
private	dẫn xuất `private`
protected	dẫn xuất `protected`

1.6.1 Dẫn xuất public

Trong dẫn xuất **public**, các thành phần, các hàm bạn và các đối tượng của lớp dẫn xuất không thể truy nhập đến các thành phần **private** của lớp cơ sở.

Các thành phần **protected** trong lớp cơ sở trở thành các thành phần **private** trong lớp dẫn xuất.

Các thành phần **public** của lớp cơ sở vẫn là **public** trong lớp dẫn xuất.

1.6.2 Dẫn xuất private

Trong trường hợp này, các thành phần **public** trong lớp cơ sở không thể truy nhập được từ các đối tượng của lớp dẫn xuất, nghĩa là chúng trở thành các thành phần **private** trong lớp dẫn xuất.

Các thành phần **protected** trong lớp cơ sở có thể truy nhập được từ các hàm thành phần và các hàm bạn của lớp dẫn xuất.

Dẫn xuất **private** được sử dụng trong một số tình huống đặc biệt khi lớp dẫn xuất không khai báo thêm các thành phần hàm mới mà chỉ định nghĩa lại các phương thức đã có trong lớp cơ sở.

1.6.3 Dẫn xuất protected

Trong dẫn xuất loại này, các thành phần **public, protected** trong lớp cơ sở trở thành các thành phần **protected** trong lớp dẫn xuất.

Bảng tổng kết các kiểu dẫn xuất

LỚP CƠ SỞ			**DẪN XUẤT `public`**		**DẪN XUẤT `protected`**		**DẪN XUẤT `private`**	
TTĐ	**TN FMA**	**TN NSD**	**TTM**	**TN NSD**	**TTM**	**TN NSD**	**TTM**	**TN NSD**
`pub`	C	C	`pub`	C	`pro`	K	`pri`	K
`pro`	C	K	`pro`	K	`pro`	K	`pri`	K
`pri`	C	K	`pri`	K	`pri`	K	`pri`	K

Ghi chú

TỪ VIẾT TẮT	**DIỄN GIẢI**
TTĐ	TRẠNG THÁI ĐẦU
TTM	TRẠNG THÁI MỚI
TN FMA	TRUY NHẬP BỞI CÁC HÀM THÀNH PHẦN HOẶC HÀM BẠN.
TN NSD	TRUY NHẬP BỞI NGƯỜI SỬ DỤNG
PRO	PROTECTED

PUB	PUBLIC
PRI	PRIVATE
C	CÓ
K	KHÔNG

HÀM ẢO VÀ TÍNH ĐA HÌNH

1.7 Đặt vấn đề

Cho đến nay, ta chỉ mới biết rằng một tham trỏ tới một đối tượng có thể nhận địa chỉ của bất cứ đối tượng con cháu nào (các đối tượng của các lớp thừa kế). Tuy vậy ưu điểm này phải trả giá: lời gọi đến một phương thức của một đối tượng được trỏ tới luôn được coi như lời gọi đến phương thức tương ứng với kiểu con trỏ chứ không phải tương ứng với đối tượng đang được trỏ đến. Ta đã có dịp minh chứng điều này trong ví dụ ở phần 2.4.2 của chương này. Ta gọi đó là "gán kiểu tĩnh-static typing" hay "gán kiểu sớm-early binding" (tức là hàm thành phần gọi từ con trỏ đối tượng được xác định ngay khi khai báo).

Thực tế có nhiều vấn đề khi xác định phương thức hay hành động cụ thể tuỳ thuộc vào thời điểm tham chiếu đối tượng, chẳng hạn khi vẽ các đối tượng hình chữ nhật, hình vuông, hình tròn, tất cả đều gọi tới phương thức vẽ được thừa kế từ lớp tổng quát hơn (lớp hình vẽ). Trong định nghĩa của phương thức đó có lời gọi đến một phương thức khác là nối điểm chỉ được xác định một cách tường minh trong từng đối tượng cụ thể là hình chữ nhật, hình vuông hay hình tròn.

Để gọi được phương thức tương ứng với đối tượng được trỏ đến, cần phải xác định được kiểu của đối tượng được xem xét tại thời điểm thực hiện chương trình bởi lẽ kiểu của đối tượng được chỉ định bởi cùng một con trỏ có thể thay đổi trong quá trình thực hiện của chương trình. Ta gọi đó là "gán kiểu động - dynamic typing" hay "gán kiểu muộn - late binding", nghĩa là xác định hàm thành phần nào tương ứng với một lời gọi hàm thành phần từ con trỏ đối tượng phụ thuộc cụ thể vào đối tượng mà con trỏ đang chứa địa chỉ. Khái niệm *hàm ảo* được C++ đưa ra nhằm đáp ứng nhu cầu này. Chương trình `polymorphism1.cpp` sau đây đưa ra một ví dụ về hàm ảo:

Ví dụ 5.7

```
/*polymorphism1.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  point() {
```

```
    cout<<"point::point()\n";
    x = 0;
    y = 0;
    }
   point(float ox, float oy) {
    cout<<"point::point(float, float)\n";
    x = ox;
    y = oy;
    }
   point(point &p) {
    cout<<"point::point(point &)\n";
    x = p.x;
    y = p.y;
    }
    virtual void display() {
    cout<<"Goi ham point::display() \n";
    cout<<"Toa do :"<<x<<" "<<y<<endl;
    }
  void move(float dx, float dy) {
    x += dx;
    y += dy;
    }
 };
/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
   unsigned int color;
 public:
  coloredpoint():point() {
    cout<<"coloredpoint::coloredpoint()\n";
    color =0;
    }
  coloredpoint(float ox, float oy, unsigned int c);
```

```
 coloredpoint(coloredpoint &b):point((point &)b) {
  cout<<"coloredpoint::coloredpoint(coloredpoint &)\n";
  color = b.color;
  }
 void display() {
  cout<<"Ham coloredpoint::display()\n";
  point::display();
  cout<<"Mau "<<color<<endl;
  }
 };
coloredpoint::coloredpoint(float ox, float oy, unsigned c) : point(ox,
oy){
  cout<<"coloredpoint::coloredpoint(float, float, unsigned)\n";
  color = c;
  }
void main() {
 clrscr();
 cout<<"coloredpoint pc(2,3,5);\n";
 coloredpoint pc(2,3,5);
 cout<<"pc.display();\n";
 pc.display();
 cout<<"point *ptr=&pc;\n";
 point *ptr=&pc;
 cout<<"ptr->display();\n";
 ptr->display();
 cout<<"point p(10,20);\n";
 point p(10,20);
 cout<<"ptr = &p\n";
 ptr = &p;
 cout<<"p.display()\n";
 p.display();
 cout<<"ptr->display();\n";
```

```
 ptr->display();
 getch();
 }
```

```
coloredpoint pc(2,3,5);
point::point(float, float)
coloredpoint::coloredpoint(float, float, unsigned)
pc.display();
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
point *ptr=&pc;
ptr->display();
Ham coloredpoint::display()
Goi ham point::display()
Toa do :2 3
Mau 5
point p(10,20);
point::point(float, float)
ptr = &p
p.display()
Goi ham point::display()
Toa do :10 20
ptr->display();
Goi ham point::display()
Toa do :10 20
```

Nhận xét

11. Trong định nghĩa của lớp `point`, ở dòng tiêu đề khai báo hàm thành phần `point::display()` có từ khoá **virtual**, từ khoá này có thể đặt trước hay sau tên kiểu dữ liệu nhưng phải trước tên hàm để chỉ định rằng hàm `point::display()` là một hàm ảo.

12. Hàm thành phần `point::dislay()` được định nghĩa lại trong lớp dẫn xuất tuy rằng trong trường hợp tổng quát điều này không bắt buộc nếu như bản thân hàm ảo đã được đinh nghĩa. Khi đó tuỳ thuộc vào kiểu của đối tượng có địa chỉ chứa trong con trỏ lớp dẫn xuất `ptr` mà lời gọi hàm `ptr->display()` sẽ gọi đến `point::display()` hay `coloredpoint::display();`

Tính đa hình còn thể hiện khi một hàm thành phần trong lớp cơ sở được gọi từ một đối tượng lớp dẫn xuất, còn bản thân hàm đó thì gọi tới hàm thành phần được định nghĩa đồng thời trong cả lớp cơ sở (khai báo **virtual** có mặt ở đây) và trong các lớp dẫn xuất. Mời bạn đọc xem chương trình `polymorphism2.cpp` sau đây:

Ví dụ 5.8

```
/*polymorphism2.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  point() {
    cout<<"point::point()\n";
    x = 0;
    y = 0;
    }
  point(float ox, float oy) {
    cout<<"point::point(float, float)\n";
    x = ox;
    y = oy;
    }
  point(point &p) {
    cout<<"point::point(point &)\n";
    x = p.x;
    y = p.y;
    }
  void display() ;
  void move(float dx, float dy) {
    x += dx;
```

```
   y += dy;
   }
   virtual void Identifier() {
    cout<<"Diem khong mau \n";
   }
 };
void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<endl;
    Identifier();
   }
/*lớp coloredpoint thừa kế từ point*/
class coloredpoint : public point {
  unsigned int color;
 public:
  coloredpoint():point() {
   cout<<"coloredpoint::coloredpoint()\n";
   color =0;
   }
  coloredpoint(float ox, float oy, unsigned int c);
  coloredpoint(coloredpoint &b):point((point &)b) {
   cout<<"coloredpoint::coloredpoint(coloredpoint &)\n";
   color = b.color;
   }
   void Identifier() {
   cout<<"Mau : "<<color<<endl;
   }
 };
coloredpoint::coloredpoint(float ox, float oy, unsigned c)
                : point(ox, oy)
 {
   cout<<"coloredpoint::coloredpoint(float, float, unsigned)\n";
   color = c;
```

```
}
void main() {
 clrscr();
 cout<<"coloredpoint pc(2,3,5);\n";
 coloredpoint pc(2,3,5);
 cout<<"pc.display()\n";
 pc.display();
 cout<<"point p(10,20);\n";
 point p(10,20);
 cout<<"p.display()\n";
 p.display();
 getch();
 }
```

```
coloredpoint pc(2,3,5);
point::point(float, float)
coloredpoint::coloredpoint(float, float, unsigned)
pc.display()
Toa do : 2 3
Mau : 5
point p(10,20);
point::point(float, float)
p.display()
Toa do : 10 20
Diem khong mau
```

Trong chương trình trên, lớp `coloredpoint` thừa kế từ lớp `point` hàm thành phần `display()`. Hàm này gọi tới hàm thành phần ảo `Identifier` được định nghĩa lại trong `coloredpoint`. Tuỳ thuộc vào đối tượng gọi hàm `display()` là của `point` hay của `coloredpoint` chương trình dịch sẽ phát lời gọi đến `point::Identifier()` hay `coloredpoint::Identifier()`. Nói cách khác trong trường hợp này `Identifier` cũng biểu hiện tính đa hình.

1.8 Tổng quát về hàm ảo

1.8.1 Phạm vi của khai báo virtual

Khi một hàm `f()` được khai báo **virtual** ở trong một lớp A, nó được xem như thể hiện của sự ghép kiểu động trong lớp A và trong tất cả các lớp dẫn xuất từ A hoặc từ con cháu của A. Xét lời gọi tới hàm `Identifier()` trong hàm `display()` được gọi từ các đối tượng khác nhau trong chương trình sau:

Ví dụ 5.9

```
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  point() {
    cout<<"point::point()\n";
    x = 0;
    y = 0;
    }
  point(float ox, float oy) {
    cout<<"point::point(float, float)\n";
    x = ox;
    y = oy;
    }
  point(point &p) {
    cout<<"point::point(point &)\n";
    x = p.x;
    y = p.y;
    }
  void display() ;
  void move(float dx, float dy) {
   x += dx;
   y += dy;
   }
```

```
  virtual void Identifier() {
   cout<<"Diem khong mau \n";
   }
};
void point::display() {
   cout<<"Toa do : "<<x<<" "<<y<<endl;
   Identifier();
   }
class coloredpoint : public point {
  unsigned int color;
 public:
  coloredpoint():point() {
   cout<<"coloredpoint::coloredpoint()\n";
   color =0;
   }
  coloredpoint(float ox, float oy, unsigned int c);
  coloredpoint(coloredpoint &b):point((point &)b) {
   cout<<"coloredpoint::coloredpoint(coloredpoint &)\n";
   color = b.color;
   }
  void Identifier() {
   cout<<"Mau : "<<color<<endl;
   }
};
class threedimpoint : public point {
  float z;
 public:
  threedimpoint() {
   z = 0;
   }
  threedimpoint(float ox, float oy, float oz):point (ox, oy) {
   z = oz;
```

```
    }
   threedimpoint(threedimpoint &p) :point(p) {
    z = p.z;
    }
   void Identifier() {
    cout<<"Toa do z : "<<z<<endl;
    }
 };
class coloredthreedimpoint : public threedimpoint {
   unsigned color;
 public:
   coloredthreedimpoint() {
    color = 0;
    }
   coloredthreedimpoint(float ox, float oy, float oz,unsigned c):
                threedimpoint (ox, oy, oz)
    {
    color = c;
    }
   coloredthreedimpoint(coloredthreedimpoint &p) :threedimpoint(p) {
    color = p.color;
    }
   void Identifier() {
    cout<<"Diem mau : "<<color<<endl;
    }
 };

coloredpoint::coloredpoint(float ox, float oy, unsigned c) :
                point(ox, oy)
  {
   cout<<"coloredpoint::coloredpoint(float, float, unsigned)\n";
    color = c;
```

```
 }
void main() {
 clrscr();
 cout<<"coloredpoint pc(2,3,5);\n";
 coloredpoint pc(2,3,5);
 cout<<"pc.display()\n";
 pc.display();/*gọi tới coloredtpoint::Identifier()*/
 cout<<"point p(10,20);\n";
 point p(10,20);
 cout<<"p.display()\n";
 p.display();/*gọi tới point::Identifier()*/
 cout<<"threedimpoint p3d(2,3,4);\n";
 threedimpoint p3d(2,3,4);
 cout<<"p3d.display();\n";
 p3d.display();/*gọi tới threedimpoint::Identifier()*/
 cout<<"coloredthreedimpoint p3dc(2,3,4,10);\n";
 coloredthreedimpoint p3dc(2,3,4,10);
 cout<<"p3dc.display();\n";
 p3dc.display();/*gọi tới coloredthreedimpoint::Identifier()*/
 getch();
 }
```

```
coloredpoint pc(2,3,5);
point::point(float, float)
coloredpoint::coloredpoint(float, float, unsigned)
pc.display()
Toa do : 2 3
Mau : 5
point p(10,20);
point::point(float, float)
p.display()
Toa do : 10 20
```

```
Diem khong mau
threedimpoint p3d(2,3,4);
point::point(float, float)
p3d.display();
Toa do : 2 3
Toa do z : 4
coloredthreedimpoint p3dc(2,3,4,10);
point::point(float, float)
p3dc.display();
Toa do : 2 3
Diem mau  : 10
```

1.8.2 Không nhất thiết phải định nghĩa lại hàm virtual

Trong trường hợp tổng quát, ta luôn phải định nghĩa lại ở trong các lớp dẫn xuất các phương thức đã được khai báo là **virtual** trong lớp cơ sở. Trong một số trường hợp không nhất thiết buộc phải làm như vậy. Khi mà hàm `display()` đã có một định nghĩa hoàn chỉnh trong `point`, ta không cần định nghĩa lại nó trong lớp `coloredpoint`. Chương trình `polymorphism4.cpp` sau đây minh chứng cho nhận định này.

Ví dụ 5.10

```
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  point() {
   cout<<"point::point()\n";
   x = 0;
   y = 0;
   }
  point(float ox, float oy) {
   cout<<"point::point(float, float)\n";
   x = ox;
```

```
    y = oy;
    }
   point(point &p) {
    cout<<"point::point(point &)\n";
    x = p.x;
    y = p.y;
    }
   virtual void display() {
    cout<<"Goi ham point::display() \n";
    cout<<"Toa do :"<<x<<" "<<y<<endl;
    }
  void move(float dx, float dy) {
   x += dx;
   y += dy;
   }
 };
class coloredpoint : public point {
  unsigned int color;
 public:
  coloredpoint():point() {
   cout<<"coloredpoint::coloredpoint()\n";
   color =0;
   }
  coloredpoint(float ox, float oy, unsigned int c);
  coloredpoint(coloredpoint &b):point((point &)b) {
   cout<<"coloredpoint::coloredpoint(coloredpoint &)\n";
   color = b.color;
   }
 };
coloredpoint::coloredpoint(float ox, float oy, unsigned c) :
                point(ox, oy)
 {
```

```
   cout<<"coloredpoint::coloredpoint(float, float, unsigned)\n";
   color = c;
 }
void main() {
 clrscr();
 cout<<"coloredpoint pc(2,3,5);\n";
 coloredpoint pc(2,3,5);
 cout<<"pc.display();\n";
 pc.display();
 cout<<"point *ptr=&pc;\n";
 point *ptr=&pc;
 cout<<"ptr->display();\n";
 ptr->display();
 cout<<"point p(10,20);\n";
 point p(10,20);
 cout<<"ptr = &p\n";
 ptr = &p;
 cout<<"p.display()\n";
 p.display();
 cout<<"ptr->display();\n";
 ptr->display();
 getch();
 }
```

```
coloredpoint pc(2,3,5);
point::point(float, float)
coloredpoint::coloredpoint(float, float, unsigned)
pc.display();
Goi ham point::display()
Toa do :2 3
point *ptr=&pc;
ptr->display();
```

```
Goi ham point::display()
Toa do :2 3
point p(10,20);
point::point(float, float)
ptr = &p
p.display()
Goi ham point::display()
Toa do :10 20
ptr->display();
Goi ham point::display()
Toa do :10 20
```

1.8.3 Định nghĩa chồng hàm ảo

Có thể định nghĩa chồng một hàm ảo và các hàm định nghĩa chồng như thế có thể không còn là hàm ảo nữa.

Hơn nữa, nếu ta định nghĩa một hàm ảo trong một lớp và lại định nghĩa chồng nó trong một lớp dẫn xuất với các tham số khác thì có thể xem hàm định nghĩa chồng đó là một hàm hoàn toàn khác không liên quan gì đến hàm ảo hiện tại, nghĩa là nếu nó không được khai báo **virtual** thì nó có tính chất "gán kiểu tĩnh-static typing". Nói chung, nếu định nghĩa chồng hàm ảo thì tất cả các hàm định nghĩa chồng của nó nên được khai báo là **virtual** để việc xác định lời gọi hàm đơn giản hơn.

1.8.4 Khai báo hàm ảo ở một lớp bất kỳ trong sơ đồ thừa kế

Trong chương trình `polymorphism5.cpp`, hàm `point::Identifier()` không là **virtual** trong khi đó khai báo **virtual** lại được áp dụng cho hàm `threedimpoint::Identifier()`. Chương trình dịch sẽ xem `point::Identifier()` và `threedimpoint::Identifier()` có tính chất khác nhau: `point::Identifier()` có tính chất "gán kiểu tĩnh- static typing", trong khi đó `threedimpoint::Identifier()` lại có tính chất "gán kiểu động-dynamic typing".

Ví dụ 5.11

```
/*polymorphism5.cpp*/
#include <iostream.h>
#include <conio.h>
class point {
```

```
  float x,y;
 public:
  point() {
   cout<<"point::point()\n";
   x = 0;
   y = 0;
   }
  point(float ox, float oy) {
   cout<<"point::point(float, float)\n";
   x = ox;
   y = oy;
   }
  point(point &p) {
   cout<<"point::point(point &)\n";
   x = p.x;
   y = p.y;
   }
  void display() ;
  void move(float dx, float dy) {
   x += dx;
   y += dy;
   }
  void Identifier() {
   cout<<"Diem khong mau \n";
   }
 };
void point::display() {
   cout<<"Toa do : "<<x<<" "<<y<<endl;
   Identifier();
   }
class threedimpoint : public point {
  float z;
```

```
public:
  threedimpoint() {
   z = 0;
   }
  threedimpoint(float ox, float oy, float oz):point (ox, oy) {
   z = oz;
   }
  threedimpoint(threedimpoint &p) :point(p) {
   z = p.z;
   }
  virtual void Identifier() {
   cout<<"Toa do z : "<<z<<endl;
   }
};
class coloredthreedimpoint : public threedimpoint {
  unsigned color;
public:
  coloredthreedimpoint() {
   color = 0;
   }
  coloredthreedimpoint(float ox, float oy, float oz,unsigned c):
              threedimpoint (ox, oy, oz)
   {
    color = c;
   }
  coloredthreedimpoint(coloredthreedimpoint &p) :threedimpoint(p) {
   color = p.color;
   }
  void Identifier() {
   cout<<"Diem mau : "<<color<<endl;
   }
};
```

```
void main() {
 clrscr();
 cout<<"point p(10,20);\n";
 point p(10,20);
 cout<<"p.display()\n";
 p.display();
 cout<<"threedimpoint p3d(2,3,4);\n";
 threedimpoint p3d(2,3,4);
 cout<<"p3d.display();\n";
 p3d.display();
 cout<<"p3d.Identifier();\n";
 p3d.Identifier();
 cout<<"coloredthreedimpoint p3dc(2,3,4,10);\n";
 coloredthreedimpoint p3dc(2,3,4,10);
 cout<<"p3dc.display();\n";
 p3dc.display();
 cout<<"p3dc.Identifier();\n";
 p3dc.Identifier();
 getch();
 }
```

```
point p(10,20);
point::point(float, float)
p.display()
Toa do : 10 20
Diem khong mau
threedimpoint p3d(2,3,4);
point::point(float, float)
p3d.display();
Toa do : 2 3
Diem khong mau
p3d.Identifier();
```

```
Toa do z : 4
coloredthreedimpoint p3dc(2,3,4,10);
point::point(float, float)
p3dc.display();
Toa do : 2 3
Diem khong mau
p3dc.Identifier();
Diem mau  : 10
```

1.8.5 Hàm huỷ bỏ ảo

Hàm thiết lập không thể là hàm ảo, trong khi đó hàm huỷ bỏ lại có thể. Ta quan sát sự cố xảy ra khi sử dụng tính đa hình để xử lý các đối tượng của các lớp trong sơ đồ thừa kế được cấp phát động. Nếu mỗi đối tượng được dọn dẹp tường minh nhờ sử dụng toán tử **delete** cho con trỏ lớp cơ sở chỉ đến đối tượng thì hàm huỷ bỏ của lớp cơ sở sẽ được gọi mà không cần biết kiểu của đối tượng đang được xử lý, cũng như tên hàm huỷ bỏ của lớp tương ứng với đối tượng (tuy có thể khác với hàm huỷ bỏ của lớp cơ sở).

Một giải pháp đơn giản cho vấn đề này là khai báo hàm huỷ bỏ của lớp cơ sở là hàm ảo, làm cho các hàm huỷ bỏ của các lớp dẫn xuất là ảo mà không yêu cầu chúng phải có cùng tên. Chương trình `polymorphism6.cpp` sau đây minh hoạ tính đa hình của hàm ảo:

Ví dụ 5.12

```
#include <iostream.h>
#include <conio.h>
class point {
  float x,y;
 public:
  point() {
    cout<<"point::point()\n";
    x = 0;
    y = 0;
    }
  point(float ox, float oy) {
    cout<<"point::point(float, float)\n";
    x = ox;
```

```
    y = oy;
    }
  point(point &p) {
    cout<<"point::point(point &)\n";
    x = p.x;
    y = p.y;
    }
  virtual ~point() {
    cout<<"point::~point() \n";
    }
  void display() ;
  void move(float dx, float dy) {
    x += dx;
    y += dy;
    }
  virtual void Identifier() {
    cout<<"Diem khong mau \n";
    }
 };
void point::display() {
    cout<<"Toa do : "<<x<<" "<<y<<endl;
    Identifier();
    }
class threedimpoint : public point {
  float z;
 public:
  threedimpoint() {
    z = 0;
    }
  threedimpoint(float ox, float oy, float oz):point (ox, oy) {
    cout<<"threedimpoint::threedimpoint(float, float,float)\n";
    z = oz;
```

```
    }
   threedimpoint(threedimpoint &p) :point(p) {
    z = p.z;
    }
   ~threedimpoint() {
    cout<<"threedimpoint::~threedimpoint()";
    }
   void Identifier() {
    cout<<"Toa do z : "<<z<<endl;
    }
};
class coloredthreedimpoint : public threedimpoint {
   unsigned color;
 public:
   coloredthreedimpoint() {
    color = 0;
    }
   coloredthreedimpoint(float ox, float oy, float oz,unsigned c):
                threedimpoint (ox, oy, oz)
    {
    cout<<"coloredthreedimpoint::coloredthreedimpoint(float,
float,float,unsigned)\n";
    color = c;
    }
   coloredthreedimpoint(coloredthreedimpoint &p) :threedimpoint(p) {
    color = p.color;
    }
   ~coloredthreedimpoint() {
    cout<<"coloredthreedimpoint::~coloredthreedimpoint()\n";
    }
   void Identifier() {
    cout<<"Diem mau : "<<color<<endl;
```

```
    }
 };
void main() {
 clrscr();
 point *p0 = new point(2,10);
 point *p1 = new threedimpoint(2,3,5);
 point *p2 = new coloredthreedimpoint(2,3,4,10);
 delete p0;
 delete p1;
 delete p2;
 getch();
 }
```

```
point::point(float, float)
point::point(float, float)
threedimpoint::threedimpoint(float, float,float)
point::point(float, float)
threedimpoint::threedimpoint(float, float,float)
coloredthreedimpoint::coloredthreedimpoint(float,
float,float,unsigned)
point::~point()
threedimpoint::~threedimpoint()point::~point()
coloredthreedimpoint::~coloredthreedimpoint()
threedimpoint::~threedimpoint()point::~point()
```

1.9 Lớp trừu tượng và hàm ảo thuần tuý

Hàm ảo thuần tuý là hàm không có phần định nghĩa. Định nghĩa một hàm ảo như thế được viết như sau:

```
class mere {
  public:
   virtual display() = 0; //hảm ảo thuần tuý
 };
```

Lớp có ít nhất một hàm thành phần ảo thuần tuý được gọi là lớp trừu tượng. Phải tuân theo một số quy tắc sau đây:

(iv) Không thể sử dụng lớp trừu tượng khi khai báo các biến, mà để mô tả lớp các đối tượng một cách chung nhất. Sau đó các lớp thừa kế sẽ được chi tiết dần dần. Ở đây, lớp trừu tượng là tổng quát hoá của các lớp thừa kế nó, và ngược lại các lớp dẫn xuất là sự cụ thể hoá của lớp trừu tượng.

(v) Được phép khai báo con trỏ có kiểu là một lớp trừu tượng.

(vi) Một hàm ảo thuần tuý khai báo trong một lớp trừu tượng cơ sở phải được định nghĩa lại trong một lớp dẫn xuất hoặc nếu không thì phải được tiếp tục khai báo ảo trong lớp dẫn xuất. Trong trường hợp này lớp dẫn xuất mới này lại là một lớp trừu tượng. Trong phần 4.5 sẽ có một ví dụ về việc sử dụng lớp trừu tượng để xây dựng một danh sách các đối tượng không đồng nhất.

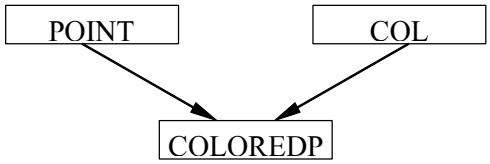
ĐA THỪA KẾ

1.10 Đặt vấn đề

Đa thừa kế cho phép một lớp có thể là dẫn xuất của nhiều lớp cơ sở, do vậy những gì đã đề cập trong phần đơn thừa kế được tổng quát hoá cho trường hợp đa thừa kế. Tuy vậy cần phải giải quyết các vấn đề sau:

13. Làm thế nào biểu thị được tính độc lập của các thành phần cùng tên bên trong một lớp dẫn xuất?
14. Các hàm thiết lập và huỷ bỏ được gọi như thế nào: thứ tự, truyền thông tin v.v.?
15. Làm thế nào giải quyết tình trạng thừa kế xung đột trong đó, lớp D dẫn xuất từ B và C, và cả hai cùng là dẫn xuất của A.

Chúng ta xem xét một tình huống đơn giản : một lớp có tên là `coloredpoint` thừa kế từ hai lớp `point` và `col`.



Giả sử các lớp `point` và `col` được trình bày như sau:

```
class point {                    class col {
        float x,y;                       unsigned color;
 public:                          public:
        point(...){...}                  col (...) {...}
        ~point{...}                      ~color() {...}
```

```
        void display(){...}
};
```

```
        void display(){...}
};
```

Chúng ta có thể định nghĩa lớp `coloredpoint` thừa kế hai lớp này như sau:

```
class coloredpoint:public point,public col
{...};
```

Quan sát câu lệnh khai báo trên ta thấy tiếp theo tên lớp dẫn xuất là một danh sách các lớp cơ sở cùng với các từ khoá xác định kiểu dẫn xuất.

Bên trong lớp `coloredpoint` ta có thể định nghĩa các thành phần mới (ở đây giới hạn với hàm thiết lập, hàm huỷ bỏ và hàm hiển thị).

Trong trường hợp đơn thừa kế, hàm thiết lập lớp dẫn xuất phải truyền một số thông tin cho hàm thiết lập lớp cơ sở. Điều này vẫn cần trong trường hợp đa thừa kế, chỉ khác là phải truyền thông tin cho hai hàm thiết lập tương ứng với hai hai lớp cơ sở. Dòng tiêu đề trong phần định nghĩa hàm thiết lập `coloredpoint` có dạng:

```
coloredpoint(.............)point(.............),col(.............)
```

CÁC THAM SỐ CHO

CÁC THAM SỐ

CÁC THAM SỐ

Thứ tự gọi các hàm thiết lập như sau: các hàm thiết lập của các lớp cơ sở theo thứ tự khai báo của các lớp cơ sở trong lớp dẫn xuất (ở đây là `point` và `col`), và cuối cùng là hàm thiết lập của lớp dẫn xuất (ở đây là `coloredpoint`).

Còn các hàm huỷ bỏ lại được gọi theo thứ tự ngược lại khi đối tượng `coloredpoint` bị xoá.

Giống như trong đơn thừa kế, trong hàm thành phần của lớp dẫn xuất có thể sử dụng tất cả các hàm thành phần **public** (hoặc **protected**) của lớp cơ sở.

Khi có nhiều hàm thành phần cùng tên trong các lớp khác nhau (định nghĩa lại một hàm), ta có thể loại bỏ sự nhập nhằng bằng cách sử dụng toán tử phạm vi "`::`". Ngoài ra để tạo thói quen lập trình tốt, khi thực hiện các lời gọi hàm thành phần nên tuân theo thứ tự khai báo của các lớp cơ sở. Như vậy bên trong hàm `coloredpoint::display()` có các câu lệnh sau:

```
point::display();
col::display();
```

Tất nhiên khi hàm thành phần trong lớp cơ sở không được định nghĩa lại trong lớp dẫn xuất thì sẽ không có sự nhập nhằng và do đó không cần dùng đến toán tử "`::`".

Việc sử dụng lớp `coloredpoint` không có gì đặc biệt, một đối tượng kiểu `coloredpoint` thực hiện lời gọi đến các hàm thành phần của `coloredpoint` cũng như là các hàm thành phần **public** của `point` và `col`. Nếu có trùng tên hàm, phải sử dụng toán tử phạm vi "**::**". Chẳng hạn với khai báo:

```
coloredpoint p(3,9,2);
```

câu lệnh

```
p.display()
```

sẽ gọi tới

```
coloredpoint::display(),
```

còn câu lệnh

```
p.point::display()
```

gọi tới

```
point::display().
```

Nếu một trong hai lớp `point` và `col` lại là dẫn xuất của một lớp cơ sở khác, thì có thể sử dụng được các thành phần bên trong lớp cơ sở mới này. Sau đây là một ví dụ hoàn chỉnh minh hoạ việc định nghĩa và sử dụng `coloredpoint`.

Ví dụ 5.13

```
/*mulinher1.cpp*/
#include <iostream.h>
#include <conio.h>
class point{
       float x,y;
 public:
       point (float ox,float oy)
         {
         cout<<"++Constr. point\n";
         x=ox;
         y=oy;
         }
       ~point(){cout<<"--Destr. point\n";}
       void display(){
```

```
        cout<<"Toa do : "<<x<<" "<<y<<"\n";
        }
};

class col {
       unsigned color;
public:
       col(unsigned c)
        {
        cout<<"++Constr. col \n";
        color=c;
        }
       ~col() {cout<<"--Destr. col\n";}
       void display() {cout<<"Mau : "<<color<<"\n";}
};

class coloredpoint :public point,public col {
public:
       coloredpoint(float ox,float oy, unsigned c) : point(ox,oy),col(c)
        {
        cout<<"++Constr. coloredpoint\n";
        }
       ~coloredpoint() {
        cout<<"--Destr. coloredpoint\n";
        }
       void display() {
        point::display();
        col::display();
        }
};
void main()
{
```

```
clrscr();
coloredpoint p(3,9,2);
cout<<"---------------\n";
p.display();
cout<<"---------------\n";
p.point::display();
cout<<"---------------\n";
p.col::display();
cout<<"---------------\n";
getch();
}
```

```
++Constr. point
++Constr. col
++Constr. coloredpoint
---------------
Toa do : 3 9
Mau : 2
---------------
Toa do : 3 9
---------------
Mau : 2
---------------
--Destr. coloredpoint
--Destr. col
--Destr. point
```

Nhận xét

Trong trường hợp các thành phần dữ liệu của các lớp cơ sở trùng tên, ta phân biệt chúng bằng cách sử dụng tên lớp tương ứng đi kèm với toán tử phạm vi "::". Xét ví dụ:

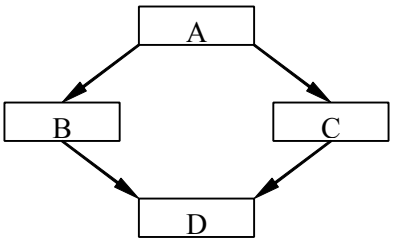
```
class A {                          class B {
   ...                                ...
  public:                           public:
   int x;                            int x;
   ...                               ...
   };                                };
```

```
class C:public A,public B
  {...};
```

Lớp `C` có hai thành phần dữ liệu cùng tên là `X`, một thừa kế từ `A`, một thừa kế từ `B`. Bên trong các thành phần hàm của C, chúng ta phân biệt chúng nhờ toán tử phạm vi"`::`": đó là `A::x` và `B::x`

1.11 Lớp cơ sở ảo

Xét tình huống như sau:



tương ứng với các khai báo sau:

```
class A {
  ...
  int x,y;
 };
class B:public A{....};
class C:public A{....};
class D:public B,public C
 {....};
```

Theo một nghĩa nào đó, có thể nói rằng D "thừa kế" A hai lần. Trong các tình huống như vậy, các thành phần của A (hàm hoặc dữ liệu) sẽ xuất hiện trong D hai lần. Đối với các hàm thành phần thì điều này không quan trọng bởi chỉ có duy nhất một hàm cho một lớp cơ sở, các hàm thành phần là chung cho mọi đối tượng của lớp. Tuy nhiên, các thành phần dữ liệu lại được lặp lại trong các đối tượng khác nhau (thành phần dữ liệu của mỗi đối tượng là độc lập).

NHƯ VẬY, PHẢI CHĂNG CÓ SỰ DƯ THỪA DỮ LIỆU? CÂU TRẢ LỜI PHỤ THUỘC VÀO TỪNG TÌNH HUỐNG CỤ THỂ. NẾU CHÚNG TA MUỐN D CÓ HAI BẢN SAO DỮ LIỆU CỦA `A`, TA PHẢI PHÂN BIỆT CHÚNG NHỜ:

`A::B::x` và `A::C::x`

Thông thường, chúng ta không muốn dữ liệu bị lặp lại và giải quyết bằng cách chọn một trong hai bản sao dữ liệu để thao tác. Tuy nhiên điều đó thật chán ngắt và không an toàn.

Ngôn ngữ C++ cho phép chỉ tổ hợp một lần duy nhất các thành phần của lớp `A` trong lớp `D` nhờ khai báo trong các lớp `B` và `C` (chứ không phải trong `D`!) rằng lớp `A` là ảo (từ khoá **virtual**):

```
class B:public virtual A{....};
class C:public virtual A{....};
class D:public B,public C{....};
```

Việc chỉ thị `A` là ảo trong khai báo của `B` và `C` nghĩa là `A` sẽ chỉ xuất hiện một lần trong các con cháu của chúng. Nói cách khác, khai báo này không ảnh hưởng đến các lớp `B` và `C`. Chú ý rằng từ khoá **virtual** có thể được đặt trước hoặc sau từ khoá **public** (hoặc **private**, **protected**). Ta xét chương trình ví dụ sau đây:

Ví dụ 5.14

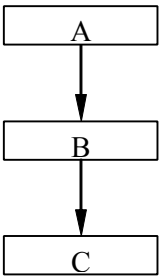
```
/*mulinher2.cpp
Solution to multiple inheritance*/
#include <iostream.h>
#include <conio.h>
class A {
  float x,y;
 public:
  void set(float ox, float oy) {
    x = ox; y = oy;
```

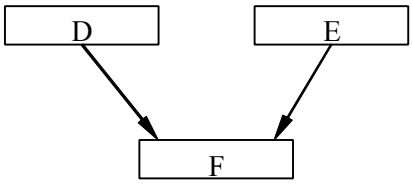
```
    }
   float getx() {
    return x;
    }
   float gety() {
    return y;
    }
 };
class B : public virtual A {
 };
class C : public virtual A {
 };
class D : public B, public C {
 };
void main() {
 clrscr();
 D d;
 cout<<"d.B::set(2,3);\n";
 d.B::set(2,3);
 cout<<"d.C::getx() = "; cout<<d.C::getx()<<endl;
 cout<<"d.B::getx() = "; cout<<d.B::getx()<<endl;
 cout<<"d.C::gety() = "; cout<<d.C::gety()<<endl;
 cout<<"d.B::gety() = "; cout<<d.B::gety()<<endl;

 cout<<"d.C::set(10,20);\n";
 d.B::set(2,3);
 cout<<"d.C::getx() = "; cout<<d.C::getx()<<endl;
 cout<<"d.B::getx() = "; cout<<d.B::getx()<<endl;
 cout<<"d.C::gety() = "; cout<<d.C::gety()<<endl;
 cout<<"d.B::gety() = "; cout<<d.B::gety()<<endl;
 getch();
 }
```

```
d.B::set(2,3);
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3
d.C::set(10,20);
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3
```

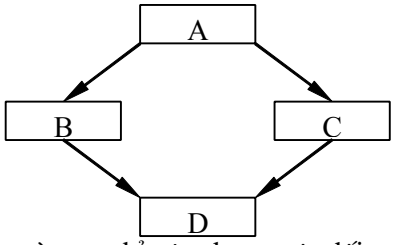
1.12 Hàm thiết lập và huỷ bỏ - với lớp ảo

Thông thường khi khởi tạo các đối tượng dẫn xuất các hàm thiết lập được gọi theo thứ tự xuất hiện trong danh sách các lớp cơ sở được khai báo, rồi đến hàm thiết lập của lớp dẫn xuất. Thông tin được chuyển từ hàm thiết lập của lớp dẫn xuất sang hàm thiết lập của các lớp cơ sở.





Trong tình huống có lớp cơ sở **virtual**, vấn đề có khác đôi chút. Xem hình vẽ sau:



Trong trường hợp này, ta chỉ xây dựng một đối tượng duy nhất có kiểu `A`. Các tham số được truyền cho hàm thiết lập `A` được mô tả ngay khi định nghĩa hàm thiết lập `D`. Và như vậy, không cần mô tả các thông tin truyền cho `A` ở mức `B` và `C`. Cần phải tuân theo quy tắc, quy định sau đây:

Thứ tự gọi các hàm thiết lập: hàm thiết lập của một lớp ảo luôn luôn được gọi trước các hàm thiết lập khác.

Với sơ đồ thừa kế có trên hình vẽ, thứ tự gọi hàm thiết lập sẽ là: A,B,C và cuối cùng là D. Chương trình sau minh chứng cho nhận xét này:

Ví dụ 5.15

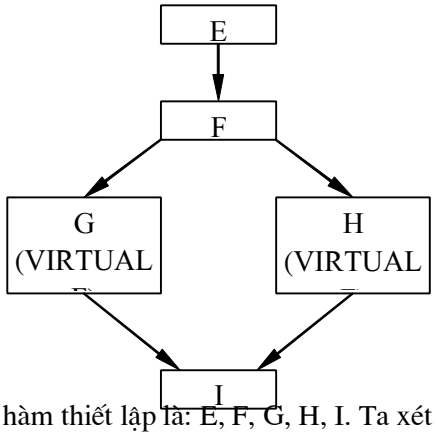
```
/*mulinher3.cpp
Solution to multiple inheritance*/
#include <iostream.h>
#include <conio.h>
class A {
  float x,y;
 public:
  A() {x = 0; y =0;}
  A(float ox, float oy) {
   cout<<"A::A(float, float)\n";
   x = ox; y = oy;
   }
  float getx() {
   return x;
   }
  float gety() {
   return y;
   }
 };
class B : public virtual A {
 public:
  B(float ox, float oy) : A(ox,oy) {
   cout<<"B::B(float, float)\n";
   }
 };
class C : public virtual A {
 public:
```

```
  C(float ox, float oy):A(ox,oy) {
    cout<<"C::C(float, float)\n";
    }
 };
class D : public B, public C {
 public:
  D(float ox, float oy) : A(ox,oy),B(10,4),C(1,1) {
    cout<<"D::D(float,float);\n";
    }
 };
void main() {
 clrscr();
 D d(2,3);
 cout<<"D d(2,3)\n";
 cout<<"d.C::getx() = "; cout<<d.C::getx()<<endl;
 cout<<"d.B::getx() = "; cout<<d.B::getx()<<endl;
 cout<<"d.C::gety() = "; cout<<d.C::gety()<<endl;
 cout<<"d.B::gety() = "; cout<<d.B::gety()<<endl;

 cout<<"D d1(10,20);\n";
 D d1(10,20);
 cout<<"d1.C::getx() = "; cout<<d1.C::getx()<<endl;
 cout<<"d1.B::getx() = "; cout<<d1.B::getx()<<endl;
 cout<<"d1.C::gety() = "; cout<<d1.C::gety()<<endl;
 cout<<"d1.B::gety() = "; cout<<d1.B::gety()<<endl;
 getch();
 }
```

```
A::A(float, float)
B::B(float, float)
C::C(float, float)
D::D(float,float);
```

```
D d(2,3)
d.C::getx() = 2
d.B::getx() = 2
d.C::gety() = 3
d.B::gety() = 3
D d1(10,20);
A::A(float, float)
B::B(float, float)
C::C(float, float)
D::D(float,float);
d1.C::getx() = 10
d1.B::getx() = 10
d1.C::gety() = 20
d1.B::gety() = 20
```

Ở tình huống khác:



thứ tự thực hiện các hàm thiết lập là: E, F, G, H, I. Ta xét chương trình sau:

Ví dụ 5.16

```
/*mulinher4.cpp
Solution to multiple inheritance*/
#include <iostream.h>
#include <conio.h>
class O {
   float o;
```

```
 public:
  O(float oo) {
   cout<<"O::O(float)\n";
   o = oo;
  }
};
class A :public O{
  float x,y;
 public:
  A(float oo,float ox, float oy):O(oo) {
   cout<<"A::A(float, float, float)\n";
   x = ox; y = oy;
  }
  float getx() {
   return x;
  }
  float gety() {
   return y;
  }
};
class B : public virtual A {
 public:
  B(float oo,float ox, float oy) : A(oo,ox,oy) {
   cout<<"B::B(float, float, float)\n";
  }
};
class C : public virtual A {
 public:
  C(float oo, float ox, float oy):A(oo,ox,oy) {
   cout<<"C::C(float, float, float)\n";
  }
};
```

```
class D : public B, public C {
 public:
  D(float oo, float ox, float oy) :
          A(oo,ox,oy),B(oo,10,4),C(oo,1,1) {
    cout<<"D::D(float, float, float);\n";
    }
 };
void main() {
 clrscr();
 D d(2,3,5);
 cout<<"D d(2,3,5)\n";
 cout<<"d.C::getx() = "; cout<<d.C::getx()<<endl;
 cout<<"d.B::getx() = "; cout<<d.B::getx()<<endl;
 cout<<"d.C::gety() = "; cout<<d.C::gety()<<endl;
 cout<<"d.B::gety() = "; cout<<d.B::gety()<<endl;
 cout<<"D d1(10,20,30);\n";
 D d1(10,20,30);
 cout<<"d1.C::getx() = "; cout<<d1.C::getx()<<endl;
 cout<<"d1.B::getx() = "; cout<<d1.B::getx()<<endl;
 cout<<"d1.C::gety() = "; cout<<d1.C::gety()<<endl;
 cout<<"d1.B::gety() = "; cout<<d1.B::gety()<<endl;
 getch();
 }
```

```
O::O(float)
A::A(float, float, float)
B::B(float, float, float)
C::C(float, float, float)
D::D(float, float, float);
D d(2,3,5)
d.C::getx() = 3
d.B::getx() = 3
```

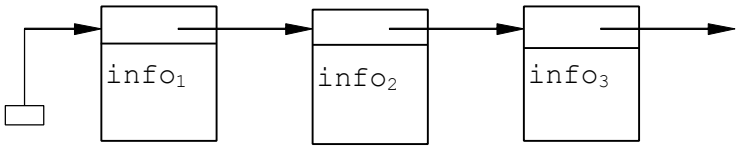
```
d.C::gety() = 5
d.B::gety() = 5
D d1(10,20,30);
O::O(float)
A::A(float, float, float)
B::B(float, float, float)
C::C(float, float, float)
D::D(float, float, float);
d1.C::getx() = 20
d1.B::getx() = 20
d1.C::gety() = 30
d1.B::gety() = 30
```

1.13 Danh sách móc nối các đối tượng

Trong phần này, chúng ta đưa ra cách xây dựng một lớp, cho phép quản lý một danh sách móc nối các đối tượng kiểu `point`. Các công việc cần phải làm là thiết kế một lớp đặc biệt để quản lý danh sách móc nối độc lập với kiểu đối tượng liên quan. Một lớp như vậy hoàn toàn mang ý nghĩa trừu tượng, cùng với `point` sẽ là lớp thừa kế cho một lớp dẫn xuất chứa danh sách móc nối.

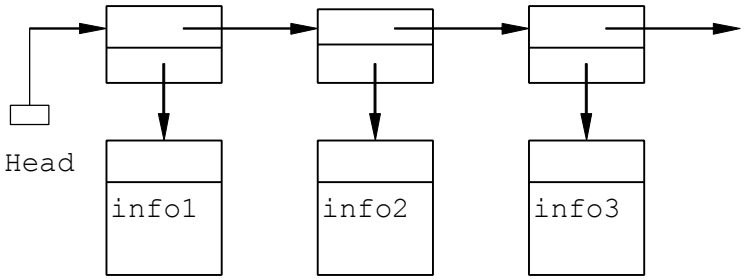
Xây dựng lớp trừu tượng

Ta giới hạn ở xây dựng một danh sách móc nối đơn giản, với các phần tử có hai thành phần: thành phần dữ liệu và thành phần con trỏ, chỉ tới phần tử đi tiếp sau. Sơ đồ của danh sách như vậy có dạng như sau:



ở đây `Head` là con trỏ đối tượng, chỉ đến phần tử đầu tiên của danh sách móc nối.

Tuy nhiên, có thể thông tin tương ứng với mỗi thành phần trong danh sách chưa được biết nên để biểu thị mỗi $info_i$ ta dùng một con trỏ chỉ tới một đối tượng dữ liệu nào đó (**void** *). Ta có sơ đồ như sau:



Ở đây, mỗi phần tử của danh sách chứa:

(vii) Một con trỏ đến phần tử tiếp theo,

(viii) Một con trỏ đến đối tượng chứa thông tin liên quan.

Còn lớp `list` để quản lý danh sách sẽ có ít nhất:

(ix) một thành phần dữ liệu: con trỏ đến phần tử đầu tiên (`Head`),

(x) một hàm thành phần có chức năng chèn vào danh sách một đối tượng tại một địa chỉ nào đó. Chú ý rằng địa chỉ này phải có kiểu **void** * để đảm bảo sự tương thích với các kiểu dữ liệu khác nhau.

Có thể hình dung sơ bộ lớp `list` được khai báo như sau:

```
struct element  //cấu trúc một phần tử của danh sách list
 {
 element *next;  //con trỏ đến thành phần đi sau
 void *content;      //con trỏ đến một đối tượng tuỳ ý
};
class list
  { element *head;    //con trỏ đến phần tử đầu tiên
   public:
     list();   //hàm thiết lập
     ~list();//hàm huỷ bỏ
     void add(void *);  //thêm một phần tử vào đầu danh sách
....
};
```

Để quản lý danh sách ta có thể thêm các chức năng xử lý khác như:

(xi) Hiển thị các đối tượng được chỉ bởi danh sách

(xii) Tìm kiếm một phần tử

(xiii) Xoá một phần tử

v.v...

Các thao tác này có thể được thực hiện trực tiếp trên `list` nhưng nhằm mục đích che dấu và do đặc điểm của các thông tin trong mỗi phần tử ta đều không biết, nên tốt nhất là để cho các hàm thành phần thực hiện.

Các thao tác trên danh sách được nêu ra ở trên đều gắn với cơ chế duyệt danh sách. Việc duyệt này cần phải được kiểm soát từ bên ngoài danh sách nhưng có thể dựa trên các thành phần hàm cơ sở là:

(xiv) Khởi tạo việc duyệt,

(xv) Chuyển sang phần tử tiếp theo.

Hai hàm thành phần này đều dùng tham số là con trỏ đến phần tử hiện tại, ta đặt tên cho nó là `current`. Ngoài ra, hai hàm thành phần trên sẽ trả về các thông tin liên quan phần tử hiện tại, có thể lựa chọn:

(xvi) Địa chỉ của phần tử hiện tại, kiểu là `element*`

(xvii) Địa chỉ của đối tượng hiện tại, có kiểu `content*`

(xviii) Nội dung dữ liệu của phần tử hiện tại.

Giải pháp đầu bao hàm giải pháp thứ hai bởi lẽ nếu `ptr` là địa chỉ của phần tử hiện tại trong danh sách, `ptr->content` sẽ cho địa chỉ của đối tượng dữ liệu tương ứng. Nhưng thực tế người lập trình không được quyền truy nhập đến các phần tử của danh sách `list`. Còn giải pháp thứ 3 làm mất khả năng kiểm tra các phần tử của danh sách và thay đổi chúng bởi lẽ chỉ với nội dung dữ liệu của phần tử hiện tại, ta không thể truy nhập tới các phần tử tiếp theo.

Ở đây, ta chọn giải pháp thứ 2. Trong trường hợp này, người ta vẫn chưa thể phát hiện phần tử cuối danh sách. Do vậy ta sẽ đưa vào một hàm thành phần bổ sung cho phép kiểm tra liệu đã gặp cuối danh sách hay chưa.

Theo các phân tích trên, ta sẽ dùng thêm ba hàm thành phần:

```
void *first();
void *prochain();
int last();
```

Sau đây là định nghĩa của lớp trừu tượng `list`:

Ví dụ 5.17

```
#include <stddef.h>//để định nghĩa NULL
```

```
//class list
struct element
 { element *next;
       void *content;
 };
class list
 {
       element *head;
       element *current;
 public:
       list()
        {
        head=NULL;current=head;
        }
       ~list();
       void add(void *)
       void first()
        {current=head;}
        void *prochain(){
       void *adel =NULL;
          if (current !=NULL) {
        adel =current->content;
              current=current->next;
             }
          return adel;
          }
       int last() {return (current==NULL);}
 };
list::~list() {
 element *suiv;
 current=head;
 while(current!=NULL)
```

```
        {suiv=current->next;delete current;current=suiv;}
 }
void list::add(void *chose) {
 element *adel= new element;
 adel->next =head;
 adel->content=chose;
 head=adel;
 }
```

Giả sử lớp `point` được định nghĩa như sau:

```
#include <iostream.h>
class point
 { int x,y;
 public:
        point(int abs=0,int ord=0) {x=abs;y=ord;}
        void display() {cout <<" Toa do : "<<x<," "<<y<<"\n";}
 };
```

Ta định nghĩa một lớp mới `list_points` thừa kế từ `list` và `point`, cho phép quản lý danh sách móc nối các đối tượng `point` như sau:

```
class list_points :public list,public point
 {
  public:
        list_points() {}
        void display();
 };
 void list_points::display()
  {
  first();
  while (!last()) {
    point *ptr=(point *)prochain();ptr->display();}
    }
```

Sau đây là chương trình hoàn chỉnh:

Ví dụ 5.18

```
/*list_hom.cpp
homogenous list*/
#include <iostream.h>
#include <stddef.h>//de dinh nghia NULL
#include <conio.h>
//class list
struct element {
  element *next;
  void *content;
};
class list {
       element *head;
       element *current;
 public:
       list() {
         head=NULL;current=head;
         }
       ~list();
       void add(void *);
       void first() {
      current=head;}
          void *nextelement() {
           void *adel =NULL;
                if (current !=NULL) {
                  adel =current->content;
                 current=current->next;
                }
               return adel;
         }
        int last() {return (current==NULL);}
};
```

```
list::~list() {
 element *suiv;
 current=head;
 while(current!=NULL)        {
   suiv=current->next;
   delete current;
   current=suiv;
   }
 }
void list::add(void *chose) {
 element *adel= new element;
 adel->next =head;
 adel->content=chose;
 head=adel;
 }
class point
 { int x,y;
 public:
        point (int abs=0,int ord=0){
           x=abs;
           y=ord;
           }
        void display(){
         cout<<" Toa do : "<<x<<" "<<y<<"\n";
         }
 };
class list_points :public list,public point {
  public:
        list_points() {}
        void display();
 };
 void list_points::display() {
```

```
  first();
  while (!last()) {
      point *ptr=(point *)nextelement();
      ptr->display();
   }
  }
void main() {
 clrscr();
 list_points l;
 point a(2,3),b(5,9),c(0,8);
 l.add(&a); l.display();cout<<"-------------\n";
 l.add(&b); l.display();cout<<"-------------\n";
 l.add(&c); l.display();cout<<"-------------\n";
 getch();
 }
```

```
Toa do : 2 3
-------------
 Toa do : 5 9
 Toa do : 2 3
-------------
 Toa do : 0 8
 Toa do : 5 9
 Toa do : 2 3
-------------
```

1.14 Tạo danh sách móc nối không đồng nhất

Phần trước đã xét ví dụ về danh sách móc nối đồng nhất. Về nguyên tắc, các phần tử trong danh sách có thể chứa các thông tin bất kỳ-danh sách không đồng nhất. Vấn đề chỉ phức tạp khi ta hiển thị các thông tin dữ liệu liên quan, vì khi đó ta hoàn toàn không biết được thông tin về kiểu đối tượng khi khai báo tham trỏ (có thể xem lại hàm `display_list()` để thấy rõ hơn).

Khái niệm lớp trừu tượng và hàm ảo cho phép ta xây dựng thêm một lớp dùng làm cơ sở cho các lớp có đối tượng tham gia danh sách móc nối. Trong khai báo của lớp cơ sở này, có một hàm thành phần ảo sẽ được định nghĩa lại trong các lớp dẫn xuất. Các hàm này có chức năng (trong các lớp cơ sở) hiển thị các thông tin liên quan đến đối tượng tương ứng của một lớp.

Đến đây, nhờ khai báo một tham trỏ đến lớp cơ sở này, mọi vấn đề truy nhập đến các hàm hiển thị thông tin về từng thành phần sẽ không còn khó khăn nữa. Sau đây là một chương trình minh hoạ cho nhận xét trên. Trong chương trình này, lớp cơ sở chung cho các lớp dùng làm kiểu dữ liệu cho các thành phần của danh sách chỉ được dùng để khai báo con trỏ để gọi tới các thành phần hàm hiển thị, nên tốt nhất ta là khai báo hàm đó như là hàm ảo thuần tuý và lớp tương ứng sẽ là lớp trừu tượng.

Ví dụ 5.19

```
/*list_hete.cpp
heterogenous list*/
#include <iostream.h>
#include <stddef.h> //chứa giá trị NULL
#include <conio.h>
//lớp trừu tượng
class mere {
 public:
   virtual void display() = 0 ; //hàm ảo thuần tuý
 };
//lớp quản lý danh sách móc nối
struct element //cấu trúc một thành phần của danh sách
{
 element *next;//chỉ đến thành phần đi sau
 void *content; //chỉ đến một đối tượng bất kỳ
};
class list
```

```
{
 element *head;
 element *current;
public:
 list(){
   head = NULL;
   current=head;
   }
 ~list();
 void add(void *);//thêm một thành phần vào đầu danh sách
 void first(){
  current=head;
  }
      /*cho địa chỉ của phần tử hiện tại , xác định thành phần tiếp theo*/
 void *nextelement() {
       void *adnext=NULL;
       if (current !=NULL)
        {
        adnext=current->content;
        current=current->next;
        }
       return adnext;
       }
  /*liệt kê tất cả các phần tử trong danh sách*/
 void display_list();
 int last() {return (current==NULL);}
};
list::~list() {
 element *suiv;
 current = head;
 while(current !=NULL)       {
       suiv =current->next;
```

```
        delete current;
        current=suiv;
        }
 }
void list::add(void * chose) {
 element *adel = new element;
 adel->next = head;
 adel->content = chose;
 head = adel;
 }
void list::display_list() {
 mere *ptr ;//chú ý phải là mere* chu khong phai void *
 first();//in từ đầu danh sách
 while (!last() )     {
        ptr =(mere *) nextelement();
        ptr->display();
        }
 }
//lớp điểm
class point : public mere {
 int x,y;
public:
 point(int abs =0, int ord =0) {x=abs;y=ord;}
 void display()
        {cout << "Toa do : "<<x<<" "<<y<<"\n";}
};
//lớp số phức
class complexe :public mere {
 double reel,imag;
public:
 complexe(double r=0,double i=0) {reel =r; imag=i;}
 void display(){
```

```
        cout<<" So phuc : "<<reel<<" + "<<imag<<"i\n";
        }
};
//chương trình thử
void main() {
 clrscr();
 list l;
 point a(2,3),b(5,9);
 complexe x(4.5,2.7), y(2.35,4.86);
 l.add(&a);
 l.add(&x);
 l.display_list();
 cout<<"----------------\n";
 l.add(&y);
 l.add(&b);
 l.display_list();
 getch();
}
```

```
So phuc : 4.5 + 2.7i
Toa do : 2 3
----------------
Toa do : 5 9
 So phuc : 2.35 + 4.86i
 So phuc : 4.5 + 2.7i
Toa do : 2 3
```

TÓM TẮT

1.15 Ghi nhớ

Thừa kế nâng cao khả năng sử dụng lại của các đoạn mã chương trình.

Người lập trình có thể khai báo lớp mới thừa kế dữ liệu và hàm thành phần tử một lớp cơ sở đã được định nghĩa trước đó. Ta gọi lớp mới là lớp dẫn xuất.

Trong đơn thừa kế, một lớp chỉ có thể có một lớp cơ sở. Trong đa thừa kế cho phép một lớp là dẫn xuất của nhiều lớp.

Lớp dẫn xuất thường bổ sung thêm các thành phần dữ liệu và các hàm thành phần trong định nghĩa, ta nói lớp dẫn xuất cụ thể hơn so với lớp cơ sở và vì vậy thường mô tả một lớp các đối tượng có phạm vi hẹp hơn lớp cơ sở.

Lớp dẫn xuất không có quyền truy nhập đến các thành phần **private** của lớp cơ sở. Tuy nhiên lớp cơ sở có quyền truy xuất đến các thành phần công cộng và được bảo vệ (**protected**).

Hàm thiết lập của lớp dẫn xuất thường tự động gọi các hàm thiết lập của các lớp cơ sở để khởi tạo giá trị cho các thành phần trong lớp cơ sở.

Hàm huỷ bỏ được gọi theo thứ tự ngược lại.

Thuộc tính truy nhập **protected** là mức trung gian giữa thuộc tính **public** và **private**. Chỉ có các hàm thành phần và hàm bạn của lớp cơ sở và lớp dẫn xuất có quyền truy xuất đến các thành phần **protected** của lớp cơ sở.

Có thể định nghĩa lại các thành phần của lớp cơ sở trong lớp dẫn xuất khi thành phần đó không còn phù hợp trong lớp dẫn xuất.

Có thể gán nội dung đối tượng lớp dẫn xuất cho một đối tượng lớp cơ sở. Một con trỏ lớp dẫn xuất có thể chuyển đổi thành con trỏ lớp cơ sở.

Hàm ảo được khai báo với từ khoá **virtual** trong lớp cơ sở.

Các lớp dẫn xuất có thể đưa ra các cài đặt lại cho các hàm ảo của lớp cơ sở nếu muốn, trái lại chúng có thể sử dụng định nghĩa đã nêu trong lớp cơ sở.

Nếu hàm ảo được gọi bằng cách tham chiếu qua tên một đối tượng thì tham chiếu đó được xác định dựa trên lớp của đối tượng tương ứng.

Một lớp có hàm ảo không có định nghĩa (hàm ảo thuần tuý) được gọi là lớp trừu tượng. Các lớp trừu tượng không thể dùng để khai báo các đối tượng nhưng có thể khai báo con trỏ có kiểu lớp trừu tượng.

Tính đa hình là khả năng các đối tượng của các lớp khác nhau có quan hệ thừa kế phản ứng khác nhau đối với cùng một lời gọi hàm thành phần.

Tính đa hình được cài đặt dựa trên hàm ảo.

Khi một yêu cầu được đưa ra thông qua con trỏ lớp cơ sở để tham chiếu đến hàm ảo, C++ lựa chọn hàm thích hợp trong lớp dẫn xuất thích hợp gắn với đối tượng đang được trỏ tới.

Thông qua việc sử dụng hàm ảo và tính đa hình, một lời gọi hàm thành phần có thể có những hành động khác nhau dựa vào kiểu của đối tượng nhận được lời gọi.

1.16 Các lỗi thường gặp

Cho con trỏ lớp dẫn xuất chỉ đến đối tượng lớp cơ sở và gọi tới các hàm thành phần không có trong lớp cơ sở.

Định nghĩa lại trong lớp dẫn xuất một hàm ảo trong lớp cơ sở mà không đảm bảo chắc chắn rằng phiên bản mới của hàm trong lớp dẫn xuất cũng trả về cùng giá trị như phiên bản cũ của hàm.

Khai báo đối tượng của lớp trừu tượng.

Khai báo hàm thiết lập là hàm ảo.

1.17 Một số thói quen lập trình tốt

Khi thừa kế các khả năng không cần thiết trong lớp dẫn xuất, tốt nhất nên định nghĩa lại chúng.

BÀI TẬP

Bài 5.1.

Mô phỏng các thao tác trên một cây nhị phân tìm kiếm với nội dung tại mỗi nút là một số nguyên.

Bài 5.2.

Mô phỏng hoạt động của ngăn xếp, hàng đợi cây nhị phân dưới dạng danh sách móc nối sử dụng mô hình lớp.