



CHƯƠNG 1

ĐỐI TƯỢNG VÀ LỚP (Object & Class)



I. ĐỐI TƯỢNG

- Đối tượng là một khái niệm trong lập trình hướng đối tượng biểu thị sự liên kết giữa dữ liệu và các thủ tục (gọi là phương thức) thao tác trên dữ liệu đó. Ta có công thức sau:

$\text{Đối tượng} = \text{Dữ liệu} + \text{Phương thức}$
--



VD: Mô tả các đối tượng điểm

```
Mô tả đối tượng điểm{  
    // dữ liệu hay thuộc tính  
    Int x,y;  
    // hàm xử lý hay phương thức  
    Void khoitao(int ox, int oy);  
    Void move(int dx, int dy);  
    Void display();  
};
```



II. LỚP

- 1 . Khai báo lớp
- Từ quan điểm của lập trình cấu trúc, lớp là một kiểu dữ liệu tự định nghĩa. Trong lập trình hướng đối tượng, chương trình nguồn được phân bố trong khai báo và định nghĩa của các lớp



Cú pháp khai báo lớp

Class < Tên lớp> {

Private:

<khai báo các thành phần riêng trong từng đối tượng>

Public:

<khai báo các thành phần chung của từng đối tượng>

};

<định nghĩa của các hàm thành phần>

Vd: khai báo và định nghĩa lớp điểm trong mặt phẳng

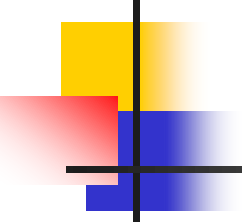
```
#include<iostream.h>
#include<conio.h>
Class point{
// khai báo các thành phần dữ liệu riêng
Private:
    int x,y;
//khai báo các hàm thành phần chung
Public:
Void khoitao(int ox, int oy);
Void move(int dx,int dy);
Void display(); };

```



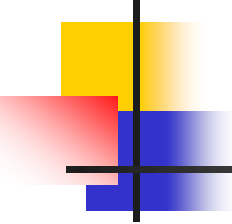
(tiếp theo)

```
//định nghĩa các hàm thành phần
Void point::khoitao(int ox, int oy)
{
    cout<<" Hàm thành phần khởi tạo\n";
    x = ox;
    y = oy;
}
```



```
Void point::move(int dx, int dy)
{ cout<<"Hàm thành phần move \n";
  x = x+dx; y = y+dy; }
```

```
Void point::display()
{ cout<<"Hàm thành phần display\n";
  cout<<"tọa độ : "<<x<<":"<<y<<"\n";
}
```

```
Int main()  
{ point p;  
  p.khoitao(2,4);  
  p.display();  
  p.move(1,3);  
  p.display();  
  Getch();  
}
```

Định nghĩa hàm thành phần bên ngoài khái báo lớp

<Tên kiểu giá trị trả về> <Tên lớp> :: <Tên hàm> (ds tham số)

{
Nội dung
}

Dấu "::" gọi là toán tử phạm vi

Nhờ có toán tử phạm vi ta có thể phân biệt được hàm thành phần với các hàm tự do cùng tên

Vd: Void point::khoitao(int ox, int oy)
Void point::move(int dx, int dy)



Gọi hàm thành phần của lớp

- Gọi hàm thành phần của lớp từ một đối tượng chính là truyền thông điệp cho hàm thành phần đó. Cú pháp như sau:
- `<Tên đối tượng>.<Tên hàm thành phần>(ds các tham số nếu có);`
- Vd: `p.khoitao(2,4);`
`p.display();`
`p.move(1,3);`



Tạo đối tượng

- <Tên lớp> <Tên đối tượng>;
- Vd: point p;
- point b,c,t;

Nhận xét:

Là khai báo biến với một kiểu dữ liệu mới
mà ta vừa định nghĩa



Các thành phần dữ liệu

- Cú pháp khai báo các thành phần dữ liệu:
- <Tên kiểu dữ liệu> <Tên thành phần>;
- Vd: int x,y;
- Float a,b,c;
- Thực chất là khai báo biến



Các hàm thành phần

- Hàm được khai báo trong định nghĩa của lớp được gọi là hàm thành phần hay phương thức của lớp. Các hàm thành phần có thể truy nhập đến các thành phần dữ liệu (kể cả thành phần private) và các hàm thành phần khác trong lớp.
- Vd : `Void khoitao(int ox, int oy);`
`Void move(int dx,int dy);`
`Void display();`



Các hàm thành phần

```
Void point::khoitao(int ox, int oy)
{ cout<<" Hàm thành phần khởi tạo\n";
  x = ox; y = oy; } // truy xuất đến các thành phần dữ liệu riêng của
lớp
```

```
Void point::move(int dx, int dy)
{ cout<<"Hàm thành phần move \n";
  x = x+dx; y = y+dy; } // truy xuất đến các thành phần dữ
liệu riêng của lớp
```



Phạm vi lớp

- Phạm vi chỉ ra phần chương trình trong đó có thể truy xuất đến một đối tượng nào đó.
- Phạm vi lớp: Tất cả các thành phần của một lớp sẽ được coi là thuộc phạm vi lớp.
- Ví dụ 3.4



Từ khóa xác định thuộc tính truy xuất

- **Private:**

Những thành phần được liệt kê trong phần private chỉ được truy xuất bên trong phạm vi lớp, bởi chúng thuộc sở hữu riêng của lớp.

- **Public:**

Những thành phần được liệt kê trong phần public đều có thể được truy xuất trong bất kỳ hàm nào, bởi chúng thuộc sở hữu chung của mọi thành phần trong chương trình



Từ khóa xác định thuộc tính truy xuất

- Các thành phần trong một lớp có thể được sắp xếp một cách tùy ý.
- Nếu thành phần private ở trước public thì không cần ghi từ khóa private (C++ mặt định trước public là private)
- Ví dụ 3.4 (Trang 51 – 52)



So sánh hai đoạn chương trình sau:

```
Struct point {  
  Int x, y;  
  Void khoitao(int ox, int oy);  
  Void move(int dx, int dy);  
  Void display();  
};
```

```
class point {  
  Public:  
  Int x, y;  
  Void khoitao(int ox, int oy);  
  Void move(int dx, int dy);  
  Void display();  
};
```



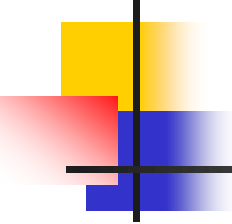
Nhận xét

- Nếu tất cả các thành phần của một lớp là public, lớp sẽ hoàn toàn tương đương với một cấu trúc, không có phạm vi lớp.



Định nghĩa chồng các hàm thành phần

- Các hàm thành phần có thể có trùng tên nhưng phải khác kiểu giá trị trả về hoặc danh sách tham số.
- Hàm thành phần được phép gọi tới các hàm thành phần khác, thậm chí trùng tên.
- Ví dụ 3.5 (trang 54 – 55)



Các tham số với giá trị ngầm định

- Lời gọi hàm thành phần có thể sử dụng giá trị ngầm định cho các tham số. Giá trị ngầm định này sẽ được khai báo trong định nghĩa hàm thành phần.
- Ví dụ 3.6 (Trang 56 – 57)



Ví dụ về hàm thành phần có tham số với giá trị ngầm định

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

```
#include<conio.h>
```

```
Class point{
```

```
int x,y;
```

```
Public:
```

```
Void khoitao(int ox=0, int oy=0);
```

```
//hàm thành phần có tham số với giá trị ngầm định
```

```
Void move(int dx,int dy);
```

```
Void display(); };
```



PHÉP GÁN CÁC ĐỐI TƯỢNG

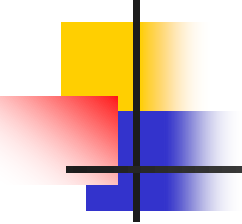
- Có thể thực hiện phép gán giữa hai đối tượng cùng kiểu.

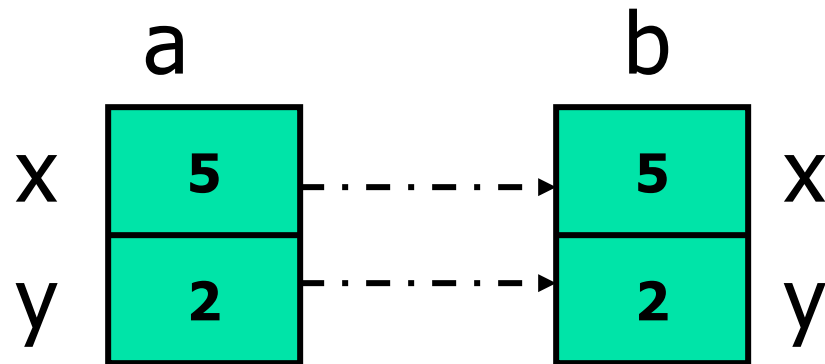
- Ví dụ: với lớp point khai báo ở trên:

Point a,b;

a.khoitao(5,2);

b=a;

- 
- Về thực chất đó là việc sao chép giá trị các thành phần dữ liệu (x,y) từ đối tượng a sang đối tượng b tương ứng từng đôi một.





HÀM THIẾT LẬP (constructor)

- Chức năng của hàm thiết lập:

Hàm thiết lập là một **hàm thành phần đặc biệt** không thể thiếu được trong một lớp. **Nó được gọi tự động mỗi khi có một đối tượng được khai báo**. Chức năng của hàm thiết lập là khởi tạo các giá trị thành phần dữ liệu của đối tượng, xin cấp phát bộ nhớ cho các thành phần dữ liệu động.

Ví dụ 3.7 (Trang 61)



CÁC ĐẶC ĐIỂM CỦA HÀM THIẾT LẬP

1. Hàm thiết lập có cùng tên với tên lớp
2. Hàm thiết lập phải có thuộc tính public
3. Hàm thiết lập không có giá trị trả về. Và không cần khai báo void
4. Có thể có nhiều Hàm thiết lập trong cùng lớp (chồng các hàm thiết lập)
5. Khi một lớp có nhiều hàm thiết lập, việc tạo các đối tượng phải kèm theo các tham số phù hợp

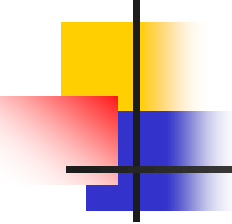
- 
-
- Xem xét ví dụ 3.8 và 3.9 từ đó rút ra nhận xét



HÀM HỦY BỎ (Destructor)

- Ngược với hàm thiết lập, hàm hủy bỏ được gọi khi đối tượng tương ứng bị xóa khỏi bộ nhớ.
- Xét ví dụ 3.12 (trang 70)

CÁC ĐẶC ĐIỂM CỦA HÀM HỦY BỎ



1. Tên của hàm hủy bỏ bắt đầu bằng dấu ~ theo sau là tên của lớp tương ứng.
2. Hàm hủy bỏ phải có thuộc tính public
3. Hàm hủy bỏ không có tham số, mỗi lớp chỉ có một hàm hủy bỏ.
4. Hàm hủy bỏ không có giá trị trả về.



HÀM BẠN VÀ LỚP BẠN

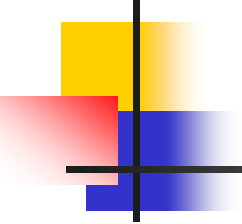
- HÀM BẠN: (trang 89)

Hàm bạn không phải là hàm thành phần của lớp nhưng nó có quyền truy cập vào các thành phần riêng của lớp.



Các kiểu hàm bạn

- Hàm tự do là bạn của một lớp
- Khai báo:
friend <kiểu trả về> <tên hàm> (tham số)
Vd : friend int coincide(point, point)
Xét ví dụ 3.18 (trang 91)

- 
-
- Hàm thành phần của lớp là bạn của lớp khác
 - Khai báo:
friend <kiểu trả về> <tên lớp> :: <tên hàm> (ds các tham số)



Ví dụ:

- Giả sử có hai lớp A và B, trong B có một hàm thành phần khai báo như sau:

```
int f(char, A);
```

- Nếu f có nhu cầu truy xuất vào các thành phần riêng của A thì f cần phải được khai báo là bạn của A ở trong lớp A bằng câu lệnh:

```
friend int B:: f(char, A);
```



Ví dụ:

```
class A;  
class B{  
    int f(char, A); // là hàm thành phần của lớp B  
};  
class A {  
    friend int B::f(char, A);  
    // f có quyền truy cập vào các thành phần riêng của lớp A  
};  
//Định nghĩa hàm f  
Int B:: f(char..., A....)  
{ .....}
```



CHƯƠNG 2

ĐỊNH NGHĨA TOÁN TỬ TRÊN LỚP (Class operators)



Hàm toán tử

- Trong C++ cho phép ta định nghĩa chồng các **hàm toán tử** (operator function) nhờ đó ta có thể định nghĩa các phép toán một ngôi hoặc hai ngôi trên các đối tượng.
- Hàm toán tử có thể dùng như là một **hàm thành phần** của lớp hoặc là **hàm bạn** của lớp.



Cú pháp hàm toán tử:

- operator<tên phép toán>
- operator+ định nghĩa phép +
- operator- định nghĩa phép -
- operator* định nghĩa phép *
- operator/ định nghĩa phép /
- operator+= định nghĩa phép +=
- operator!= định nghĩa phép so sánh khác



Hàm toán tử là hàm thành phần của lớp

- Xét ví dụ 4.1 (trang 110)



Nhận xét:

- $c = a + b$ thực ra là $c = a.operator+(b)$
- Cách viết $c = a + b$ chỉ là một quy ước của chương trình dịch cho phép người sử dụng viết gọn lại, nhờ đó cảm thấy tự nhiên hơn.
- Hàm toán tử phải có thuộc tính public



Nhận xét:

- Trong lời gọi $a.operator+(b)$, a đóng vai trò là tham số ngầm định của hàm thành phần và b là tham số tường minh.
- Số tham số tường minh cho hàm toán tử thành phần luôn ít hơn số ngôi của phép toán là 1 vì có một tham số ngầm định là đối tượng gọi hàm toán tử.
- Với phép toán một ngôi thì hàm toán tử thành phần không có tham số tường minh



Nhận xét:

- Trong ví dụ 4.1 lệnh $c = 3+a$ và lệnh $c = a+3$ có thực hiện được không? Vì sao?.
- $3+a \Leftrightarrow 3.operator(a)$. 3 không phải là đối tượng của lớp **complex** nên không thể gọi hàm thành phần của lớp.
- $a+3 \Leftrightarrow a.operator(3)$. 3 không phải là đối tượng nên không truyền được cho hàm (Tham số của hàm thành phần `operator+` là 1 đối tượng)



Hàm toán tử là hàm bạn của lớp

- Lúc này hàm toán tử là một hàm tự do
- Cú pháp:
- `friend <kiểu trả về>operator<phép toán>(tham số)`
- Trong ví dụ 4.1 định nghĩa 2 phép toán $3+a$ và $a+3$.



Một số ví dụ tiêu biểu

- Ví dụ 4.6 (trang 126)
- Ví dụ 4.9 (trang 135)
- Ví dụ 4.10 (trang 137)



CHƯƠNG 3

KỸ THUẬT THỪA KẾ (Inheritance)



Nội dung chính

- Cài đặt sự thừa kế.
- Sử dụng các thành phần của lớp cơ sở.
- Định nghĩa lại các hàm thành phần.
- Truyền thông tin giữa các hàm thiết lập của lớp dẫn xuất và lớp cơ sở.
- Các loại dẫn xuất khác nhau và sự thay đổi của các thành phần lớp cơ sở.
- Sự tương thích giữa các đối tượng của lớp dẫn xuất và lớp cơ sở.
- Hàm ảo và tính đa hình
- Hàm ảo thuần túy và lớp cơ sở trừu tượng

Giới thiệu kỹ thuật thừa kế

- 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.

LỚP A (cơ sở)

Thuộc tính:

x,y,z

Phương thức:

f1(),f2()

LỚP B (dẫn xuất)

Thuộc tính:

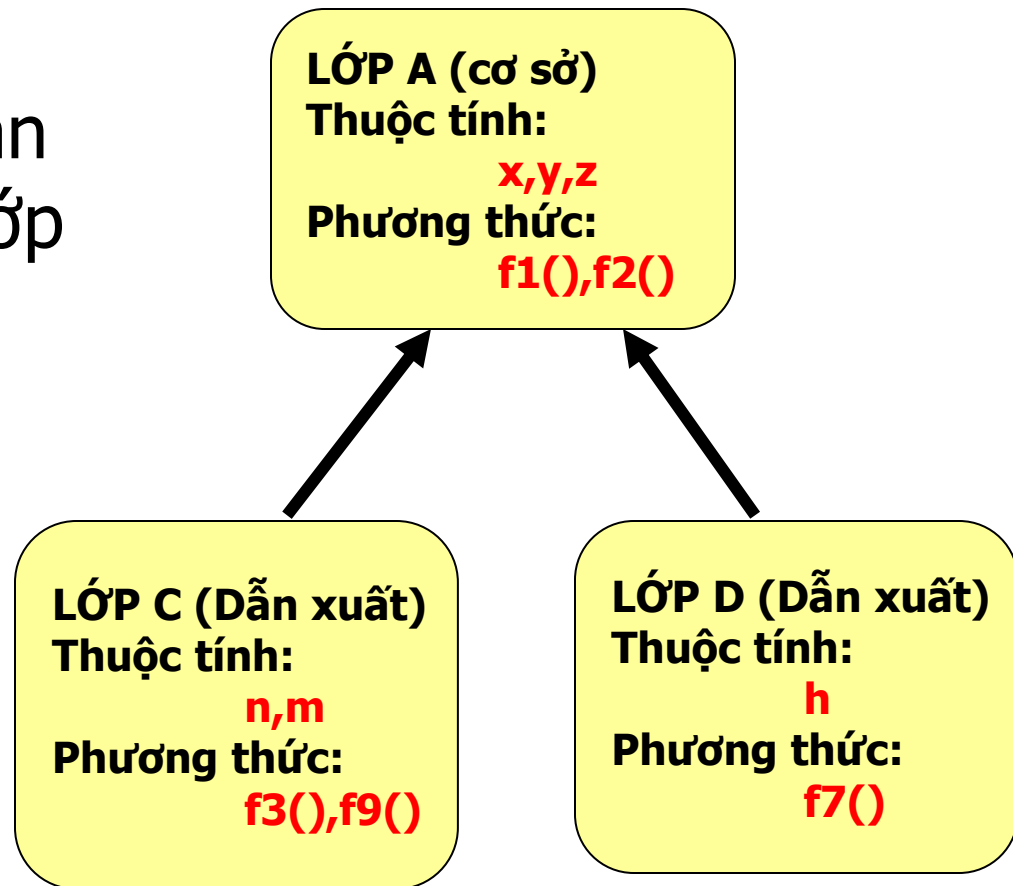
u,v

Phương thức:

f4(),f5()

Giới thiệu kỹ thuật thừa kế (tt)

- Thừa kế cho phép nhiều lớp có thể dẫn xuất từ cùng một lớp cơ sở.



Giới thiệu kỹ thuật thừa kế (tt)

- Một lớp dẫn xuất có thể lại là lớp cơ sở cho các lớp dẫn xuất khác.

LỚP A (cơ sở)
Thuộc tính:

x,y,z

Phương thức:

f1(),f2()

LỚP B (dẫn xuất từ A)
Thuộc tính:

u,v

Phương thức:

f4(),f5()

LỚP G (dẫn xuất từ B)
Thuộc tính:

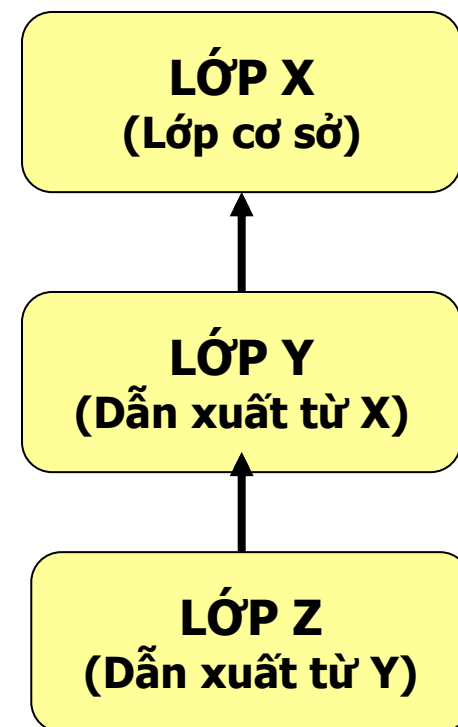
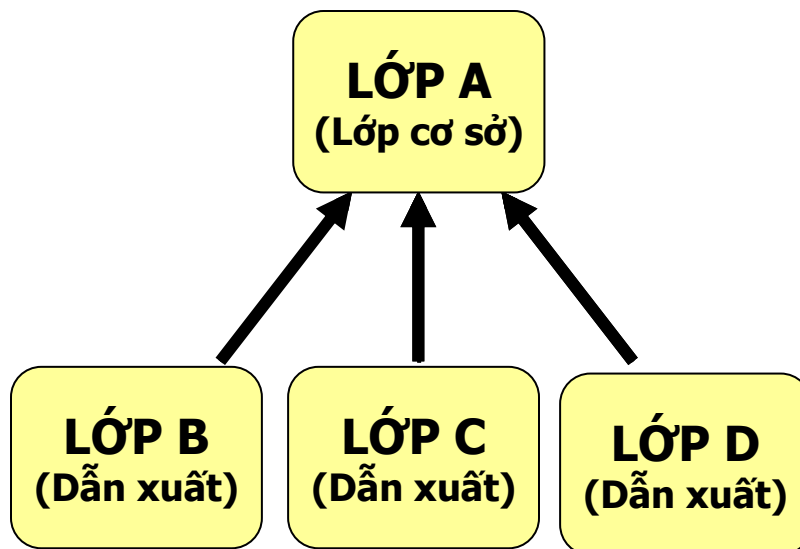
k

Phương thức:

f7(),f8()

Thừa kế

- Đơn thừa kế là lớp dẫn xuất chỉ dẫn xuất từ một lớp cơ sở duy nhất.



Kỹ thuật cài đặt Lớp thừa kế

```
■ class A {  
    int x,y,z;  
    public:  
    A(int x, int y, int y); }  
■ class B : public class A {  
    char u,v;  
    public:  
    B(int x, int y, int z, char u, char v)  
}
```

LỚP A (cơ sở)

Thuộc tính:

x,y,z

Phương thức:

f1(),f2()

LỚP B (dẫn xuất)

Thuộc tính:

u,v

Phương thức:

f4(),f5()



Xét ví dụ sau:

1. Xây dựng lớp điểm (point) trong mặt phẳng có tọa độ (x,y).
2. Xây dựng lớp điểm màu (color point) thừa kế từ lớp điểm có thêm thuộc tính màu (color).

LỚP POINT

Thuộc tính : x,y

Phương thức:

Point();move();

display(); ~Point();

LỚP COLORPOINT

Thuộc tính : color

Phương thức:

colorpoint();updatecolor();

display(); ~colorpoint();





Cài đặt ví dụ

```
#include<iostream.h>
#include<conio.h>
Class point{ // khai báo lớp điểm
Private:
    int x,y;
Public:
    point(int ox=0, int oy=0);
    Void move(int dx,int dy);
    Void display();
    ~point(); };

```



```
/* định nghĩa các hàm thành phần của  
lớp point */
```

```
point::point(int ox, int oy) {  
    cout<<" Hàm thiết lập của point \n";  
    x = ox; y = oy; }
```

```
Void point::move(int dx, int dy)  
{ cout<<"Hàm move của point \n";  
    x = x+dx; y = y+dy; }
```

```
Void point::display()  
{ cout<<"Hàm display của point \n";  
    cout<<"tọa độ : "<<x<<": "<<y<<"\n";}
```



//Lớp colorpoint thừa kế từ point

Class colorpoint : **public point**{

 char color;

Public:

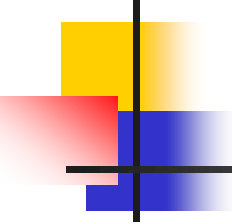
 colorpoint(int ox=0,int oy=0,char cl='b');

 Void updatecolor(char cl);

 Void display();

 ~colorpoint();

};



```
/* định nghĩa các hàm thành phần của  
lớp coloredpoint */
```

```
Colorpoint::colorpoint(int ox,int oy,char cl) :
```

```
    point(ox,oy) {color = cl; } //gọi hàm dựng của lớp point
```

```
Void Colorpoint::updatecolor(char cl)
```

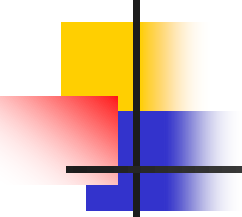
```
{ color = cl; }
```

```
Void Colorpoint::display()
```

```
{cout<<"Hàm display của colorpoint \n";
```

```
    point::display(); //gọi tới hàm cùng tên trong lớp cơ sở
```

```
    cout<<"Màu : "<<color; }
```

```
Int main()
{ point b(7,5); colorpoint a(2,4,'R');
b.display(); //goi display của lớp point => b.point::display()
a.display(); //goi display của lớp colorpoint => a.colorpoint::display()
Cout<<" chỉ hiển thị tọa độ của a";
a.point::display(); // gọi display trong lớp point
a.move(3,3); // lớp dẫn xuất sử dụng phương thức của lớp cơ sở
a.display(); // lớp dẫn xuất sử dụng phương thức của chính nó
a.updatecolor('b');
a.display();
// b.updatecolor('g'); có chạy không ?
Getch(); }
```



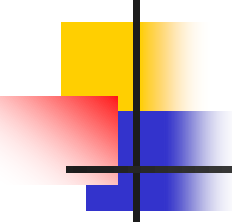
Đơn thừa kế

- Xét ví dụ 5.1 (trang 165)
- Ví dụ này đề cập đến khá nhiều khía cạnh, liên quan đến kỹ thuật cài đặt tính thừa kế trong C++ đó là:
 1. Truy nhập các thành phần của lớp cơ sở từ lớp dẫn xuất.
 2. Định nghĩa lại (đề) các hàm thành phần lớp cơ sở trong lớp dẫn xuất.
 3. Truyền thông tin giữa các hàm thiết lập



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 lớp dẫn xuất.
- Phạm vi lớp chỉ mở rộng cho bạn bè, mà không được mở rộng đến các lớp con cháu.



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

- Cần phân biệt việc định nghĩa lại một hàm thành phần khác với định nghĩa chồng hàm thành phần.
- Khái niệm định nghĩa lại chỉ được xét đến 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 ở trong lớp cơ sở.



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

- Trong lập trình hướng đối tượng, một đối tượng của lớp dẫn xuất có thể “thay thế” một đối tượng của lớp cơ sở.
- Ví dụ 1: Ta có **lớp trái cam** thừa kế từ **lớp trái cây**. Vậy ta có thể nói trái cam là trái cây \Leftrightarrow trái cây = trái cam.
- Ví dụ 2: `point p , colorpoint pc(2,3,'r');`
lúc này ta có thể viết `p = pc;`



Hàm thiết lập trong lớp dẫn xuất

- 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.



Hàm thiết lập trong lớp dẫn xuất

- Cơ chế trên được gọi một cách ngầm định, ta 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)



Các kiểu dẫn xuất khác nhau

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

Từ khóa	Kiểu dẫn xuất
public	dẫn xuất public
private	dẫn xuất private
protected	dẫn xuất protected



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.



Dẫn xuất private

- Trong dẫn xuất private, 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 mới mà chỉ định nghĩa lại các phương thức đã có trong lớp cơ sở.



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



Diễn giải từ viết tắt

Từ viết tắt	Diễn giải
TTĐ	Trạng thái đầu
TTM	Trạng thái mới
TN FMA	Truy cập bởi các hàm thành phần, hàm bạn
TN NSD	Truy cập bởi người sử dụng
Pro	Protected
Pub	Public
Pri	Private
C	Có
K	không



Hàm ảo và tính đa hình

- Xét ví dụ 5.7 (trang 183)



Nhận xét ví dụ 5.7

- Trong định nghĩa của lớp point, khi khai báo hàm thành phần `display()` phía trước có từ khóa **virtual** để chỉ định đây là một hàm ảo.
- Tuy nhiên vai trò của hàm ảo trong trường hợp này là không rõ ràng



Hàm ảo và tính đa hình

- Hàm thành phần có từ khóa **virtual** đặt trước kiểu dữ liệu thì được gọi là hàm ảo
- Xét ví dụ sau đây và nêu vai trò của hàm ảo trong sơ đồ thừa kế.



//khai báo lớp point

#include<iostream.h>

#include<conio.h>

Class point{

Private:

int x,y;

Public:

point(int ox=0, int oy=0);

Void move(int dx,int dy);

virtual Void hamao(); // hàm ảo

Void display(); };



```
/* định nghĩa các hàm thành phần của  
lớp point */
```

```
point::point(int ox, int oy)
```

```
{ x = ox; y = oy; }
```

```
Void point::move(int dx, int dy)
```

```
{ x = x+dx; y = y+dy; }
```

```
Void point::hamao()
```

```
{ cout<<"Đây là lớp point: \n"; }
```

```
Void point::display()
```

```
{ hamao() ;
```

```
cout<<"tọa độ : "<<x<<":"<<y<<"\n"; }
```



//khai báo Lớp colorpoint thừa kế từ point

Class colorpoint : **public point**{

 char color;

Public:

 colorpoint(int ox=0,int oy=0,char cl='b')

 Void updatecolor(char cl);

void hamao();

 Void display();

};



```
/* định nghĩa các hàm thành phần của  
lớp coloredpoint */
```

```
Colorpoint::colorpoint(int ox,int oy,char cl) :  
    point(ox,oy) {color = cl; }
```

```
Void Colorpoint::updatecolor(char cl)  
    { color = cl; }
```

```
Void Colopoint::hamao() // định nghĩa lại hàm ảo  
    { cout<<"Đây là lớp colorpoint: \n"; }
```

```
Void Colorpoint::display() // định nghĩa lại display()  
{ point::display(); //gọi tới hàm cùng tên trong lớp cơ sở  
  cout<<"Màu : "<<color; }
```



```
/* khai báo lớp threecolorpoint thừa kế từ  
lớp colorpoint */
```

```
Class threecolorpoint : public colorpoint {  
    int z;
```

```
Public:
```

```
    threecolorpoint(int ox=0,int oy=0,char  
                    cl='b', int oz=0);
```

```
    Void update_oz(int dz);
```

```
    void hamao();
```

```
    Void display();
```

```
};
```



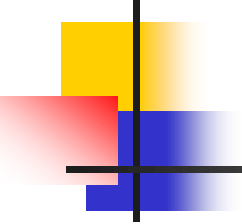
```
/* định nghĩa các hàm thành phần của  
lớp threecoloredpoint */
```

```
threecolorpoint::threecolorpoint(int ox,int oy,char  
cl,int oz) : colorpoint(ox,oy,cl) { z = oz; }
```

```
Void threecolorpoint::update_oz(int dz)  
{ z = z + dz; }
```

```
Void threecolorpoint::hamao() //định nghĩa lại hàm ảo  
{ cout<<"Đây là lớp threecolorpoint: \n"; }
```

```
Void threecolorpoint::display() //định nghĩa lại display()  
{ colorpoint::display(); //gọi tới hàm cùng tên trong lớp cha  
cout<<"z : "<<z; }
```



```
Int main()
{ point a(7,5); colorpoint b(2,4,'R'); threecolorpoint c(3,7,'G',5);
a.display();
b.display();
c.display();
Cout<<" chỉ hiển thị tọa độ của C : ";
c.point::display(); // gọi display trong lớp point
c.move(3,3); // lớp dẫn xuất sử dụng phương thức của lớp cơ sở
c.display(); // lớp dẫn xuất sử dụng phương thức của chính nó
c.updatecolor('b');
c.display();
Getch();
}
```



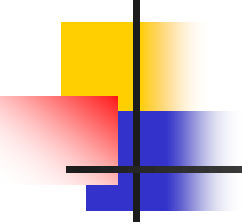
Hàm ảo thuần túy

- Hàm ảo thuần túy là hàm không có phần định nghĩa. Định nghĩa một hàm ảo thuần túy được viết như sau:
- Class ABC {
 public :
 virtual void fun() = 0;
 //hàm ảo thuần túy
};



lớp trừu tượng

- Lớp có ít nhất một hàm thành phần ảo thuần túy được gọi là lớp trừu tượng.
- Lớp trừu tượng không có đối tượng cụ thể.
- Lớp trừu tượng là tổng quát hóa của các lớp thừa kế nó, và ngược lại các lớp dẫn xuất là sự cụ thể hóa của lớp trừu tượng.

- 
-
- Một hàm ảo thuần túy 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.



Chương 6

KHUÔN HÌNH (Template)



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;**



Khuôn hình hàm

- 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 với kiểu dữ liệu `int` như sau:
- `int min (int a, int b)`
`{ if (a < b) return a;`
`else return b; }`



Tạo một khuôn hình hàm

- Giả sử, ta lại phải viết định nghĩa hàm min cho kiểu double, float, char . . .
- `float min (float a, float b)`
 { if (a < b) return a;
 else return b; }



Nhận xét

- Các hàm hoàn toàn tương tự nhau, chỉ khác kiểu dữ liệu
- Vậy ta có thể viết một hàm để sử dụng cho tất cả các kiểu dữ liệu được không?



Tạo một khuôn hình hàm

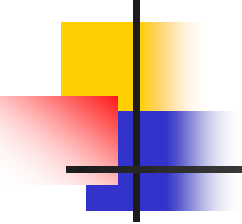
```
#include<iostream.h>
```

```
#include<conio.h>
```

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

```
{ if ( a < b ) return a ;
```

```
    else return b; }
```

- 
-
- `Template <class T> T min (T a, T b)`
 - Trong đó
 - `Template <class T>` là một khuôn hình với tham số kiểu `T` // `T` là kiểu dữ liệu tượng trưng
 - `T min (T a, T b)` là một hàm với 2 tham số hình thức kiểu `T` và có giá trị trả về cũng là kiểu `T`



Sử dụng khuôn hình hàm

```
Int main ()
{ int n = 4, p = 12;
  float x=3.75, y=3.756;
  cout<<" min (n, p) = "<<min(n,p)<<"\n";
  cout<<" min (x, y) = "<<min(x,y)<<"\n";
  Getch();
}
```



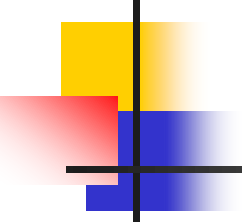
Khuôn hình lớp

- 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.



Định nghĩa khuôn hình lớp

```
Template <class T> class point {  
    private : T x, Ty;  
    public :  
        point( T ox, T oy)  
    void display();  
    . . . };
```

- 
-
- Cũng giống như khuôn hình hàm, tập hợp Template `<class T>` xác định rằng đó là một khuôn hình trong đó có một tham số kiểu T.
 - Class T : T là một tham số hình thức đại diện cho một kiểu dữ liệu nào đó.