LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VỚI TURBO C++

Chapter 1. Introduction

Giới thiệu

Mục đích

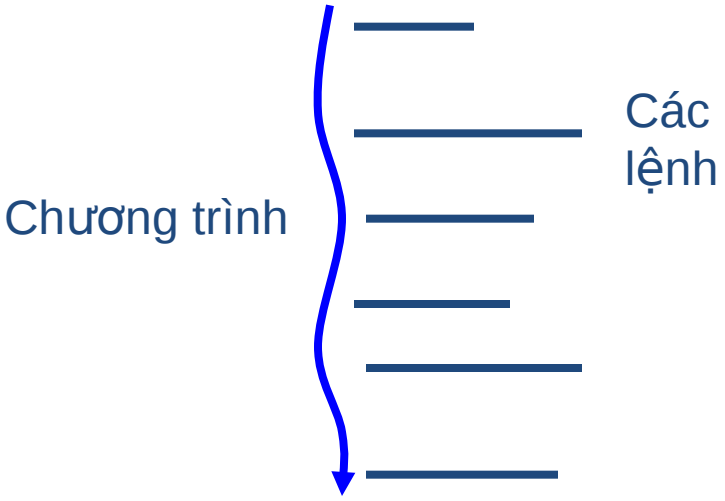
- Giới thiệu cơ bản về phương pháp lập trình tuyến tính, lập trình cấu trúc, lập trình hướng đối tượng.
- Hướng tiếp cận lập trình hướng đối tượng.
- Các khái niệm của lập trình hướng đối tượng.

Nội dung

- Phương pháp lập trình tuyến tính.
- Phương pháp lập trình cấu trúc.
- Phương pháp lập trình hướng đối tượng.
- Bài toán quan hệ gia đình.
- Một số khái niệm của lập trình hướng đối tượng.

Lập trình tuyến tính

- Phát triển vào những ngày đầu của ngành khoa học máy tính.
- Chương trình gồm nhiều lệnh viết theo trật tự tuyến tính.



Lập trình tuyến tính *(tiếp)*

- **Trong chương trình không có thủ tục:**
  - Chương trình dài vì lệnh được chép lặp lại khi nó được thực hiện nhiều lần trong chương trình.
  - Dữ liệu là dữ liệu toàn cục.
  - Chỉ phù hợp với các chương trình nhỏ, không phù hợp với những chương trình lớn

Lập trình cấu trúc

- Phát triển mạnh vào thập kỷ 70.
- Chương trình được chia nhỏ thành các chương trình con.
- Các chương trình con được thiết kế càng độc lập các tốt.
- Mỗi chương trình con tự quản lý biến địa phương của nó. Không cho phép ai ngoài phạm vi chương trình con được truy nhập.

Lập trình cấu trúc *(tiếp)*

- Chương trình = Dữ liệu + Giải thuật.
- Trừu tượng hoá chức năng (abstraction) được đưa vào trong lập trình cấu trúc. Nghĩa là chỉ cần biết 1 chương trình con làm được 1 công việc gì là đủ còn làm thế nào mà chương trình con đó được thực hiện thì không quan trọng. Ví dụ: x = sqrt(x) thì sqrt(x) là sự trừu tượng hoá chức năng tính căn bậc 2 của x.

Lập trình cấu trúc *(tiếp)*

- Tóm lại:
  - Phương pháp lập trình cấu trúc có tính trong sáng do đó nó đã tỏ ra hiệu quả khi triển khai và bảo trì một chương trình.
  - Tuy nhiên khi phát triển các phần mềm lớn thì nó bắt đầu xuất hiện một số nhược điểm:
    - —Trong một chương trình, cấu trúc dữ liệu đóng vai trò quan trọng, khi thay đổi dữ liệu thì phải điều chỉnh nhiều modul có liên quan.
    - —Khi một nhóm người phát triển, làm giảm tính modul hoá công việc.

Lập trình hướng đối tượng

- Khái niệm Hướng đối tượng được xây dựng trên nền tảng của lập trình cấu trúc và sự trừu tượng hoá dữ liệu (data abstraction).
- Sự trừu tượng hoá dữ liệu nghĩa là các cấu trúc dữ liệu và các phần tử có thể được sử dụng mà không cần để ý đến chi tiết cụ thể đã xây dựng nên cấu trúc dữ liệu đó.

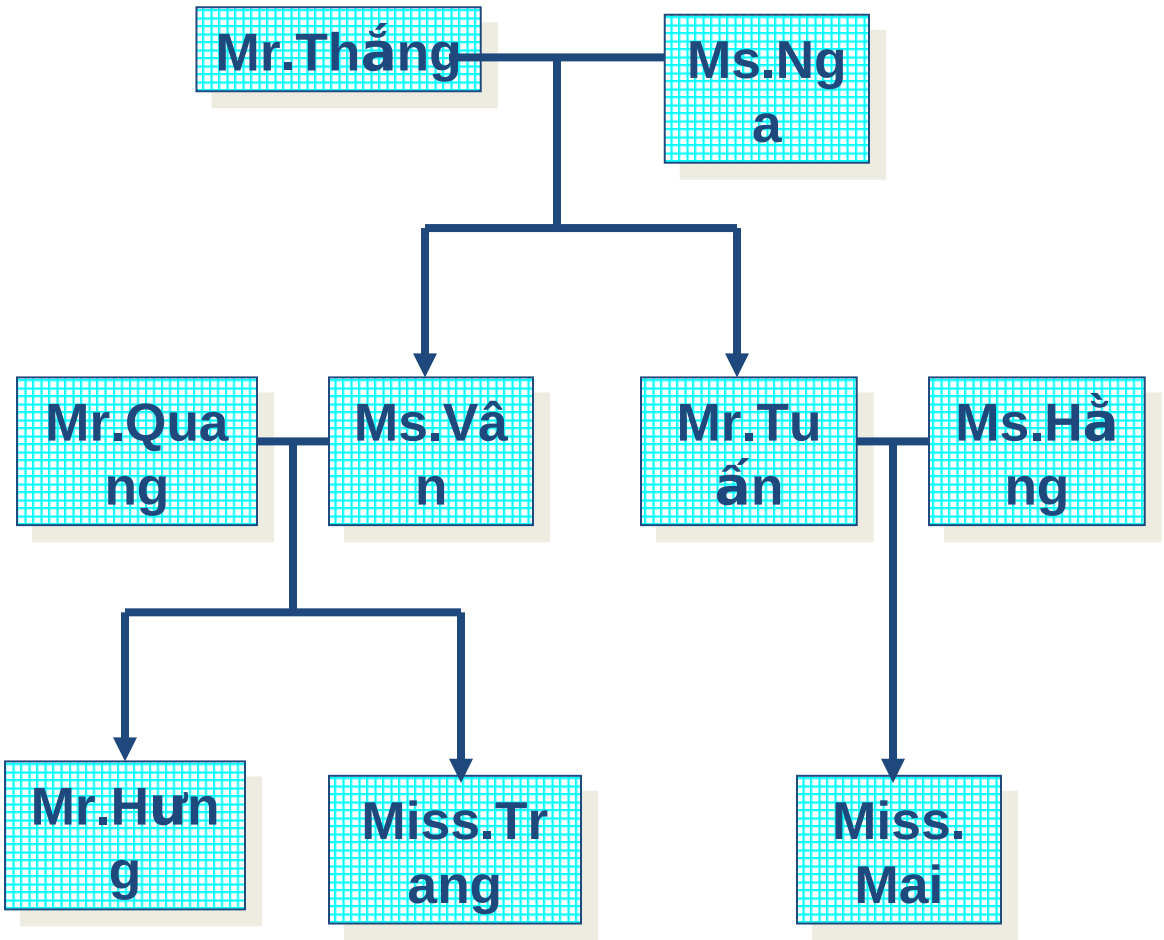
Lập trình hướng đối tượng *(tiếp)*

- Điểm cơ bản của phương pháp lập trình hướng đối tượng là thiết kế đối tượng xoay quanh dữ liệu của nó, nghĩa là các thao tác xử lý của đối tượng liền với dữ liệu của nó.
- Sự đóng gói dữ liệu và các hàm xử lý vào một khối gọi là một đối tượng.
- Sự gắn kết dữ liệu và các hàm xử lý vào một đối tượng làm cho tính modul hoá cao hơn.

Bài toán quan hệ gia đình

- Trong xã hội, mọi người đều có một gia đình trong đó tồn tại nhiều mối quan hệ gia đình khá phức tạp như ông, bà, cha, mẹ...
- Thông thường để biểu diễn mối quan hệ này người ta thường biểu diễn bằng một sơ đồ cây.
- Xét quan hệ trong 1 gia đình với ba thế hệ như sau:

Bài toán quan hệ gia đình *(tiếp)*

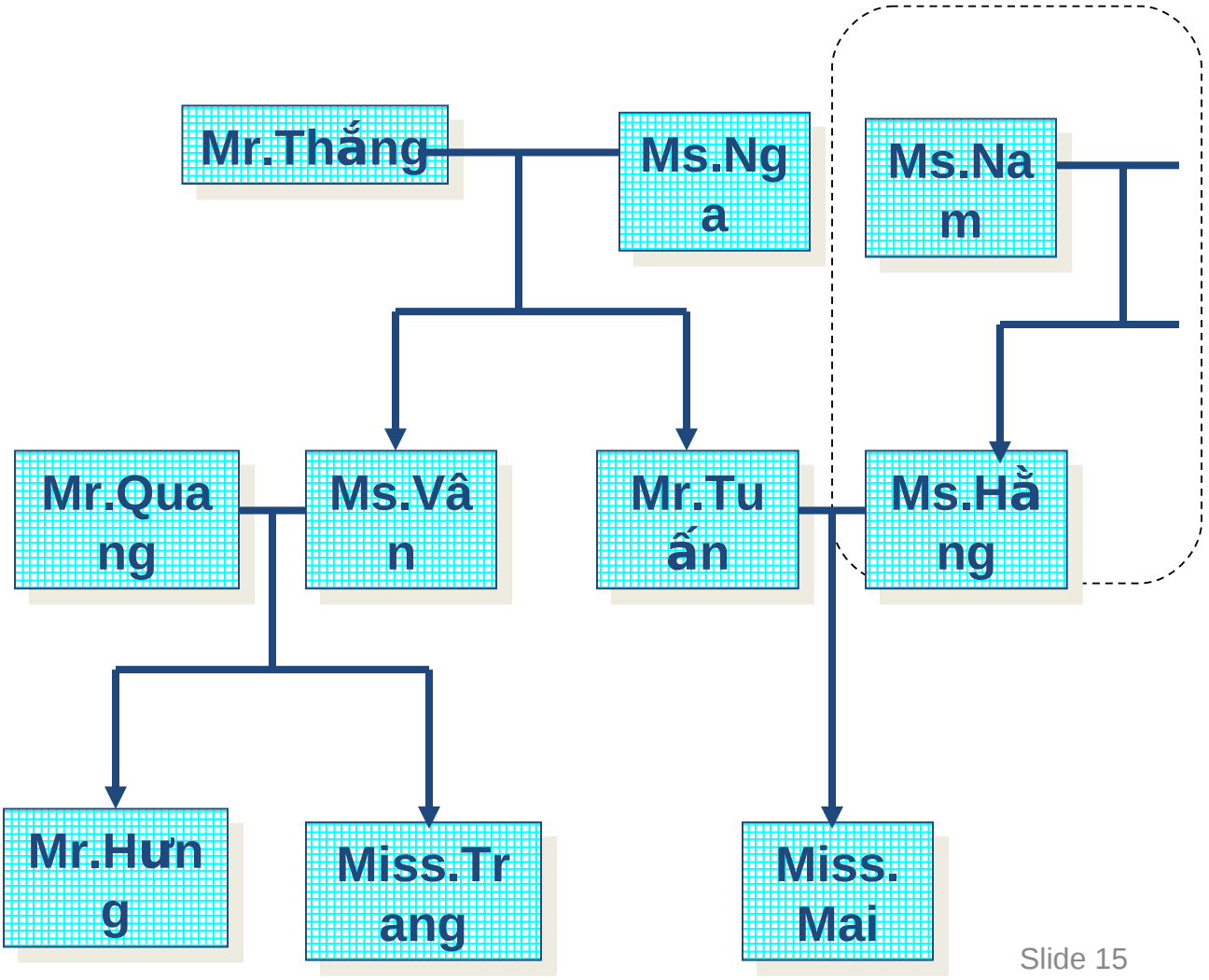


Bài toán quan hệ gia đình *(tiếp)*

- Tiếp cận theo phương pháp lập trình cấu trúc:
  - Phải xây dựng cấu trúc dữ liệu cây thể hiện được cây quan hệ trên.
  - Phải xây dựng giải thuật cập nhật thông tin cho các nút của cây.
  - Phải xây dựng giải thuật tìm kiếm quan hệ của 2 nút trên cây.
  - Các giải thuật này tương đối phức tạp.

Bài toán quan hệ gia đình *(tiếp)*

– Nếu mở rộng cây quan hệ gia đình như có thêm các mối thông gia thì phải xây dựng lại chương trình.

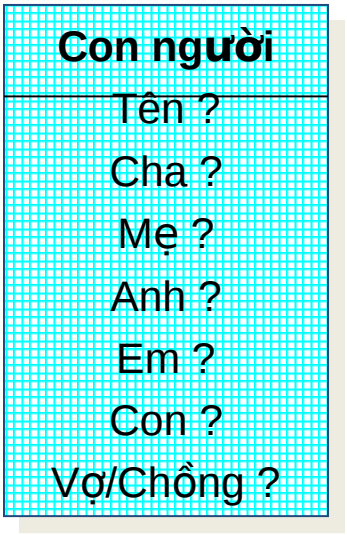


Bài toán quan hệ gia đình *(tiếp)*

- Tiếp cận theo lập trình hướng đối tượng:
  - Bài toán được xem xét dưới góc độ quản lý các tập đối tượng **Con người**.
  - Để biết mối quan hệ gia đình của mỗi người cần thể hiện một số mối thuộc tính cơ bản như: Họ tên, tên cha, tên mẹ, tên anh, tên em, tên con, tên vợ/chồng của cá thể đó.

Bài toán quan hệ gia đình *(tiếp)*

- Một đối tượng con người có thể mô tả như sau:



- Nếu chỉ xét như vậy thì giống với một cấu trúc bản ghi trong lập trình cấu trúc.

Bài toán quan hệ gia đình *(tiếp)*

- Vấn đề của phương pháp lập trình hướng đối tượng là xem xét các mối quan hệ gia đình được hình thành 1 cách tự nhiên do các sự kiện cụ thể tạo ra.
- Hai sự kiện chính tác động lên mối quan hệ gia đình:
  - Sự hôn nhân
  - Sự sinh con.

Bài toán quan hệ gia đình *(tiếp)*

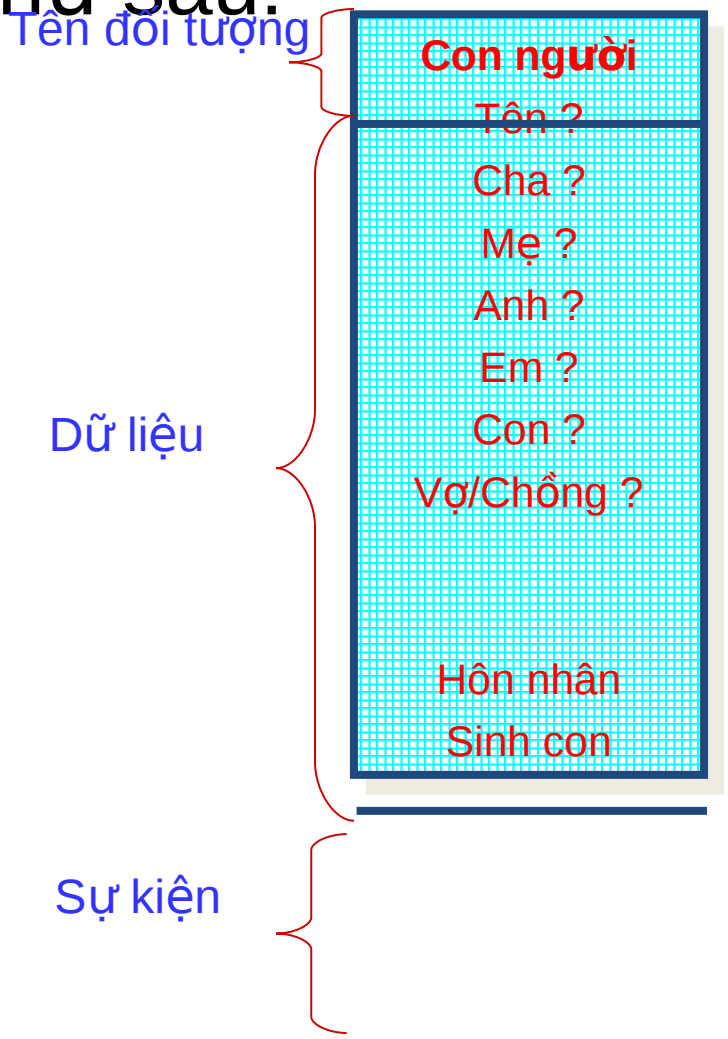
- Sự kiện hôn nhân: Thêm mối quan hệ thông gia.
- Sự kiện sinh con: Khi người phụ nữ sinh con, đứa bé cô ta sinh ra sẽ có:
  - Mẹ là cô ta.
  - Bố là chồng cô ta.
  - Đứa bé sẽ có thêm những người anh/chị.
  - Chồng cô ta có con là đứa bé.
  - Những người Anh/Chị có thêm đứa em.
  - ...

Bài toán quan hệ gia đình *(tiếp)*

- Khi nói đến một sự kiện nào thì phải chỉ ra nó được phát sinh bởi người nào.
- Khi một sự kiện của một người nào đó xẩy ra thì các dữ liệu của người đó sẽ bị thay đổi và các dữ liệu của các người liên quan sẽ thay đổi theo.
- Sự đóng gói giữa dữ liệu và sự kiện tạo ra đối tượng.

Bài toán quan hệ gia đình *(tiếp)*

- Đối tượng con người được mô tả như sau:



Bài toán quan hệ gia đình *(tiếp)*

- Để trả lời các câu hỏi về mối quan hệ gia đình "*X và Y có quan hệ với nhau như thế nào ?*", ta cần trả lời các câu hỏi nhỏ:
  - X có phải là chồng của Y không ?
  - X có phải là con của Y không ?
  - ..
- Để trả lời chúng ta chỉ cần kiểm tra các thuộc tính của Y có tồn tại X hay không.

Bài toán quan hệ gia đình *(tiếp)*

- Dễ thấy rằng chúng ta không cần quan tâm đến cách tạo ra một cấu trúc cây quan hệ mà vẫn có thể giải quyết được bài toán.
- Bài toán được phân tích rất gần với thực tế.

Lập trình hướng đối tượng

- Đối tượng = Dữ liệu + Phương thức.
- Lớp: Tập các đối tượng có cùng cấu trúc dữ liệu.
- Tính kế thừa: Cho phép định nghĩa một lớp mới dựa trên các lớp đã có và bổ sung thêm những thành phần dữ liệu hay phương thức mới.
- Tính tương ứng bội.

Chương 2. Mở rộng của C++

IT Faculty, Vinh University

Mục đích

- Giới thiệu một số mở rộng của C++ hỗ trợ cho lập trình hướng đối tượng.

Nội dung

- Toán tử xuất, nhập
- Toán tử phạm vi
- Biến tham chiếu
- Tham số ngầm định của hàm
- Hàm inline
- Định nghĩa chồng hàm
- Định nghĩa chồng toán tử
- Toán tử New và Delete

Toán tử xuất, nhập

- Yêu cầu: Mở thư viện iostream.h
- Toán tử xuất: <<
  - Cú pháp: cout<<[biểu thức 1]<<...
  - Ý nghĩa: Dùng để in giá trị của biểu thức.
  - Chú ý:
    - [Biểu thức] có thể chứa các ký tự điều khiển (\n, \t, ...).
    - Không cần định dạng dữ liệu khi xuất.

Toán tử xuất, nhập

- Toán tử nhập: >>
  - Cú pháp: cin>>[biến 1] >>[biến 2] >>...
  - Ý nghĩa: Dùng để nhập giá trị cho các biến.
  - Chú ý:
    - Biến phải được khai báo trước.
    - Không cần định dạng dữ liệu nhập.
    - Không nhận dữ liệu nhập là dấu cách, dấu tab.

Toán tử xuất, nhập

- Ví dụ 1: Viết chương trình tính diện tích và chu vi của hình chữ nhật.
- Ví dụ 2: Viết chương trình nhập vào 1 mảng 2 chiều n dòng, m cột các số thực. In mảng đã nhập dạng ma trận và ma trận chuyển vị.

Toán tử phạm vi

- Khi có một khai báo trùng tên giữa biến cục bộ trong hàm và biến tổng thể, nếu truy nhập đến biến trùng tên trong hàm thì bộ biên dịch hiểu là truy nhập biến cục bộ.
- Để truy nhập đến biến tổng thể, sử dụng toán tử phạm vi (::).

Toán tử phạm vi

- Ví dụ 2.2: Cho đoạn chương trình:

```
int i=5;
void main()
 {int i=2, j=3;
    i++;
    i+=::i+j;
    j++;
    ::i+=i+j;
 }
```

  – Tìm giá trị của i cục bộ, i tổng thể.

Biến tham chiếu

- Biến tham chiếu (reference) là *bí danh* của một đối tượng.
- Một biến tham chiếu dùng để tham chiếu tới một biến cùng kiểu trong bộ nhớ.
- *Các phép toán thao tác trên biến tham chiếu thực chất là thao tác đến biến nhớ mà nó tham chiếu đến*.
- Khai báo:

  <Kiểu dữ liệu> &<biến tham chiếu>=<tên biến>;

Biến tham chiếu

- Ví dụ 2.3: Biến tham chiếu

```
void main() {
    int i=2, j=3;
    int &r=i;          // r tham chiếu đến i
    i++;               // i=3, r= ?
    r=6;               // r=6, i=?
    int &p=j;          // p tham chiếu đến j
    p=i;               // i=?, p=?, j=?, r=?
}
```

Tham số hàm là tham chiếu

- Khi khai báo biến tham chiếu phải xác lập biến mà nó tham chiếu đến.
- Có thể dùng biến tham chiếu làm tham số cho hàm.
- Khi sử dụng biến tham chiếu làm tham số hàm, chương trình dịch sẽ truyền địa chỉ của biến cho hàm (truyền tham biến).

Tham số hàm là tham chiếu

- Ví dụ 2.4: Hoán đổi giá trị 2 biến số thực
  - Hàm hoán đổi 1: Sử dụng biến

```
void swap(float x, float y) {
    float t = x; x = y; y= t;
}
```

  - Hàm hoán đổi 2: Sử dụng con trỏ

```
void swap(float *x, float *y) {
    float t = *x; *x = *y; *y= t;
}
```

Tham số hàm là tham chiếu

- Hàm hoán đổi 3: Sử dụng tham chiếu

```
void swap(float &x, float &y) {
    float t = x; x = y; y= t;
}
```

- Hỏi hàm nào hoán đổi được ?
- Hoàn thiện chương trình, sử dụng debug để xem xét việc truyền tham trị và tham biến.
- Lợi ích của truyền tham biến so với truyền tham trị ?.

Hàm trả về tham chiếu

- Định nghĩa hàm trả về tham chiếu:

  <type> &<tên hàm> (các tham số) {

  <Nội dung hàm>

  return <biến có phạm vi toàn cục>;

  }

  – <type>: Kiểu dữ liệu trả về của hàm.
- Tìm ví dụ minh hoạ khái niệm này.

Tham số ngầm định của hàm

- Đối với hàm định nghĩa tham số ngầm định, khi gọi hàm có thể khuyết các tham số có giá trị ngầm định, khi đó hàm lấy giá trị truyền vào là giá trị ngầm định.
- Ví dụ 2.5: Viết các hàm tính diện tích và chu vi hình chữ nhật có định nghĩa các tham số có giá trị ngầm định.

Tham số ngầm định của hàm

- Hàm tính diện tích:

```
float dientich(float a=1, float b=2) {
    return a*b;
}
```

- Hàm tính chu vi:

```
float chuvi(float a, float b=1) {
    return (a+b)*2;
}
```

Tham số ngầm định của hàm

- Chương trình chính:

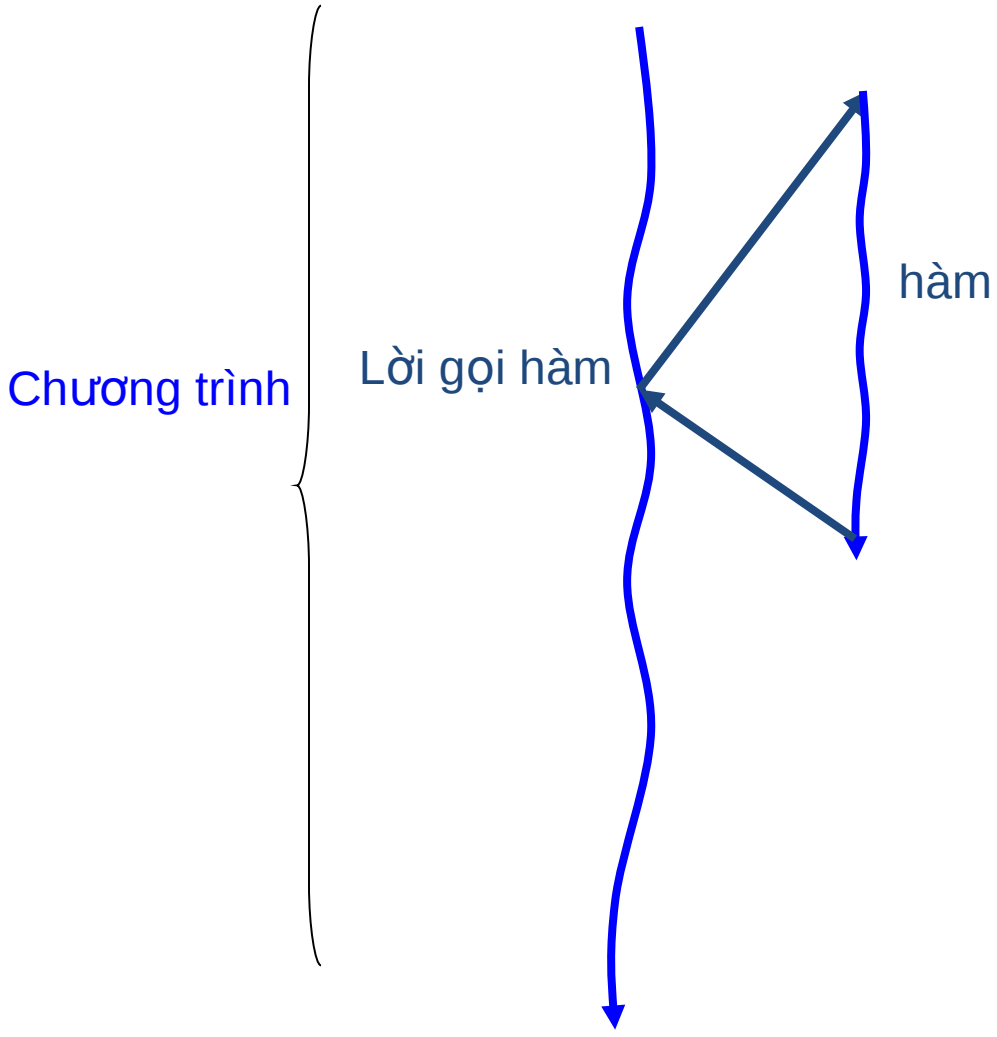
```
void main() {
    cout<<“dien tich :”<<dientich();
                // ?
    cout<<“dien tich :”<<dientich(3);   // ?
    cout<<“dien tich :”<<dientich(2,3);// ?
    cout<<“chu vi :”<<chuvi();          // ?
    cout<<“chu vi :”<<chuvi(3);         // ?
    cout<<“chu vi :”<<chuvi(2,3);
                // ?
    getch();
}
```

Tham số ngầm định của hàm

- Các tham số ngầm định phải đặt ở cuối danh sách các tham số của hàm. Khi 1 tham số phía sau không có giá trị ngầm định thì các tham số trước nó cũng không có giá trị ngầm định.
  - void f(int a, int b=1, int c=2) // OK
  - void f(int a, int b=1, int c)
    // !OK
- Các tham số ngầm định có thể lấy giá trị của 1 biểu thức. (46K1)

Hàm inline

- Khi có 1 lời gọi hàm trong chương trình:



Hàm inline

- Khi có lời gọi hàm trong chương trình thì chương trình phải thực hiện:
  - Lưu các trạng thái đang thực hiện dở.
  - Lưu địa chỉ trở về.
  - Chuyển sang thực hiện hàm.
  - Cấp phát bộ nhớ cục bộ của hàm.
  - Thực hiện hàm.
  - Giải phóng vùng nhớ cục bộ.
  - Khôi phục các trạng thái đã cất và thực hiện tiếp chương trình.

Hàm inline

- Nếu hàm đơn giản, chỉ có ít dòng lệnh thì thời gian gọi hàm lâu hơn thời gian thực hiện hàm
- Để tránh điều này -> định nghĩa hàm inline.
- Định nghĩa hàm inline: Thêm từ khoá inline lên đầu hàm thông thường.
- Khi có lời gọi hàm ở đâu thì bộ biên dịch chèn trực tiếp đoạn mã của hàm vào mà không gọi theo cách thông thường.

Hàm inline

- Ví dụ 2.6: Định nghĩa hàm inline tìm giá trị lớn nhất của 2 biến số thực:

```
inline float max(float x, float y) {
    return (x>y)? x:y;
}
```

- Trong C++, những hàm có cấu trúc lặp không nên định nghĩa là hàm inline.

Định nghĩa chồng hàm

- Định nghĩa chồng hàm là định nghĩa các hàm trùng tên mà có các tham số khác nhau.
- Khi gọi hàm, trình biên dịch dựa vào tham số hàm để xác định hàm nào được gọi.
- Ví dụ 2.6: Định nghĩa chồng hàm
  - Hàm tìm max của 2 số nguyên:

```
int max(int x, int y) {
    return (x>y) ? x:y;
}
```

Định nghĩa chồng hàm

– Hàm tìm max của 2 số thực:

```
float max(float x, float y) {
    return (x>y) ? x:y;
}
```

- Chương trình chính:

```
void main() {
    cout<<"\n Max ="<< max(2,5);        // gọi hàm max ?
    cout<<"\n Max ="<< max(2.5,5);      // gọi hàm max ?
    getch();
}
```

Định nghĩa chồng toán tử

- Định nghĩa các toán tử trùng tên cho các kiểu dữ liệu khác nhau.
- Ví dụ 2.7: Cho cấu trúc phân số như sau:

```
struct ps{
    int ts;
    int ms;
};
```

  - Định nghĩa các phép toán +, -, *, /, -(đảo dấu)...?

Định nghĩa chồng toán tử

- Phương án 1: Viết các hàm có dạng:

```
ps cong(ps a, ps b) {
    ps c;
    c.ts  =  a.ts*b.ms + a.ms*b.ts;
    c.ms =  a.ms*b.ms;
    return c;
}
```

  – Tương tự cho -, *, /, -(đảo dấu),...

Định nghĩa chồng toán tử

- Sử dụng các hàm đã viết tính giá trị của:
  - s1 = -a+b–c;
  - s2 = -a*b+c*d-e;
  - với a, b, c, d, e là các phân số.
- Cách viết:
  - s1 = cong(daodau(a),tru(b,c)) hoặc
  - s2 = tru(cong(daodau(a),b),c)
- **Chú ý:**
  - Cách biểu diễn là không duy nhất.
  - Cách biểu diễn rất phức tạp, không thể hiện rõ công thức.

Định nghĩa chồng toán tử

- Phương án 2: Định nghĩa chồng toán tử
  - Định nghĩa toán tử giống như định nghĩa hàm, chỉ thay tên hàm bằng từ khoá operator <toán tử>.
  - Định nghĩa:

```
<type> operator <toán tử> (các tham số) {
    <Nội dung>
}
```

  - <type> là kiểu dữ liệu trả về của hàm.

Định nghĩa chồng toán tử

- Định nghĩa phép toán cho cấu trúc phân số
  - Phép toán +

```
ps operator +(ps a, ps b) {
    ps c;
    c.ts  =  a.ts*b.ms + a.ms*b.ts;
    c.ms =  a.ms*b.ms;
    return c;
}
```

Định nghĩa chồng toán tử

– Phép toán -

```
ps operator - (ps a, ps b) {
        ps c;
        c.ts  =  a.ts*b.ms - a.ms*b.ts;
        c.ms =  a.ms*b.ms;
        return c;
}
```

– Phép toán – (đảo dấu): Gọi hàm được sử dụng chồng hàm.

Toán tử new và delete

- Toán tử new
  - Dùng để cấp phát bộ nhớ động
  - Cú pháp: <con trỏ> = new <type>[n] (n≥ 0)
  - Ví dụ:
    - float *x = new float[50];
    - char *s = new char[30];
- Toán tử delete
  - Dùng để xoá vùng nhớ đã cấp bởi toán tử new.
  - Ví dụ: delete x; delete a;

Bài tập

- Bài 1: Cho cấu trúc phân số

```
struct ps{
    int ts;
    int ms;
}
```

  - Viết hàm nhập 1 phân số (hoặc toán tử nhập)
  - Viết hàm in 1 phân số (hoặc toán tử xuất)
  - Viết hàm rút gọn 1 phân số
  - Định nghĩa các toán tử: +, -, *, /

Bài tập

- Định nghĩa toán tử -(đảo dấu).
- Định nghĩa các toán tử so sánh: >, >=, <, <=, ==, !=;
- Định nghĩa các toán tử: ++, --, +=, -=, *=.
- Viết chương trình nhập vào 2 phân số, minh hoạ các phép toán trên.
- Viết chương trình nhập vào 1 mảng a gồm n phân số:
  - Tính tổng, tích của mảng
  - Tìm phân số lớn nhất, bé nhất.

Bài tập

- Bài 2: Cho cấu trúc số phức
  - struct complex{
    - foat pt;
    - float pa;
  - }
  - Viết hàm nhập 1 số phức
  - Viết hàm in 1 số phức
  - Định nghĩa các toán tử: +, -, *.
  - Viết chương trình nhập 2 số phức, tính và in ra tổng, hiệu, tích của chúng.

Bài tập

- Bài 3: Sử dụng toán tử new để tạo 1 mảng 2 chiều các số thực n dòng, m cột. Tính và in ra tổng các hàng của mảng. Giải phóng vùng nhớ đã cấp cho mảng.
- Sử dụng tài liệu tham khảo [1].
- Kiểm tra bài tập

Chapter 3. Object & Class

Faculty of Information Technology

Vinh University

Mục đích

- Giới thiệu các khái niệm cơ bản của lập trình hướng đối tượng.
- Trang bị các kỹ năng xây dựng lớp và các thành phần của lớp.
- Sau khi kết thúc chương có thể đặc tả và giải quyết các bài toán dựa trên hướng đối tượng.

Nội dung

- Đối tượng
- Lớp
- Hàm thiết lập và hàm huỷ bỏ
- Hàm thiết lập sao chép
- Phép gán đối tượng
- Các thành phần tĩnh
- Hàm bạn và lớp bạn
- Bài tập + Kiểm tra

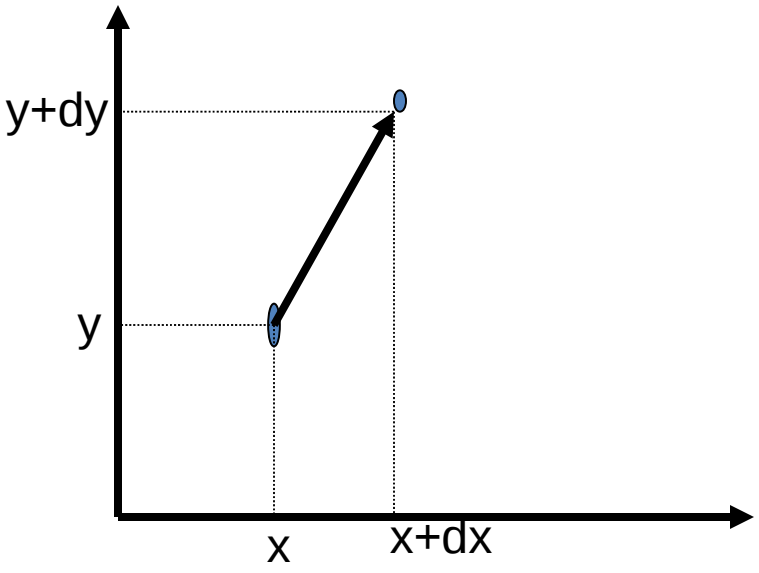
Đối tượng (object)

- Đối tượng là sự đóng gói của dữ liệu và phương thức.
- Đối tượng = Dữ liệu + Phương thức

  (object = data + method)
  - Dữ liệu: Mô tả đối tượng.
  - Phương thức: Các hàm xử lý của đối tượng.
  - Trong C++, phương thức là các hàm.
- Mỗi đối tượng có dữ liệu riêng và phương thức riêng.

Đối tượng

- Ví dụ 3.1. Một mô tả về 1 đối tượng điểm như sau:
  - Một đối tượng được xác định bởi cặp toạ độ (x,y).



Đối tượng

- Các thao tác tác động lên đối tượng điểm gồm:
  - Hàm đặt toạ độ của điểm ở toạ độ (ox, oy);
  - Hàm tịnh tiến điểm có toạ độ (x,y) đến điểm có toạ độ (x+dx, y+dy).
  - Hàm hiển thị toạ độ điểm.

- Đối tượng điểm có thể được mô tả như sau:
  - Dữ liệu:
    - Cặp toạ độ (x,y).
  - Phương thức:
    - Hàm đặt toạ độ điểm.
    - Hàm tịnh tiến.
    - Hàm hiển thị toạ độ của điểm

Đối tượng

- Khai báo về dữ liệu:
  - float x,y;
- Phương thức xử lý dữ liệu:
  - Hàm đặt toạ độ của điểm tại (ox,oy)

```
void init(float ox, float oy) {
        x = ox;
        y = oy;
}
```

Đối tượng

- Hàm tịnh tiến toạ độ điểm:

```
void move(float dx, float dy) {
    x+ = dx;
    y+ = dy;
}
```

- Hàm hiển thị toạ độ điểm:

```
void display() {
    cout<<“\n x = “<<x;
    cout<<“\n y= “<<y;
}
```

Đối tượng

- Bài tập 3.1:
  - Hãy mô tả bài toán tính diện tích và chu vi đường tròn về dạng mô tả 1 đối tượng đường tròn.
  - Hãy mô tả bài toán tính diện tích và chu vi hình chữ nhật về dạng mô tả 1 đối tượng hình chữ nhật.
  - Hãy mô tả bài toán giải phương trình bậc nhất về dạng mô tả 1 đối tượng phương trình bậc nhất.

Lớp (class)

- Một mô tả chung cho các đối tượng cùng loại gọi là lớp.
- Lớp là mô tả tổng quát của đối tượng, đối tượng là 1 thể hiện cụ thể (instance) của lớp.
- Trong C++, lớp được định nghĩa như 1 cấu trúc nhưng có thêm các hàm thành phần.

Khai báo lớp

- Khai báo lớp:

```
class <tên lớp>{
    private:
        <Khai báo và định nghĩa các thành phần riêng>
    public:
        <Khai báo và định nghĩa các thành phần công cộng>
};  // Kết thúc khai báo lớp
<Định nghĩa các thành phần chưa đ/n trong khai báo lớp>.
```

Khai báo lớp

- Trong đó:
  - Tên lớp: được đặt theo quy tắc đặt tên biến.
  - Thành phần: được hiểu là dữ liệu và các hàm.
  - Khai báo:
    - Đối với biến: Khai báo như khai báo biến.
    - Đối vơi hàm: Khai báo nguyên mẫu của hàm.
  - Định nghĩa:Viết nội dung hàm.
  - Thứ tự khai báo dữ liệu và các hàm là không quan trọng.
  - Hàm định nghĩa trong khai báo lớp là hàm inline.

Xác định quyền truy nhập

- Quyền truy nhập là khả năng truy nhập của một hàm, một lớp nào đó đến các thành phần của lớp.
- Mọi thành phần liệt kê trong phần public của lớp truy nhập được ở bất kỳ các hàm, các lớp khác.
- Mọi thành phần liệt kê trong phần private của lớp chỉ truy nhập được trong lớp.

Khai báo lớp

- Ví dụ 3.2. Lớp các điểm trên mặt phẳng

```
#include <iostream.h>
#include <conio.h>
class point{
    private:
        float x, y;
    public:
        // Khai báo và định nghĩa hàm thành
           phần
        void init(float ox, float oy){
            x=ox; y=oy;
        }
```

Khai báo lớp

```
    //Khai báo nguyên mẫu các hàm.
    void move(float dx, float dy);
    void display();
}; // Kết thúc khai báo lớp
//Định nghĩa các hàm chưa định nghĩa trong lớp.
    void point::move(float dx, float dy) {
        x+ = dx; y+ = dy;
    }
    void point::display() {
        cout<<“\n x= “<<x<<“ y =“<<y;
    }
```

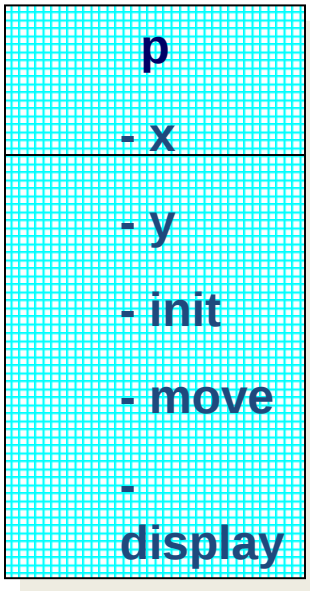
Khai báo lớp

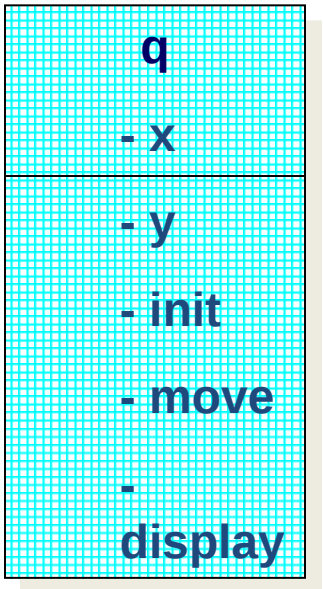
- Chương trình chính

```
void main() {
    point p;                    // Khai báo đối tượng
    p.init(2,3);                // Gọi hàm thành phần của đối tượng.
    p.display();
    p.move(3,4);
    p.display();
    getch();
}
```

Khai báo lớp

- Tạo đối tượng:
  - <Tên lớp> <Tên đối tượng>;
  - Mỗi đối tượng có 1 tập các thành phần riêng.
  - point p,q;





Khai báo lớp

- Đối tượng là 1 kiểu dữ liệu do đó có thể khai báo con trỏ và tham chiếu.

Khai báo con trỏ

```
point q;
q.init(2,3);
point *p;
p=&q;
p->move(3,4)
(q.x =5, q.y =7)
```

Khai báo tham chiếu

```
point q;
q.init(2,3);
point &r =q;
r.move(3,4)
(q.x =5, q.y =7)
```

Các thành phần dữ liệu

- Khai báo: <Kiểu dữ liệu> <tên thành phần>;
- Kiểu dữ liệu:
  - Kiểu dữ liệu chuẩn:(int, float, char, char *...)
  - Kiểu cấu trúc.
  - Kiểu lớp.
- Ví dụ 3.2. Kiểu dữ liệu là lớp
  - Xây dựng 1 lớp point gồm:
    - Các thuộc tính x,y mô tả toạ độ của điểm
    - Hàm đặt toạ độ của điểm tại (ox, oy).

Các thành phần dữ liệu

- Hàm tịnh tiến toạ độ của điểm (x,y) đến điểm (x+dx,y+dy).
- Hàm hiển thị toạ độ của điểm.

- Xây dựng 1 lớp đường tròn gồm:
  - Thuộc tính r là 1 số thực mô tả bán kính đường tròn.
  - Thuộc tính O là 1 điểm (point) mô tả tâm đường tròn.
  - Hàm tạo đường tròn có bán kính là k và tâm tại điểm A.
  - Hàm hiển thị toạ độ tâm và bán kính của đường tròn.

Các thành phần dữ liệu

```
class point{
    private:
        float x, y;
    public:
        void init(float ox, float oy){
            x=ox; y=oy;
        }
        void move(float dx, float dy){
            x+=dx; y+=dy;
        }
        void display(){
            cout<<“\n x= “<<x<<“ y=<<y;
        }
};
```

Các thành phần dữ liệu

```
class circle{
    private:
        point O;
        public
          float r;
    public:
        void init(float k, float xx, float yy){
            r=k;
            O.init(xx,yy);
        }
        void display(){
            cout<<“\n r =“<< r;
            O.display();
        }
};
```

Các thành phần dữ liệu

- Chương trình chính:
  - Nhập vào cặp toạ độ (x, y) và 1 số thực r. Tạo đường tròn C có tâm tại toạ độ (x, y) và bán kinh r. Hiển thị tâm và bán kính đường tròn.
- Bài tập 3.2.
  - Mở rộng bài toán này bằng cách thêm hàm tịnh tiến đường tròn. Nhập vào (dx, dy) và tịnh tiến đường tròn C.

Các hàm thành phần

- Hàm được khai báo trong khai báo của lớp gọi là hàm thành phần của lớp.
- Các hàm thành phần được phép truy nhập đến các thành phần dữ liệu và hàm thành phần khác trong lớp.
- Định nghĩa hàm thành phần có thể đặt trong hay ngoài khai báo lớp.
- Khi đặt trong lớp, nó là hàm inline.

Các hàm thành phần

- Khi đặt ngoài lớp, cách định nghĩa như sau:

  <Kiểu dữ liệu> <tên lớp>::<tên hàm>(tham số){

  <Nội dung hàm>

  }
- Cũng có thể khai báo lớp ở 1 tệp .h và định nghĩa hàm thành phần ở 1 tệp khác.
- Bài tập 3.3.
  - Chuyển khai báo lớp point vào tệp mypoint.h và viết các hàm ở tệp mypoint.cpp.

Phạm vi lớp

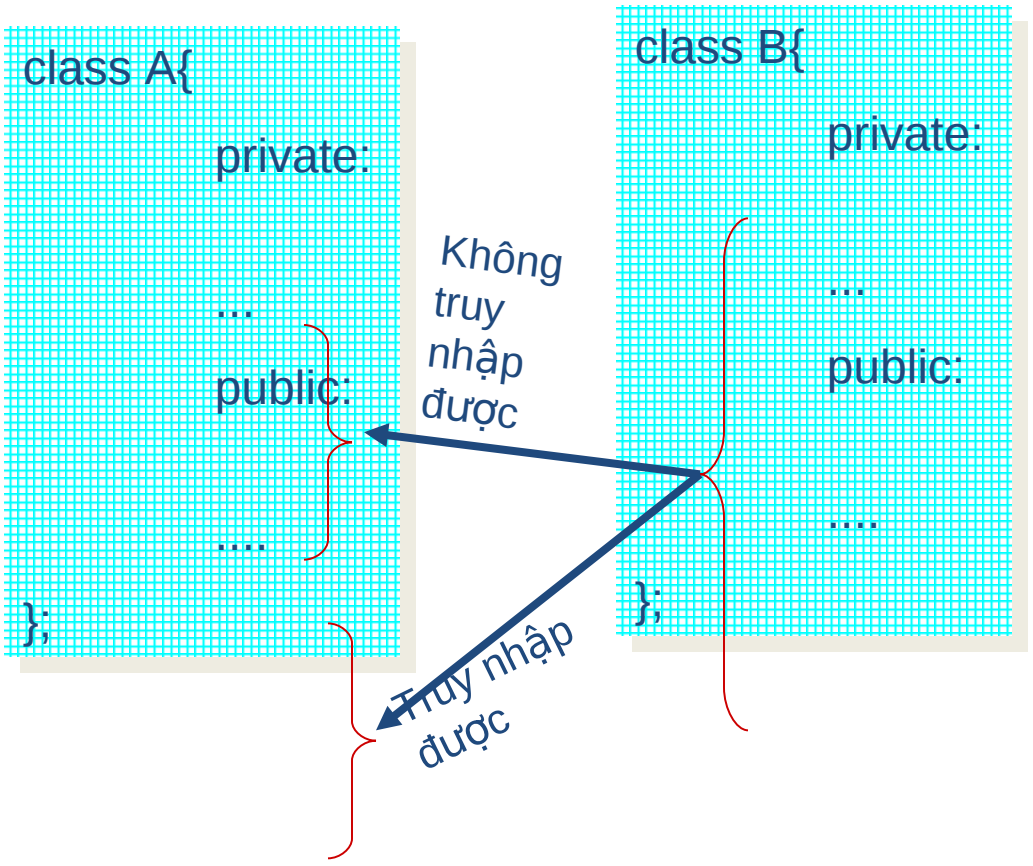
- Phạm vi lớp là khái niệm kiểm soát truy nhập đến các thành phần dữ liệu và các hàm thành phần của lớp.
- Tất cả các thành phần của lớp là thuộc phạm vi lớp, tức là nó có thể truy nhập đến các thành phần khác của cùng lớp.

Xác định quyền truy nhập

- Quyền truy nhập là khả năng truy nhập của một hàm, một lớp nào đó đến các thành phần của lớp.
- Mọi thành phần liệt kê trong phần public của lớp truy nhập được ở bất kỳ các hàm, các lớp khác.
- Mọi thành phần liệt kê trong phần private của lớp chỉ truy nhập được trong lớp.

Xác định quyền truy nhập

- Trong lớp có thể có nhiều nhãn private và public.



Khả năng của hàm thành phần

- Cho phép định nghĩa chồng hàm thành phần.
- Cho phép sử dụng tham số ngầm định.
- Cho phép sử dụng đối tượng làm tham số hàm thành phần.
- Cho phép đối tượng làm giá trị trả về.
- Con trỏ this: Là địa chỉ của đối tượng đang xét.

Hàm thiết lập (constructor)

- Khi 1 đối tượng được tạo ra, nó luôn gọi đến 1 hàm thiết lập của nó.
- Chức năng hàm thiết lập:
  - Khởi tạo các thành phần dữ liệu tĩnh.
  - Xin cấp phát bộ nhớ cho các thành phần dữ liệu động.
- Các quy định xây dựng hàm thiết lập:
  - Hàm thiết lập có tên trùng với tên lớp.
  - Hàm thiết lập phải có quyền public.

Hàm thiết lập

- Hàm không có giá trị trả về và không cần từ khoá void.
- Một lớp có thể định nghĩa nhiều hàm thiết lập. Khi định nghĩa nhiều hàm thiết lập có thể khai báo đối tượng theo nhiều dạng khác nhau.
- Hàm thiết lập có thể khai báo với các tham số có giá trị ngầm định.
- Khi không có hàm thiết lập được định nghĩa, lớp sử dụng hàm thiết lập ngầm định (hàm không có tham số và không làm gì cả).

Hàm thiết lập

- Ví dụ 3.3. Xây dựng lớp point sử dụng hàm thiết lập.
  - Hàm thiết lập không tham số

```
point() {
    x=0; y=0;
}
```

  - Hàm thiết lập 2 tham số

```
point(float ox, float oy) {
    x=ox; y=oy;
}
```

Hàm thiết lập

```
class point{
    private:
        float x, y;
    public:
        point() {
            x=0; y=0;
        }
        point(float ox, float oy){
            x=ox; y=oy;
        }
        void move(float dx, float dy);
        void display();
};
```

Hàm thiết lập

- Định nghĩa các hàm chưa được định nghĩa

```
void point::move(float dx, float dy) {
        x+=dx; y+=dy;
}
void display() {
        cout<<“\n x =“<< x <<“ y =
  “<<y;
}
```

- Dựa trên các hàm thiết lập, xác định các khai báo của đối tượng:
  - pointA; // Hàm thiết lập nào được gọi ?, x=?, y=?
  - point B(1,2); // Hàm thiết lập nào được gọi ?, x =?, y =?
  - point C(2,3); // Hàm thiết lập nào được gọi ?

Hàm thiết lập

- Ví dụ 3.4. Cho khai báo lớp:

```
class abc{
    ...
    public:
        abc(int x);
        abc(char *s=null):
        abc(float x, float y, float z=0);
        abc(int x, char *s = null);
};
```

  - Hãy tìm ra các khai báo đối tượng có thể có.

Hàm thiết lập

- Ví dụ 3.5. Cho 2 khai báo lớp:

```
class A{
    ...
    public:
        A() ;
        A(int x);
};
```

```
class B{
    ...
    public:
        //Không có hàm thiết lập.
};
```

- Lớp A sử dụng hàm thiết lập ?, lớp B sử dụng hàm thiết lập nào?

Hàm huỷ bỏ (Destructor)

- Hàm huỷ bỏ được gọi khi đối tượng bị huỷ khỏi bộ nhớ.
- Chức năng:
  - Giải phóng bộ nhớ cho do đối tượng chiếm dữ.
- Các quy định xây dựng hàm huỷ bỏ:
  - Tên hàm: ~<tên lớp>
  - Hàm huỷ bỏ có quyền public
  - Hàm huỷ bỏ không cần tham số

Hàm huỷ bỏ

- Mỗi lớp chỉ cần hàm huỷ bỏ
- Khi không định nghĩa hàm huỷ bỏ, lớp sử dụng hàm huỷ bỏ ngầm định.
- Hàm huỷ bỏ không có giá trị trả về.
- Đối với các lớp không có thành phần dữ liệu động, chỉ cần sử dụng hàm huỷ bỏ ngầm định.

Hàm thiết lập sao chép

- Là hàm thiết lập tạo ra một đối tượng mới từ đối tượng đã có.
- Chức năng:
  - Tạo ra đối tượng mới
  - Sao chép các thành phần dữ liệu từ đối tượng đã có sang đối tượng mới.
- Khai báo hàm thiết lập sao chép:
  - Hàm chỉ có 1 tham số là 1 tham chiếu đến 1 đối tượng cùng lớp.

Hàm thiết lập sao chép

- Nếu trong lớp không định nghĩa hàm thiết lập sao chép thì lớp sử dụng hàm thiết lập sao chép ngầm định.
- Nếu lớp không có thành phần dữ liệu động thì chỉ cần sử dụng hàm thiết lập sao chép ngầm định, ngược lại phải định nghĩa hàm thiết lập sao chép.

Phép gán đối tượng

- Phép gán đối tượng sẽ sao chép dữ liệu từ đối tượng đã có sang đối tượng được gán.
- Nếu trong lớp có thành phần dữ liệu động thì phải định nghĩa toán tử gán, ngược lại chỉ cần sử dụng toán tử gán ngầm định
- Ví dụ:
  - point a(2,3);
  - point b=a; // Sử dụng toán tử gán ngầm định

Phép gán đối tượng

- Bài tập 3.3. Xây dựng lớp véc tơ
  - Định nghĩa các hàm thiết lập 1 tham số.
  - Định nghĩa hàm thiết lập sao chép.
  - Định nghĩa hàm huỷ bỏ.
  - Định nghĩa phép gán đối tượng.
  - Viết hàm nhập 1 vector.
  - Viết hàm in 1 véc tơ.
  - Viết chương trình chính.
  - Sử dụng debug kiểm tra cách gọi phép gán, hàm thiết lập sao, hàm huỷ bỏ.

Phép gán đối tượng

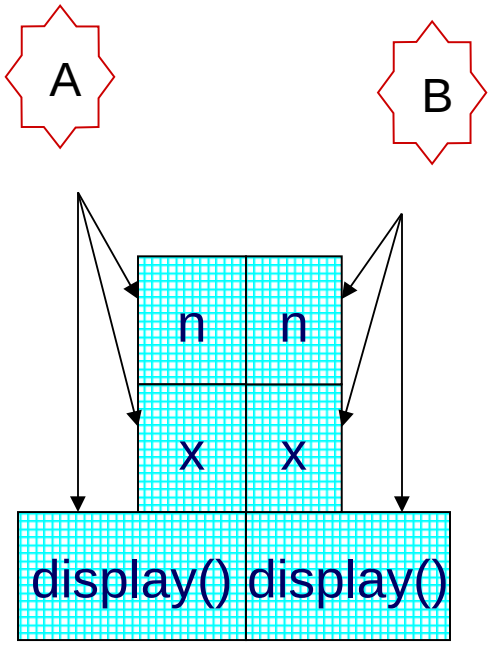
```
class vector{
    private:
        int n;                          // Số
          phần tử
        float *v;                       // Con trỏ
          trỏ đến các pt.
    public:
        vector(int size);                       //
          Hàm thiết lập.
        vector(vector &a);              // Hàm
          thiết lập sao chép.
        ~vector();                      // Hàm
          huỷ bỏ.
        vector operator =(vector &a);       //
          Phép gán
        void add();                     // Hàm
          nhập
        void display();                         //
          Hàm in vector
};
```

Các thành phần tĩnh

- Trong 1 chương trình, các đối tượng thuộc cùng 1 lớp sở hữu các thành phần dữ liệu riêng và các hàm riêng của nó.
- Ví dụ khai báo lớp:

Khai báo: abc A, B;

```
class abc{
        private:
                int n;
                float x;
        public:
                void
                display();
};
```

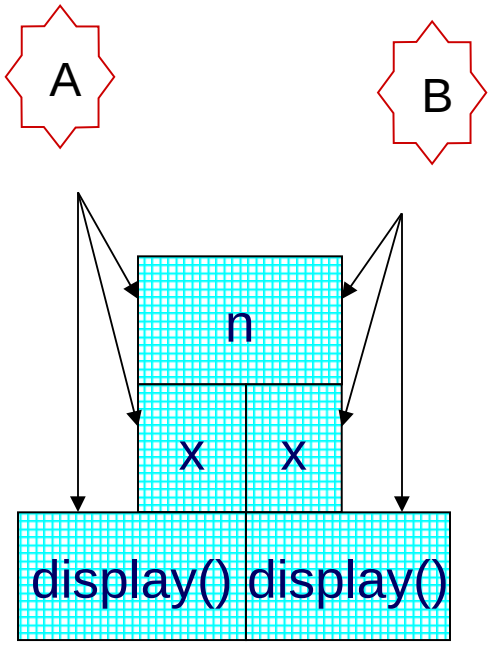


Thành phần tĩnh

- Để chia sẽ các thành phần chung của các đối tượng, đặt khoá static trước thành phần khai báo.
- Ví dụ với khai báo lớp:

Khai báo: abc A, B;

```
class abc{
        private:
                static   int
                n;
                float x;
        public:
                void
                display();
};
```



Thành phần tĩnh

- Khởi tạo các thành phần tĩnh
  - Đối với dữ liệu, thành phần tĩnh phải được khởi tạo ngoài lớp, ngoài các hàm.
  - Đối với hàm thành phần, có thể gọi thông qua các đối tượng hoặc gọi theo cách:
    - <Tên lớp>::<tên hàm>(các tham số);
- Chú ý rằng thành phần dữ liệu tĩnh có thể khai báo ở private hoặc public.

Thành phần tĩnh

- Ví dụ: Tạo 1 lớp student mô tả các đối tượng sinh viên, lớp gồm:
  - Thuộc tính name mô tả tên sinh viên.
  - Thuộc tính fee mô tả tiền học phí.
  - Thuộc tính total mô tả tổng học phí đã nhập.
  - Viết chương trình nhập thông tin và tạo 3 đối tượng sinh viên, tính tổng học phí của 3 sinh viên đã tạo, hiển thị các thông tin về sinh viên.

Thành phần tĩnh

```
class student{
    private:
        char *name;
        float fee;
        static float total;
    public:
        student(char *s= NULL, float f=0){
            name = strdup(s);
            fee=f;
            total += fee;
        }
        void Display() {
        cout<<“\n Name:”<<name<<“\t
           Fee:”<<fee;
        cout<<“\n Total:”<<total;
        }
};
```

Thành phần tĩnh

```
//Khởi tạo thành phần dữ liệu tĩnh ở ngoài lớp,
    ngoài các hàm.
float student::total=0;
//Chương trình chính
void main() {
      student
        s1(“a”,10),s2(“b”,20),s3(“c”,30);
      s1.Display();
      s2.Display();
      s3.Display();
      getch();
}
```

Hàm bạn & lớp bạn

- Một lớp cho phép 1 hàm hoặc 1 lớp khác truy nhập đến thành phần private của nó bằng cách khai báo hàm đó hoặc lớp đó là friend của lớp.
- Các trường hợp hàm bạn, lớp bạn:
  - Hàm tự do là bạn của lớp.
  - Hàm thành phần của lớp là bạn của lớp khác.
  - Hàm bạn của nhiều lớp.
  - Lớp bạn của 1 lớp.

Hàm bạn & lớp bạn

- Khai báo hàm tự do là bạn của lớp A:

```
class A{
        private:
                ....
        public:
                ....
                friend void f();        // Khai
                báo hàm void f() bạn.
};
// Hàm tự do f()
void f() {
...
}
```

Hàm bạn & lớp bạn

- Hàm thành phần của lớp là bạn của lớp khác:
  - Giả sử có hai lớp A và B, trong lớp B có 1 hàm f.

```
class A; //Khai báo A là 1 lớp
class B{
        ...
        void f();
};
class A{
        ...
        friend void B::f(); //Khai báo hàm f của
        B là bạn của A
};
```

Hàm bạn & lớp bạn

- Hàm bạn của nhiều lớp:
  - Khai báo 1 hàm tự do f() là bạn của 2 lớp A, B:

```
class A{
        ...
        friend void f(); // Khai báo hàm f là bạn
        của lớp A
};
class B{
        ...
        friend void f(); // Khai báo hàm f là bạn
        của lớp B
};
```

Hàm bạn & lớp bạn

- Lớp bạn:
  - Giả sử có 2 lớp A và B. Khai báo lớp B là bạn của lớp A như sau:

```
class B;
class A{
        ...
        friend clas B; // Khai báo lớp B là bạn
        của lớp A
};
class B{
        ...
};
```

Hàm bạn & lớp bạn

- Bài tập 3.4. Hàm tự do là bạn của lớp
  - Xây dựng 1 lớp point gồm:
    - Thuộc tính x, y mô tả toạ độ của điểm.
    - Hàm thiết lập với 2 tham số có giá trị ngầm định bằng 0.
    - Hàm hiển thị toạ độ của điểm.
    - Khai báo hàm tự do tính khoảng cách giữa 2 point là bạn với lớp.
  - Xây dựng 1 lớp line gồm:
    - Hai điểm xác định đoạn thẳng.
    - Hàm thiết lập với 2 tham số là 2 điểm.
    - Hàm hiển thị toạ độ, khoảng cách 2 điểm.

Hàm bạn & lớp bạn

- Viết hàm tự do tính khoảng cách giữa 2 điểm.
- Viết chương trình nhập vào 2 cặp toạ độ (x1,y1) và (x2,y2) của 2 điểm A, B. Tạo đoạn thẳng xác định bởi 2 điểm đó. Hiển thị toạ độ và khoảng cách của 2 điểm xác định đoạn thẳng.
- Mở rộng bài toán trên cho trường hợp tam giác.

Hàm bạn & lớp bạn

- Bài tập 3.5. Định nghĩa toán tử của lớp:
  - Tìm lệnh sai và giải thích lệnh sai trong ví dụ sau:

```
#include<iostream.h>
#include<conio.h>
class integer{
    private:
        int value;
    public:
        integer(int i=0){
            value = i;
        }
```

Bài tập về hàm bạn và lớp bạn

```
integer operator + (integer i){
    return value + i.value;
}
void display(){
    cout<<value<<"\n";
}
}; //Hết lớp
void main(){
    integer i1 = 10;
    integer i2 = i1+ 5;
    integer i3 = 5+i1;
    integer i4 = i2+i3;
}
```

Hàm bạn & lớp bạn

- Giải pháp khắc phục: Chuyển toán tử cộng thành hàm tự do là bạn của lớp.

```
#include<iostream.h>
#include<conio.h>
class integer{
    private:
        int value;
    public:
        integer(int i=0){
            value = i;
        }
```

Bài tập về hàm bạn và lớp bạn

```
friend integer operator + (integer i1, integer i2);
void display(){
    cout<<value<<“\n”;
}
}; //Hết lớp
integer operator + (integer i1, integer i2){
    return i1.value + i2.value;
}
void main(){
    integer i1 = 10;
    integer i2 = i1+ 5;
    integer i3 = 5+i1;
    integer i4 = i1+i2;
}
```

Hàm bạn và lớp bạn

- Nhận xét:
  - Nếu 1 toán tử có n toán hạng, khi định nghĩa nó là hàm thành phần của lớp thì hàm có n-1 tham số, khi định nghĩa nó là hàm tự do thì hàm có n tham số.
- Bài tập cuối chương
  - Xem đề cương.
- Kiểm tra điều kiện 60’.

Tóm tắt

- Đối tượng
- Khai báo lớp, các thành phần dữ liệu, hàm.
- Hàm thiết lập và hàm huỷ bỏ
- Hàm thiết lập sao chép
- Phép gán đối tượng
- Các thành phần tĩnh
- Hàm bạn và lớp bạn

Chapter 4. Inheritance

Faculty of Information Technology

Vinh University

Mục đích

- Giới thiệu cơ bản về đơn kế thừa, đa kế thừa.
- Tính tương ứng bội.
- Bài tập.
- Kiểm tra.

Nội dung

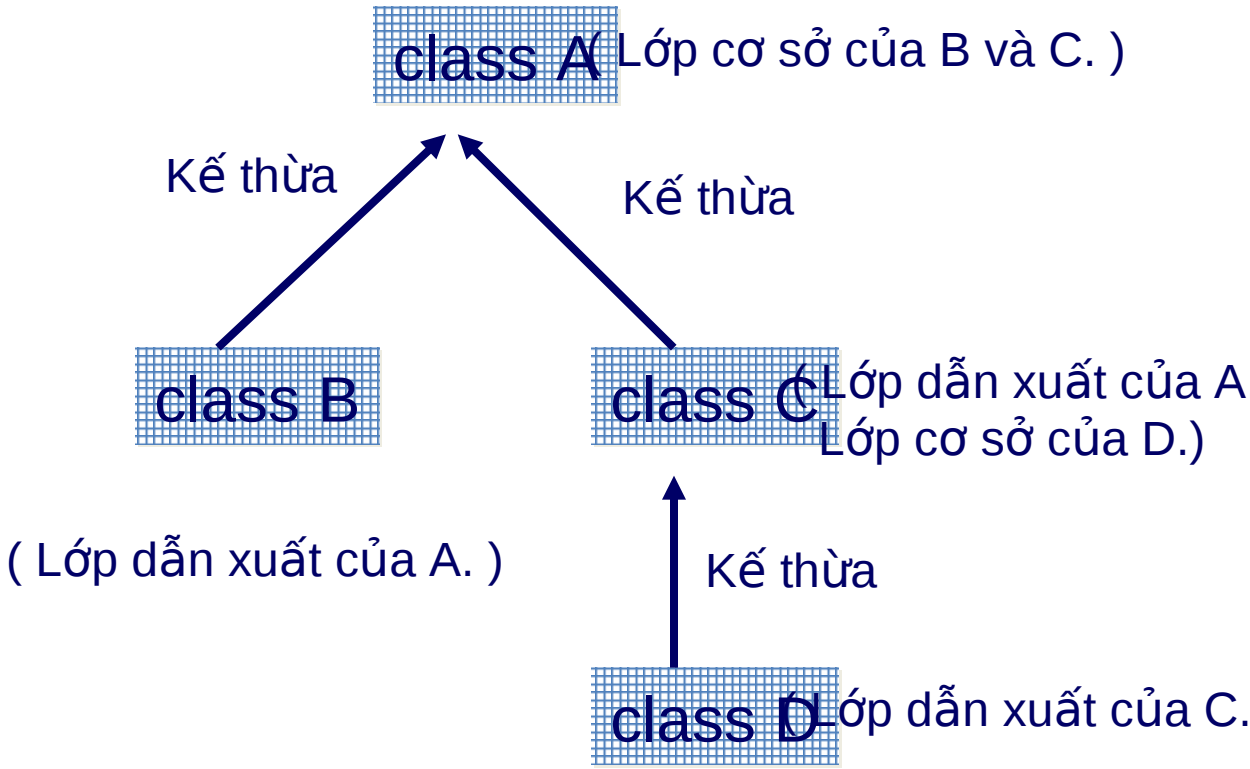
- Khái niệm kế thừa
- Kế thừa đơn giản
  - Truy nhập các thành phần lớp cơ sở
  - Định nghĩa lại hàm thành phần
  - Tính kế thừa trong lớp dẫn xuất
- Kế thừa nhiều lớp
- Lớp cơ sở ảo
- Tương ứng bội

Khái niệm

- Tính kế thừa cho phép định nghĩa một lớp mới dựa trên các lớp đã có.
- Một lớp kế thừa từ 1 lớp khác được gọi là lớp dẫn xuất (derived class).
- Một lớp được lớp khác kế thừa gọi là lớp cơ sở (base class).
- Lớp dẫn xuất sẽ kế thừa các thành phần dữ liệu và hàm thành phần của lớp cơ sở đồng thời bổ sung thêm các thành phần mới.

Khái niệm

- Sự kế thừa cũng cho phép nhiều lớp có thể dẫn xuất từ cùng 1 lớp cơ sở, một lớp dẫn xuất cũng có thể là lớp cơ sở cho lớp khác.



Kế thừa đơn giản

- Ví dụ 4.1. Đơn kế thừa
  - Xây dựng 1 lớp **point** mô tả các điểm trên mặt phẳng, lớp gồm:
    - Hai thuộc tính (x,y) mô tả toạ độ của điểm.
    - Hàm thiết lập không tham số đặt x=0, y=0.
    - Hàm thiết lập 2 tham số (ox, oy).
    - Hàm thiết lập sao chép
    - Hàm tịnh tiến toạ độ của điểm theo dx, dy.
    - Hàm hiển thị toạ độ của điểm.

Kế thừa đơn giản

- Xây dựng 1 lớp **coloredpoint** mô tả các điểm màu. Lớp được kế thừa từ lớp **point** và bổ sung thêm các thành phần:
  - Thuộc tính color mô tả màu của điểm.
  - Hàm thiết lập không tham số đặt x=0, y=0, color =0.
  - Hàm thiết lập 3 tham số (ox, oy, c).
  - Hàm thiết lập sao chép.
  - Hàm hiển thị toạ độ của điểm và màu của điểm.
- Viết chương trình tạo điểm màu, gọi hàm hiển thị và hàm tịnh tiến của lớp cơ sở, lớp dẫn xuất.

Kế thừa đơn giản

- Chưa kế thừa

class point{
- float x, y;
- Hàm thiết lập không tham số;
- Hàm thiết lập 2 tham số;
- Hàm thiết lập sao chép;
- Hàm tịnh tiến;
- Hàm hiển thị;

};

class coloredpoint{
- int color;
- Hàm thiết lập không tham số;
- Hàm thiết lập 2 tham số;
- Hàm thiết lập sao chép;
- Hàm hiển thị;

};

Kế thừa đơn giản

- Sau khi kế thừa

```
class coloredpoint{
    float x, y;                    // Kế thừa từ lớp point
    int color;
    Hàm thiết lập không tham số;
    Hàm thiết lập 2 tham số;
    Hàm thiết lập sao chép;
    Hàm tịnh tiến;                 // Kế thừa từ lớp point
    Hàm hiển thị;                  // Định nghĩa lại của lớp point
};
```

Kế thừa đơn giản

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

Kế thừa đơn giản

```
point(point &p){
      x=p.x; y=p.y;
}
void display(){
    cout<<“\n x =<<x<<“ y=“<<y;
}
void move(float dx, dy){
    x+=dx; y+=dy;
}
};// End class
```

Kế thừa đơn giản

```
class coloredpoint : public point{
    private:
        int color;
    public:
        coloredpoint() : point(){
            color=0;
        }
        coloredpoint(float ox, float oy, int c) :
          point(ox,oy){
            color=c;
        }
```

Kế thừa đơn giản

```
    coloredpoint(coloredpoint &b) : point((point &)b){
        color=b.color;
    }
    void display(){
        point::display();        // Gọi hàm thành phần
      của lớp cơ sở.
        cout<<“ color =<<color;
    }
    };// End class

void main() {
    clrscr();
```

Kế thừa đơn giản

```
coloredpoint m(1,2,3);          // Khai báo đối
    tượng m.
m.display();                    // Gọi hàm của lớp
    dẫn xuất
m.point::display();             // Gọi hàm của lớp
    cơ sở
m.move(4,5);                    // Gọi hàm của lớp
    dẫn xuất
m.display();                    // Gọi hàm của lớp
    dẫn xuất
m.point::move(6,7);             // Gọi hàm của lớp
    cơ sở
m.display();
//point p;                      // Khai báo đối
    tượng điểm
//m=p;                          // Ok
//coloredpoint n;
//n=p;                          // !Ok
}
```

Kế thừa đơn giản

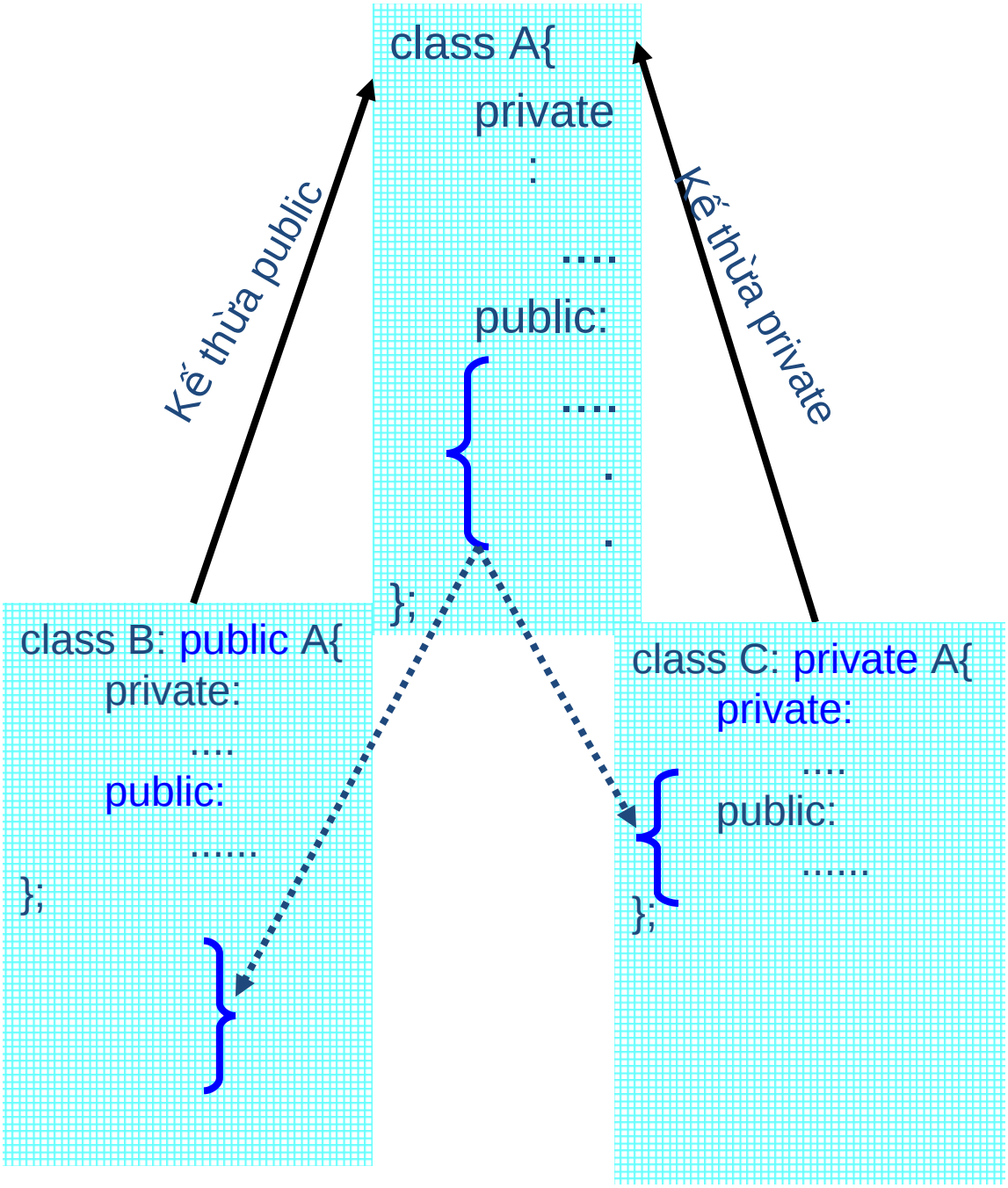
- Truy nhập các thành phần lớp cơ sở:
  - Một lớp dẫn xuất không thể truy nhập các thành phần private của lớp cơ sở.
  - Một lớp dẫn xuất truy nhập được các thành phần public của lớp cơ sở.
  - Việc truy nhập các thành phần của lớp cơ sở từ bên ngoài phạm vi lớp dẫn xuất được quy định bởi từ khoá xác định quyền truy nhập đặt t[illegible]ịnh nghĩa kế thừa lớp dẫn xuất.

```
class coloredpoint : public point
```

Kế thừa đơn giản

- Nếu một lớp dẫn xuất kế thừa lớp cơ sở là public thì mọi thành phần public của lớp cơ sở sẽ trở thành thành phần public của lớp dẫn xuất.
- Nếu một lớp dẫn xuất kế thừa lớp cơ sở là private thì mọi thành phần public của lớp cơ sở sẽ trở thành thành phần private của lớp dẫn xuất.
- Nếu không có từ khoá chỉ định kế thừa từ lớp cơ sở thì lớp dẫn xuất ngầm định là kế thừa private.

Kế thừa đơn giản



Kế thừa đơn giản

- Ví dụ 4.2. Truy nhập thành phần lớp cơ sở.

```
class Base{
    private:
        int x;
    public:
        void set(int i) {
            x = i;
        }
        void display() {
            cout<<“\n x =“<<x;
        }
};
```

Kế thừa đơn giản

```
class Derived1 : public Base{
    public:
        void display() {
            cout<<“\n x =“<<x;
        }
};
class Derived2 : private Base{
    public:
        void display() {
            Base::display();
        }
};
```

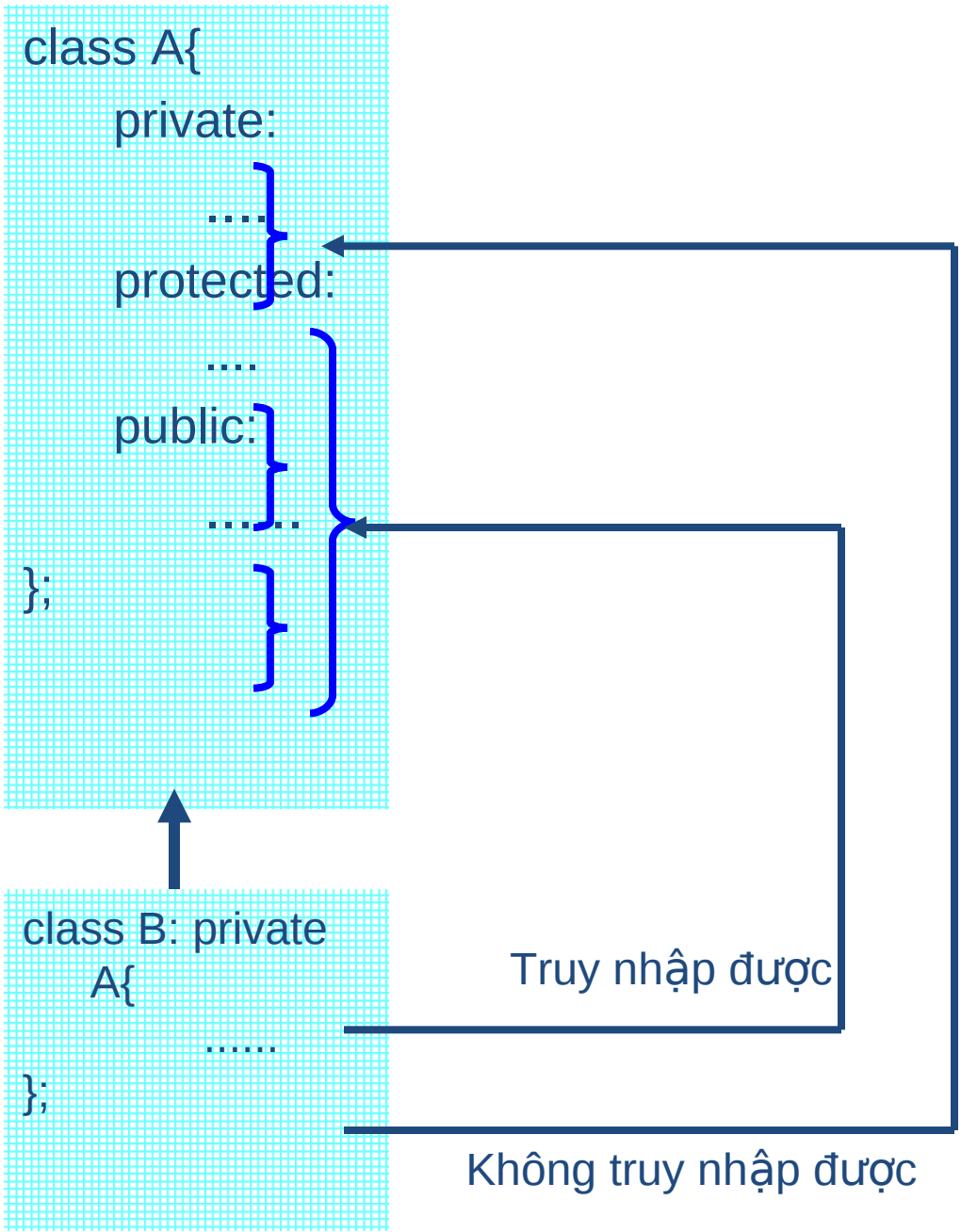
Kế thừa đơn giản

- Khai báo chương trình chính:
  - Derived1 p;
  - Derived2 q;
  - p.set(1);
  - d1.display();
  - q.set(2);
  - q.display();
- Tìm ra các lệnh sai của các đoạn chương trình trên ?.

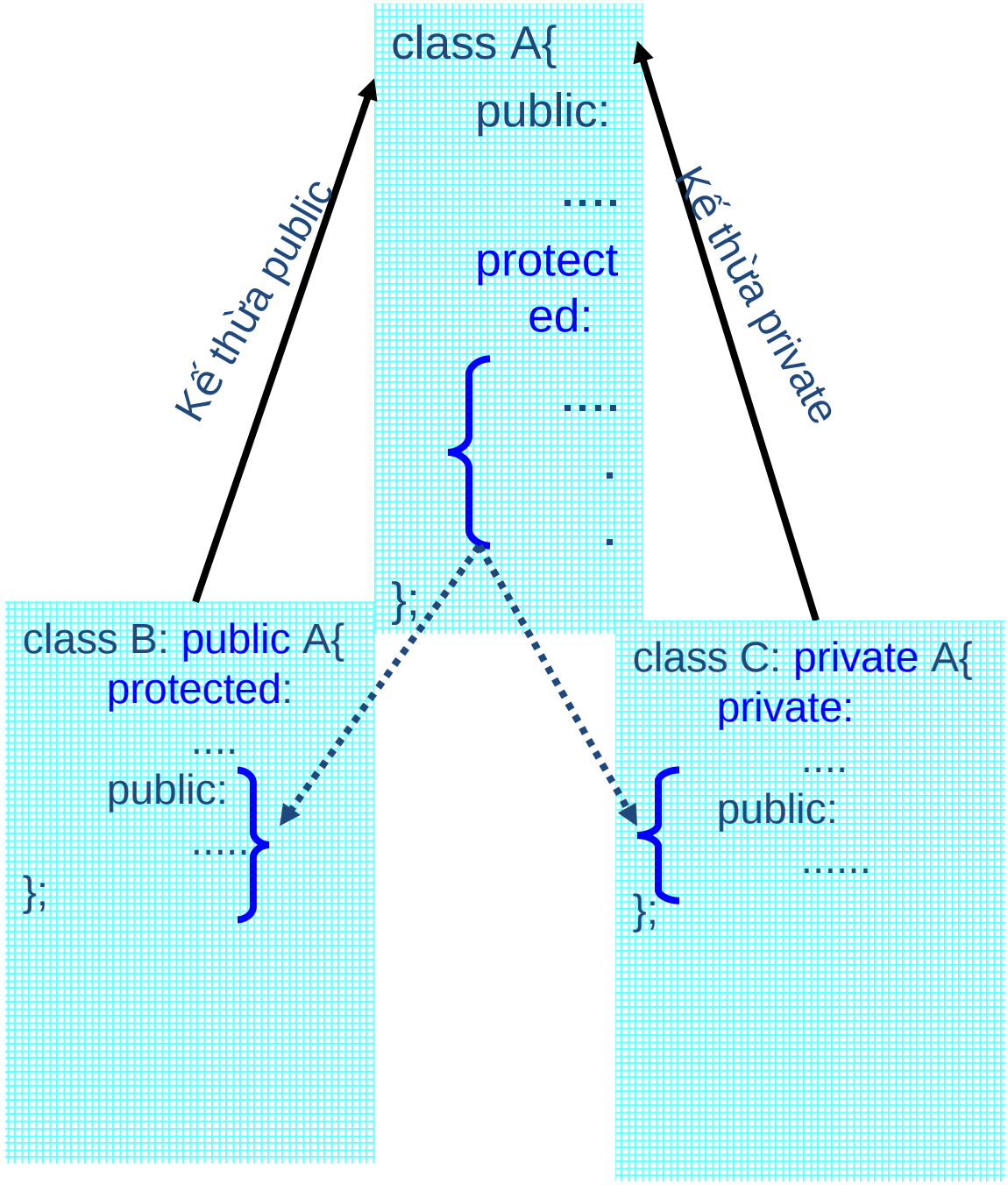
Kế thừa đơn giản

- Từ khoá xác định quyền truy nhập protected
  - Khi cần 1 lớp dẫn xuất truy nhập các thành phần private của lớp cơ sở nhưng không muốn các thành phần của lớp cơ sở là public thì sử dụng từ khoá protected.
  - Thành phần protected của lớp cơ sở truy nhập được trong lớp dẫn xuất nhưng không thể truy nhập được ở các hàm khác, lớp khác.

Kế thừa đơn giản



Kế thừa đơn giản



Kế thừa đơn giản

- Định nghĩa lại hàm thành phần trong lớp dẫn xuất:
  - Sự định nghĩa lại 1 làm thành phần khác với định nghĩa chồng hàm thành phần:
    - Hàm định nghĩa lại và hàm bị định nghĩa lại giống nhau về tên, tham số trả về, chỉ khác nhau là 1 hàm ở lớp cơ sở và một hàm ở lớp dẫn xuất.
    - Hàm chồng chỉ trùng tên, khác nhau về danh sách tham số và chúng đều thuộc cùng 1 lớp.

Kế thừa đơn giản

- Có thể khai báo các thành phần dữ liệu trong lớp dẫn xuất trùng tên với các thành phần dữ liệu đã có trong lớp cơ sở. Để truy nhập thành phần trùng tên của lớp cơ sở trong lớp dẫn xuất phải sử dụng:

  <Tên lớp cơ sở>::<Tên thành phần>

- Hãy đưa ra ví dụ về thành phần trùng tên.

Kế thừa đơn giản

- Tính kế thừa trong lớp dẫn xuất
  - Một đối tượng của lớp dẫn xuất có thể thay thế một đối tượng của lớp cơ sở. Nghĩa là tất cả các thành phần của lớp cơ sở đều tìm thấy trong lớp dẫn xuất.
  - Một đối tượng lớp cơ sở không thể thay thế 1 đối tượng lớp dẫn xuất.
  - Một con trỏ đối tượng lớp cơ sở có thể trỏ đến một đối tượng lớp dẫn xuất.

Kế thừa đơn giản

- Một con trỏ lớp dẫn xuất không thể trỏ đến đối tượng lớp cơ sở trừ trường hợp ép kiểu.
- Một tham chiếu đối tượng lớp cơ sở có thể tham chiếu đến một đối tượng lớp dẫn xuất.
- Một tham chiếu lớp dẫn xuất không thể tham chiếu đến đối tượng lớp cơ sở trừ trường hợp ép kiểu.

Kế thừa đơn giản

- Ví dụ:
  - coloredpoint m(1,2,3);
  - point *p = &m; // Con trỏ lớp cơ sở
  - p->display();
  - point &r = m; // Tham chiếu lớp cơ sở
  - r.move(2,3);

Kế thừa đơn giản

- Hàm thiết lập trong lớp dẫn xuất
  - Lớp dẫn xuất = Lớp cơ sở + thành phần bổ sung.
  - Gọi hàm thiết lập lớp dẫn xuất gồm:
    - Gọi 1 hàm thiết lập lớp cơ sở tạo dữ liệu phần cơ sở.
    - Gọi 1 hàm thiết lập lớp dẫn xuất tạo dữ liệu bổ sung.
  - Lớp dẫn xuất không kế thừa hàm thiết lập lớp cơ sở. Hàm thiết lập lớp dẫn xuất phải chứa thông tin làm tham số cho hàm thiết lập lớp cơ sở. Trong định nghĩa hàm thiết lập lớp dẫn xuất phải gọi

Kế thừa đơn giản

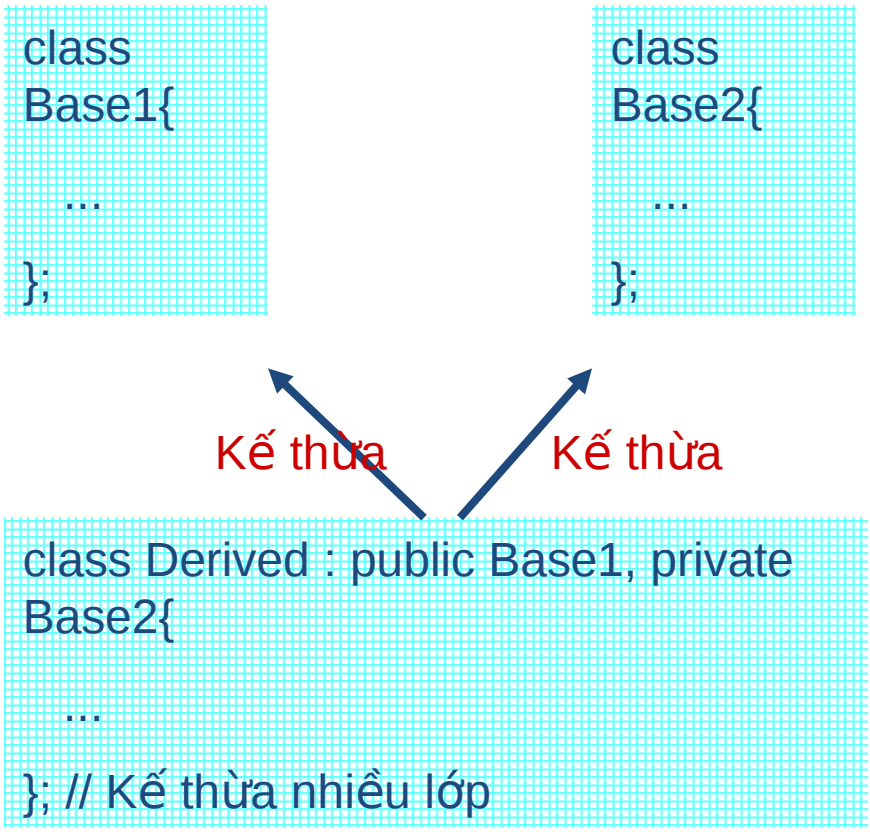
- Hàm thiết lập lớp dẫn xuất

```
coloredpoint():point() {
    color=0;
}
coloredpoint(float ox, float oy, int
  c):point(ox, oy) {
    color=c;
}
coloredpoint(coloredpoint
  &b):point((point &) b) {
    color=b.color;
}
```

- Hàm huỷ bỏ tương tự hàm thiết lập

Kế thừa nhiều lớp

- Một lớp có nhiều lớp cơ sở gọi là kế thừa nhiều lớp.



Kế thừa nhiều lớp

- Khi kế thừa nhiều lớp, có thể có các thành phần của lớp cơ sở giống nhau, để chỉ định truy nhập đến thành phần của lớp nào, phải sử dụng: <Tên lớp>::<Tên thành phần>;
- Ví dụ:
  - Trong lớp Base1 có hàm thành phần Set()
  - Trong lớp Base2 có hàm thành phần Set()
  - Khi đó trong lớp Derived có 2 hàm Set(). Để truy nhập hàm Set của lớp Base1, viết Base1::Set().

Kế thừa nhiều lớp

- Nếu một lớp có nhiều lớp cơ sở, các tham số của hàm thiết lập cho tất cả các lớp cơ sở này có thể nêu ra trong hàm thiết lập lớp dẫn xuất.
- Nếu một lớp cơ sở định nghĩa một hàm thiết lập không tham số hoặc định nghĩa 1 hàm thiết lập mà mọi tham số có giá trị ngầm định thì hàm thiết lập lớp dẫn xuất không nhất thiết gọi đến hàm thiết lập của lớp cơ sở.

Kế thừa nhiều lớp

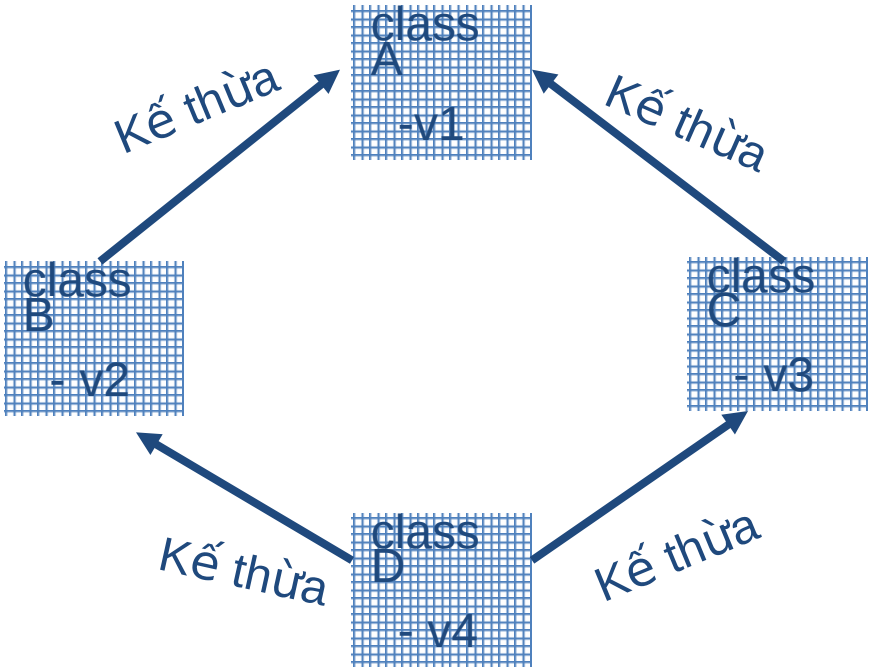
```
class Base1{
    private:
        float x,y;
    public:
        Base1(float
          ox, float ox)
          {
          x=ox; y=oy;
        }
        ...
};
```

```
class Base2{
    private:
        float x;
    public:
        Base2(float
          s=0){
            x=s;
        }
        ...
};
```

- Hàm thiết lập của lớp dẫn xuất từ lớp Base1 và Base2 bắt buộc gọi hàm thiết lập lớp Base1 nhưng không cần gọi hàm thiết lập lớp Base2.

Lớp cơ sở ảo - virtual class

- Không thể khai báo hai lần cùng 1 lớp trong danh sách của các lớp cơ sở cho 1 lớp dẫn xuất. Điều này sẽ sinh ra lỗi vì không phân biệt được lớp cơ sở gốc.



Lớp cơ sở ảo - virtual class

- Ta có thể nói D kế thùa A hai lần.Trong tình huống này, các thành phần của A (hàm hoặc dữ liệu) sẽ xuất hiện trong D hai lần. Đối với hàm thành phần thì điều này không quan trọng vì chỉ có duy nhất một hàm cho một lớp cơ sở. 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

Lớp cơ sở ảo - virtual class

- Thông thường chúng ta không muốn dl bị lặp lại và phải giải quyết bằng cách chọn một trang hai bản sao dl để thao tác. Điều này thật chán 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)

Lớp cơ sở ảo

```
#include<iostream.h>
#include<conio.h>
class A{
      public:
            float v1;
};
class B : public A{
      public:
            float v2;
};
```

```
class C : public A{
      public:
            float v3;
};
class D : public B, public
      C{
      public:
            float v4;
};
void main(){
      D x;
      x.v1=2;          // Xẩy
      ra lỗi
};
```

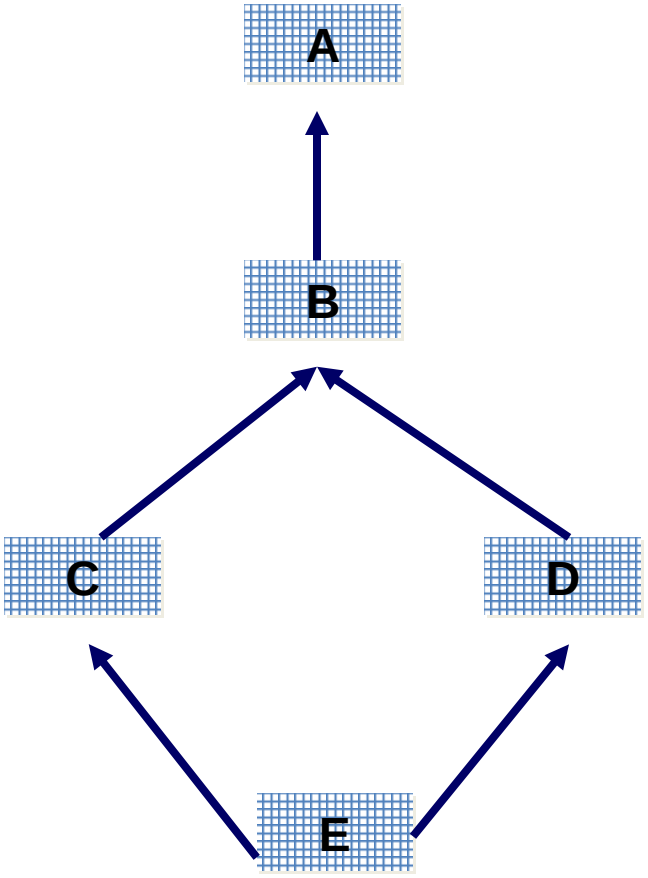
Lớp cơ sở ảo

- Giải pháp cho vấn đề này là khai báo A là 1 lớp cơ sở kiểu virtual cho cả lớp B và lớp C. Định nghĩa của lớp B và C như sau:

```
class B : virtual public A{
        public:
                float v2;
};
class C : virtual public A{
        public:
                float v3;
};
```

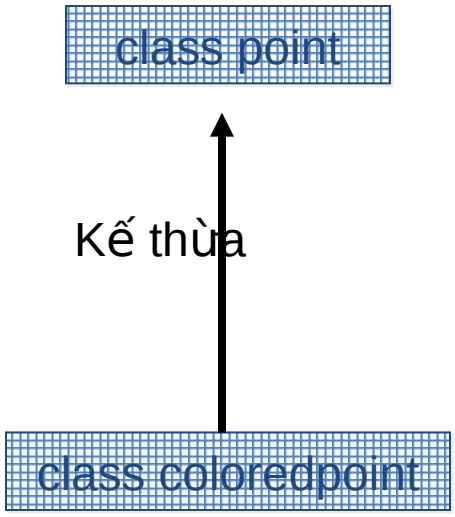
Lớp cơ sở ảo

- Cho sơ đồ kế thừa sau, tìm các lớp cơ sở ảo:



Tính tương ứng bội(tính đa hình)

- Tính tương ứng bội (polymorphism) là khả năng xử lý các lớp liên hệ với nhau 1 cách tổng quát.
- Xét ví dụ sau:



Tính tương ứng bội x như thế nào ???

Tính tương ứng bội

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

Tính tương ứng bội

```
class coloredpoint : public point{
    private:
        int color;
    public:
        coloredpoint(float ox=0, float oy=0,
          int c=0):point(ox,oy){
            color = c;
        }
        virtual void display(){
            point::display();
            cout<<“ color =“<<color;
        }
};
```

Tính tương ứng bội

```
void main(){
    coloredpoint m(1,2,3);     //  Khai báo đối tượng
       coloredpoint.
    m.display();                // Gọi hàm display của
       coloredpoint.
    point p(4,5);        // Khai báo đối tượng point.
    p.display();                // Gọi hàm display của
       lớp point.
    point *ptr;                 // Khai báo con tro và
       trỏ point.
    ptr = &p;                   // Cho ptr trỏ đến p.
    ptr->display();             // Gọi hàm display của
       lớp point.
    ptr = &m;                   // Cho ptr trỏ đến m.

    ptr->display();             // Gọi hàm display của
       coloredpoint.
    getch();
}
```

Tính tương ứng bội

- Trong định nghĩa lớp point, hàm display() có từ khoá virtual để chỉ rằng nó là hàm ảo.
- Từ khoá virtual có thể đặt trước hoặc sau tên kiểu dữ liệu.
- Hàm display() được định nghĩa lại trong lớp dẫn xuất. Từ khoá virtual trước hàm display() của lớp coloredpoint không cần thiết phải có.

Tính tương ứng bội

- Tính tương ứng bội đã xẩy ra:
  - Tuỳ thuộc vào kiểu đối tượng có địa chỉ chứa trong con trỏ ptr mà lời gọi hàm ptr->display() sẽ gọi đến hàm display() của lớp point hay lớp coloredpoint.
  - Tính tương tứng bội còn thể hiện khi một hàm thành phần trong lớp cơ sở được gọi từ 1 đối tượng của lớp dẫn xuất, còn bản thân hàm đó thì gọi tới hàm thành phần được định nghĩa đồng thời trong lớp cơ sở và lớp dẫn xuất.

Tính tương ứng bội

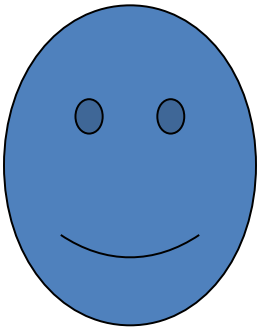
```
class point{
    private:
        float x, y;
    public:
        point(float ox =0, float oy=0){
            x=ox; y=oy;
        }
        void display(){
            cout<<“\n x =“<<x<<“ y =“<<y;
            displaycolor();
        }
        virtual void displaycolor() {}// Hàm
          rỗng
};
```

Tính tương ứng bội

```
class coloredpoint : public point{
    private:
        int color;
    public:
        coloredpoint(float ox=0, float oy=0,
          int c=0):point(ox,oy){
            color = c;
        }
        virtual void displaycolor (){
            cout<<“ color =“<<color;
        }
};
```

Tính tương ứng bội

```
void main(){
        coloredpoint m(1,2,3);
        m.display();
        point p(4,5);
        p.display();
        point *ptr;
        ptr = &p;
        ptr->display();
        ptr = &m;
        ptr->display();
        getch();
}
```



Tuỳ thuộc vào đối tượng có địa chỉ ở con trỏ ptr mà hàm display() sẽ gọi đến hienthi() của lớp point hay hienthi() của lớp coloredpoint. Đó là tính tương ứng bội !!!

Tính tương ứng bội

- Những đặc trưng của hàm virtual
  - Tất cả các hàm virtual ở lớp cơ sở và lớp dẫn xuất phải được định nghĩa có cùng tên, cùng danh sách tham số, cùng kiểu trả về. Nếu các kiểu của hàm khác nhau, từ khoá virtual sẽ bị bỏ qua và chương trình dịch sẽ hiểu rằng lớp dẫn xuất đã gạt hàm này.
  - Không bắt buộc phải ghi rõ từ khoá virtual khi định nghĩa hàm virtual trong lớp dẫn xuất.

Tính tương ứng bội

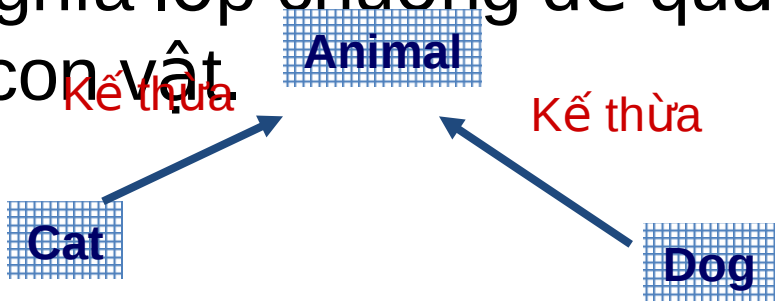
- Vì các hàm virtual dựa trên lớp nguyên thuỷ của đối tượng mà qua đó chúng được gọi, nên chúng phải có 1 đối tượng ẩn và do đó chúng phải là những hàm thật sự của lớp. Điều này nghĩa là các hàm friend không thể là hàm virtual được. Tuy nhiên 1 hàm virtual của lớp có thể được khai báo là friend trong một lớp khác.
- Nếu lớp dẫn xuất không định nghĩa hàm tương ứng bội thì hàm đã định nghĩa cho lớp cơ sở sẽ được sử dụng.

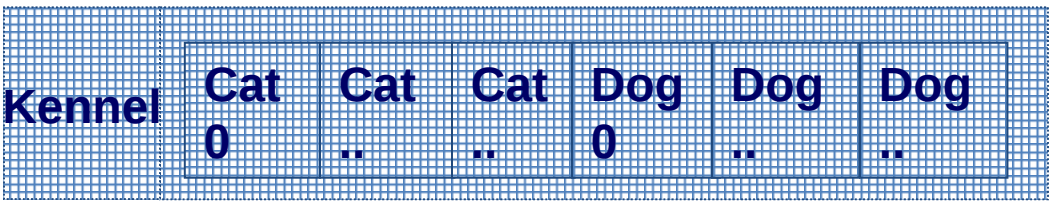
Tính tương ứng bội

- Bài tập
  - Cô jody có 1 cái chuồng 20 ngăn nuôi những con thú. Cô thường nhốt những con mèo vào một nữa số ngăn và những con chó vào nữa số ngăn còn lại. Hãy xây dựng 1 chương trình quản lý các con thú gồm:
    - Nhập các con thú vào các chuồng.
    - Lấy con thú ra khỏi các chuồng.
    - Hiển thị tên các con thú và số hiệu chuồng của nó.

Tính tương ứng bội

- Có bốn lớp cần định nghĩa cho bài toán:
  - Định nghĩa lớp con vật.
  - Định nghĩa lớp con mèo kế thừa từ lớp con vật.
  - Định nghĩa lớp con chó kế thừa từ lớp con vật.
  - Định nghĩa lớp chuồng để quản lý các con vật.





Tính tương ứng bội

- Định nghĩa lớp con vật:

```
class Animal{
    protected:
        char *Name;
    public:
        Animal(){Name = null;}
        Animal(char *n){ Name = strdup(n);}
        ~Animal(){delete Name}
        void Display(){
            cout<<“\n Con vat chung”;
        }
};
```

Tính tương ứng bội

- Lớp Animal là lớp cơ sở cho lớp Dog và lớp Cat. Khi một đối tượng Animal được tạo ra, nó được đặt 1 tên để lưu trữ cùng với đối tượng này. Thành phần dữ liệu Name được khai báo là protected, như vậy nó có thể được truy nhập ở các lớp dẫn xuất.

Tính tương ứng bội

- Định nghĩa lớp con mèo:

```
class Cat : public Animal {
    public:
        Cat() : Animal(){}
        Cat(char *n) : Animal(n){}
        void Display(){
            cout<<“\n Con meo ten la:”<<Name;
        }
};
```

Tính tương ứng bội

- Định nghĩa lớp con chó:

```
class Dog : public Animal {
    public:
        Dag() : Animal(){}
        Dag(char *n) : Animal(n){}
        void Display(){
            cout<<“\n Con cho ten la:”<<Name;
        }
};
```

Tính tương ứng bội

- Lớp Cat và lớp Dog rất giống nhau. Mỗi lớp chỉ định nghĩa những hàm thiết lập gửi tham số cho hàm thiết lập của lớp cơ sở Animal.
- Hai lớp đều có phương thức Display() để hiển thị loại và tên của đối tượng.

Tính tương ứng bội

- Định nghĩa lớp chuồng:

```
class Kennel{
    private:
        int MaxCats;        // Số con mèo tối đa.
        int NumCats;        // Số con mèo có trong chuồng.
        Cat  **Kitties;     // Mảng con trỏ chứa các con mèo.

        int MaxDogs;        // Số con chó tối đa.
        int NumDogs;        // Số con chó có trong chuồng.
        Dog  **Doggies;     // Mảng con trỏ chứa các con chó.
```

Tính tương ứng bội

```
    public:
        Kennel(int maxc, int maxc);
        ~Kennel();
        // Phương thức đưa 1 con thú vào
           chuồng.
        int Accept(Dog *d);
        int Accept(Cat *c);
        // Phương thức lấy một con thú ra khỏi
           chuồng pen.
        Dog *ReleaseDog(int pen);
        Cat  *ReleaseCat(int pen);
        // Phương thức hiển thị các con thú
           trong chuồng.
        void ListAnimal();
};
```

Tính tương ứng bội

- Các phương thức cho lớp Kennel quá phức tạp, không nên dùng hàm inline, do đó chúng được định nghĩa ngoài lớp.
- Về cơ bản, lớp Kennel có chứa 2 con trỏ trỏ đến các đối tượng Dog và các đối tượng Cat.
- Tiếp theo, định nghĩa các phương thức của lớp Kennel:

Tính tương ứng bội

- Hàm thiết lập cho lớp Kennel nhận hai tham số định nghĩa số lượng lớn nhất các con mèo và các con chó có thể chứa.

```
Kennel::Kennel(int maxc, int maxd){
        MaxCats  = maxc;
        MaxDogs = maxd;
        NumCats  = 0;
        NumDogs = 0;
        Kitties    = new Cat *[MaxCats];   //
          Mảng con trỏ
        Doggies = new Dos *[MaxDogs];  //
          Mảng con trỏ
        for (int i=0; i<MaxCats; i++) Kitties[i]
          = NULL;
        for (int j=0; j<MaxDogs; j++)
          Doggies[j] = NULL;
};
```

Tính tương ứng bội

- Hàm huỷ bỏ cho lớp Kennel xoá các con trỏ đã cấp phát:

```
Kennel::~Kennel() {
    delete Kitties;
    delete Doggies;
};
```

Tính tương ứng bội

- Có hai phương thức có tên là Accept, một nhận tham số là 1 con trỏ trỏ đến Dog và một nhận con trỏ trỏ đến Cat. Đây là ví dụ về định nghĩa chồng hàm:

```
int Kennel::Accept(Dog *d) {
    if (NumDogs == MaxDogs) return 0;
    NumDogs++;
    int i=0;
    while (Doggies[i] !=NULL) i++;
    Doggies[i]=d;
    return i+1;
};
```

Tính tương ứng bội

```
int Kennel::Accept(Cat *d) {
    if (NumDogs == MaxDogs) return 0;
    NumDogs++;
    int i=0;
    while (Doggies[i] !=NULL) i++;
    Doggies[i]=d;
    return i+1;
};
```

- Phương thức Accept sẽ lưu con trỏ nhận ở đối số vào mảng cho loại thú tương ứng. Nếu chuồng còn trống, trả về số chuồng (pen) mà con thú được nhốt. Ngược lại trả về 0.

Tính tương ứng bội

- Phương thức thả 1 con chó ra khỏi chuồng:

```
Dog *Kennel::ReleaseDog(int pen){
        if ((pen > MaxDogs) return NULL;
        pen --;
        if (Doggies[pen] !=NULL){
         Dog *temp = Doggies[pen];
         Doggies[pen] = NULL;
         NumDogs --;
         return temp;
        }
        else
         return NULL;
};
```

Tính tương ứng bội

- Phương thức thả 1 con mèo ra khỏi chuồng:

```
Cat *Kennel::ReleaseCat(int pen){
        if ((pen > MaxCats) return NULL;
        pen --;
        if (Kitties[pen] !=NULL){
         Cat *temp = Kitties[pen];
         Kitties[pen] = NULL;
         NumCats --;
         return temp;
        }
        else
         return NULL;
};
```

Tính tương ứng bội

```
void Kennel::ListAnimals(){
      if (NumDogs>0)
            for (int i=0; i<MaxDogs; i++)
                  if (Doggies[i] !=NULL){
                        cout<<“\n Con cho trong chuong “<<
                           i;
                        Doggies->Display();
                  }
      if (NumCats>0)
            for (int i=0; i<MaxCats; i++)
                  if (Kitties[i] !=NULL){
                        cout<<“\n Con meo trong chuong
                           “<< i;
                        Kitties->Display();
                  }
};
```

Tính tương ứng bội

- Chương trình chính

```
void main(){
    Kennel
        K(10,10)
        ;
    Dog
        d1("a");
    Dog
        d2("b");
    Dog
        d3("c");
    Cat c1("x");
    Cat c2("y");
    Cat c3("z");
    K.Accept(&d1);
    K.Accept(&d2);
    K.Accept(&d3);
    K.Accept(&c1);
    K.Accept(&c2);
    K.ListAnimal();
    }
```

Tính tương ứng bội

- Phân tích chương trình đã viết:
  - Giả sử cô Jody cần nuôi 15 con lợn và 5 con mèo, cô phải thay đổi khai báo của chương trình.
  - Cô jody cần nuôi thêm những con lợn (Pig) vào chuồng, khi đó cô ta cần định nghĩa thêm lớp con lợn và sửa đổi lại lớp Kennel để đưa thêm lớp mới, tức là phải thêm các thành phần dữ liệu mới, thay đổi các hàm thiết lập, thêm các phương thức mới. Rõ ràng cần làm lại chương trình này.
  - Để xây dựng được bài toán dạng tổng quát, sử dụng tính tương ứng bội.

Tính tương ứng bội

- Xây dựng lớp Animal:

```
class Animal{
    protected:
        char *Name;
    public:
        Animal(){Name = NULL;}
        Animal(char *n){ Name = strdup(n);}
        ~Animal(){ delete Name;}
        virtual Display(){}
};
```

Tính tương ứng bội

- Xây dựng lớp Cat kế thừa từ lớp Animal:

```
class Cat : public Animal{
    public:
        Cat() : Animal(){}                  // Hàm rỗng
        Cat(char *n) : Animal(n){}          // Hàm rỗng
        virtual void Display(){
            cout<<“\n Con meo ten :”<<Name;
        }
};
```

Tính tương ứng bội

- Xây dựng lớp Dog kế thừa từ lớp Animal:

```
class Dog : public Animal{
    public:
        Dog() : Animal(){}              // Hàm rỗng
        Dog(char *n) : Animal(n){}      // Hàm rỗng
        virtual void Display(){
            cout<<“\n Con cho ten :”<<Name;
        }
};
```

Tính tương ứng bội

- Xây dựng lớp chuồng:

```
class Kennel{
    private:
        int MaxAnimals;              // Số con vật tối đa.
        int NumAnimals;              // Số con vật hiện có.
        Animal **Resident;           // Mảng chứa các con thú.
    public:
        Kennel(int max);             // Hàm thiết lập.
        ~Kennel();                   // Hàm huỷ bỏ.
        int Accept(Animal *d);       // Hàm nhốt 1 con thú.
        Animal *Release(int pent);   // Hàm thả 1 con thú.
        void ListAnimals();          // Hàm hiển thị.
};
```

Tính tương ứng bội

- Định nghĩa hàm thiết lập và hàm huỷ bỏ:

```
Kennel::Kennel(int max) {
    MaxAnimals = max;
    NumAnimals = 0;
    Resident = new Animal
      *[MaxAnimals];
    for (int i=0; i< MaxAnimals; i++)
        Resident[i] = NULL;
}
~Kennel() {
    delete Resident;
}
```

Tính tương ứng bội

- Định nghĩa hàm nhốt một con thú:

```
int Kennel::Accept(Animal *d) {
    if (NumAnimals ==MaxAnimals)
      return 0;
    NumAnimals++;
    int i=0;
    while (Resident[i] !=NULL) i++;
    Resident[i] = d;
    return i+1;
}
```

Tính tương ứng bội

- Định nghĩa hàm thả một con thú:

```
Animal *Kennel::Release(int pen) {
    if (pen>MaxAnimals) return NULL;
    pen--;
    if (Resident[pen] !=NULL){
        Animal *temp = Resident[pen];
        Resident[pen]=NULL;
        NumAnimals --;
        return temp;
    }
    else return NULL;
}
```

Tính tương ứng bội

- Định nghĩa hàm hiển thị các con thú:

```
void Kennel::ListAnimals() {
    if (NumAnimals >0)
        for (int i = 0; i<MaxAnimals; i++)
            if (Resident[i] !=NULL){
                cout<<"\n Con thu o chuong "<< i;
                Resident[i]->Display();
            }
    }
```

- Tự sinh viên xây dựng chương trình chính.

Tính tương ứng bội

- Có hai thay đổi cần chú ý:
  - Hầu hết các vấn đề trong phiên bản trước nảy sinh từ việc xử lý riêng lẽ các đối tượng Cat và Dog.
  - Phiên bản này định nghĩa Cat và Dog để chúng có thể được xử lý giống như là có liên quan với nhau.
  - Tính tương ứng bội đã xảy ra ở hàm Display(). Từ khoá virtual trong định nghĩa hàm Dispay() ở lớp Dog và lớp Cat là không quan trọng.

Tính tương ứng bội

- Vấn đề tiếp theo là làm thế nào để thêm các con lợn ?.
  - Việc thêm vào các con lợn chỉ cần thêm lớp:

```
class Pig : Animal{
    public:
        Pig() : Animal(){}
        Pig(char *n) : Animal(n){};
        virtual Display(){
        cout<<“\n Con lon ten la :”<< Name;
        }
};
```

Tính tương ứng bội

- Nếu cô Jody muốn tách con theo thành mèo đực và mèo cái, cô ta làm thế nào ?
  - Thêm lớp mèo cái như sau:

```
class FemaleCat : public Cat{
    public:
        FemaleCat(): Cat(){}
        FemaleCat(char *n): FemaleCat(n){}
        virtual void Display(){
        cout<<“\n Meo cai ten “<<Name;
        }
};
```

Tính tương ứng bội

- Tương tự đối với lớp mèo đực.

- Thế mạnh của tương ứng bội gồm:
  - Xử lý các khái niệm có liên hệ với nhau theo một cáh giống nhau, làm cho chương trình tổng quát hơn.
  - Tính tương ứng bội cũng có thể dùng để viết những chương trình có thể mở rộng. Khi một loại mới được thêm vào có liên hệ với các lớp đang có bản chất tương ứng bội sẽ làm cho nó thích ứng với hệ thống mà cần không thay đổi hệ thống.

Các lớp cơ sở trừu tượng

- Một lớp cơ sở trừu tượng (Abstract Base Class) là một lớp chỉ được dùng làm cơ sở cho lớp khác. Không hề có đối tượng nào của 1 lớp trừu tượng được tạo ra vì nó chỉ được dùng để định nghĩa 1 khái niệm tổng quát cho các lớp khác.
- Lớp trừu tượng thường được áp dụng cho các hàm virtual thuần túy.

Các lớp cơ sở trừu tượng

- Một hàm virtual thuần túy là 1 hàm mà trong định nghĩa lớp nó được định nghĩa “không có gì cả”.
- Ví dụ:

```
class Abstract{
    public:
        void Print() =0;
        void Process()=0;
        int Status();
};
```

Các lớp cơ sở trừu tượng

- Hàm Print() và Process() được khai báo là các hàm virtual thuần túy bằng cách gán bằng 0 thay cho việc định nghĩa hàm này.
- Hàm Status() là 1 hàm thành phần bình thường và sẽ có 1 định nghĩa ở đâu đó.
- Không có 1 đối tượng nào của 1 lớp trừu tượng được tạo ra, tuy nhiên các con trỏ và tham chiếu đến các đối tượng của lớp trừu tượng thì vẫn hợp lệ.

Các lớp cơ sở trừu tượng

- Bất kỳ lớp nào dẫn xuất từ 1 lớp cơ sở trừu tượng phải khai báo lại tất cả các hàm virtual thuần túy mà nó thừa hưởng.
- Một lớp dẫn xuất phải định nghĩa lại tất cả các hàm virtual thuần túy mà nó thừa hưởng, hoặc bằng các hàm virtual hoặc bằng định nghĩa hàm thực sự.

Các lớp cơ sở trừu tượng

- Ví dụ:

```
class Derived : public Abstract{
  public:
      void Print() =0;
      void Process(){
          //Định nghĩa của lớp Derived cho
            hàm này.
      }
  }
```

  - Lớp Animal định nghĩa trong bản thứ 2 là 1 lớp trừu tượng nếu thay đổi hàm Display() thành 1 hàm virtual thuần túy.

Tóm tắt

- Khái niệm kế thừa
- Đơn kế thừa
- Đa kế thừa
- Tính tương ứng bội
- Bài tập

Chapter 5.Template

IT Faculty, Vinh University

Mục đích

- Giới thiệu về việc sử dụng mô hình xây dựng các bài toán tổng quát gồm:
- Mô hình hàm
- Mô hình lớp

Nội dung

- Mô hình hàm
  - Định nghĩa và sử dụng
  - Giới hạn của mô hình hàm
  - Cụ thể hóa mô hình hàm
- Mô hình lớp
  - Định nghĩa và sử dụng
  - Giới hạn của mô hình lớp
  - Cụ thể hóa mô hình lớp

Mô hình hàm

- Trong lập trình nhiều khi gặp một loạt các hàm giống nhau về giải thuật, chỉ khác nhau về kiểu dữ liệu. Để tránh viết lặp lại các giải thuật, ta xây dựng mô hình hàm.
- Ví dụ 5.1. Hàm tìm max cho số nguyên, thực:

```
int max(int a, int b) {
        return (a>b) ? a:b;
}
float max(float a, float b) {
        return (a>b) ? a:b;
}
```

Mô hình hàm

- Hai hàm này chỉ khác nhau điểm duy nhất là kiểu dữ liệu.
- Mô hình hàm cho phép định nghĩa một mô hình giải thuật chung cho hàm max bằng kiểu dữ liệu là tên 1 lớp trung gian. Tên lớp trung gian này sẽ được thay thế bằng kiểu dữ liệu cụ thể khi gọi mô hình.

Mô hình hàm

- Định nghĩa mô hình hàm:

```
template <class T>
<type> <tên hàm>(Các tham số) {
    <Nội dung hàm>
}
```

  - Ví dụ:

```
template <class T>
T max(T a, T b) {
    return (a>b) ? a:b;
}
```

Mô hình hàm

- Khai báo template <class T> có nghĩa T là tên lớp của mô hình. T sẽ được thay thế bằng kiểu dữ liệu cụ thể như int, float,.. khi gọi mô hình.
- Gọi mô hình hàm giống như gọi hàm bình thường.
- Khi gọi mô hình hàm max với tham số truyền vào, chương trình dịch nhận biết kiểu dữ liệu truyền vào và sinh ra 1 hàm cụ thể.

Mô hình hàm

```
#include <iostream.h>
#include <conio.h>
template <class T>
T max(T a, T b){
	return (a>b) ? a:b;
}
void main(){
	int a=2, b=3;
	cout <<“\n Max cua a va b =“<<max(a,b);  //
	    T = int
	float x=5.2, y=3;
	cout <<“\n Max cua x va y =“<<max(x,y);  //
	    T = float
	cout <<“\n Max cua a va x =“<<max(a,x);  //
	    Error
	cout<<“\n Max cua 2 ky tu = “<<max(‘a’,’c’);
}
```

Mô hình hàm

- Ngoài kiểu dữ liệu chuẩn (int, float, char,...), mô hình cũng có thể ứng dụng cho các kiểu dữ liệu của người sử dụng.
- Giả sử có lớp phân số và trong lớp này có định nghĩa toán tử > là toán tử được sử dụng trong mô hình hàm max thì có thể gọi max(a,b) với a và b là các phân số.
- Có thể có nhiều hơn 1 lớp làm lớp mô hình.

Mô hình hàm

- Bài tập
  - Xây dựng 1 lớp phân số gồm:
    - Hàm nhập 1 phân số.
    - Hàm in 1 phân số
    - Định nghĩa toán tử >
  - Xây dựng 1 mô hình hàm max
  - Viết chương trình:
    - Nhập vào một mảng n phân số, tìm và in ra phân số lớn nhất.
    - Nhập vào một mảng n số thực, tìm và in ra số lớn nhất.

Mô hình hàm

- Giới hạn của mô hình hàm:
  - Các tham số truyền vào cho mô hình hàm phải đảm bảo sao cho trình biên dịch ánh xạ 1-1 trong việc thay thế các lớp mô hình bởi kiểu dữ liệu thực.
  - Ví dụ: Lời gọi hàm max(a,x) với a là kiểu int, x là kiểu float sẽ gây ra lỗi vì chương trình dịch không biết thay tế T bởi int hay float.
  - Mô hình hàm chỉ áp dụng được cho các lớp dữ liệu mà có các hàm, các toán tử, hàm thiết lập được sử dụng trong mô hình.

Mô hình hàm

- Cụ thể hoá mô hình hàm:
  - Cụ thể hoá mô hình hàm là định nghĩa hàm trùng tên cho các kiểu dữ liệu đặc biệt mà thuật toán của nó không tuân theo mô hình chung.
  - Xét mô hình hàm max với dữ liệu kiểu xâu:

    char *s1 = “SPTIN”, *s2 = “CNTIN”;

    Theo mô hình, việc so sánh 2 xâu là dựa trên toán tử >. Như vậy là so sánh địa chỉ 2 xâu mà không phải so sánh nội dung 2 xâu. Cần phải cụ

Mô hình hàm

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
template <class T>
T max(T a, T b){
	return (a>b) ? a:b;
}
//Cụ thể hoá mô hình cho dữ liệu kiểu xâu.
char *max(char *s1, char *s2){
	return (strcmp(s1,s2)>0) ? s1:s2;
}
```

Mô hình hàm

```
void main() {
        int a = 2, b=3;
        cout<<“\n Max cua a va
            b:”<<max(a,b);       // Gọi mô hình
        float x = 5.2, b=3;
        cout<<“\n Max cua x va y:”<<max(x,y);
                // Gọi mô hình
        char *s1 = “SPTIN”;
        char *s2 = “CNTIN”;
        cout<<“\n Max cua s1 va s2:”<<max(s1,s2);
            //Gọi hàm cụ thể
        getch();
}
```

Mô hình lớp

- Mô hình lớp cho phép xây dựng một mô hình chung cho các kiểu dữ liệu sau đó áp dụng mô hình lớp cho các kiểu dữ liệu cụ thể để được các lớp cụ thể.
- Ví dụ:
  - Giả sử có 1 lớp các điểm trên mặt phẳng, phụ thuộc vào mặt phẳng là rời rạc hay liên tục mà định nghĩa các thành phần dữ liệu có kiểu int hay double.

Mô hình lớp

- Lớp các điểm thuộc mặt phẳng rời rạc:

```
class point{
    private:
        int x, y;
    public:
        point(int ox = 0, int oy = 0){
            x=ox; y=oy;
        }
        void move(int dx, int dy){
            x+=dx; y+=dy;
        }
        void display(){
            cout<<“\n x= “<<x<<“ y =“<<y;
        }
};
```

Mô hình lớp

- Lớp các điểm thuộc mặt phẳng liên tục:

```
class point{
    private:
        double x, y;
    public:
        point(double ox = 0, double oy = 0){
            x=ox; y=oy;
        }
        void move(double dx, double dy){
            x+=dx; y+=dy;
        }
        void display(){
            cout<<“\n x= “<<x<<“ y =“<<y;
        }
};
```

Mô hình lớp

- Hai lớp này chỉ khác nhau điểm duy nhất là kiểu dữ liệu, do đó có thể định nghĩa 1 mô hình lớp.
- Để định nghĩa một mô hình lớp, ta sử dụng từ khoá template giống như mô hình hàm.
- Định nghĩa mô hình lớp point như sau:

Mô hình lớp

```
template <class T>
class point{
        private:
                T x, y;
        public:
                point(T ox = 0, T oy = 0){
                        x=ox; y=oy;
                }
                void move(T dx, T dy);
                }
                void display(){
                        cout<<“\n x= “<<x<<“ y =“<<y;
                }
};
```

Mô hình lớp

- Hàm thành phần của lớp định nghĩa trong lớp giống như định nghĩa hàm thông thường.
- Hàm thành phần định nghĩa ngoài lớp, phải nhắc lại từ khoá: template <class T>.
- Hàm move(T dx, T dy) định nghĩa ngoài lớp.

```
template <class T>
void point<T>::move(T dx, T dy) {
    x+=dx; y+=dy;
}
```

Mô hình lớp

- Khai báo đối tượng với lớp thể hiện kiểu int:
  - point<int> p(4,5);
  - p.display();
- Khai báo đối tượng với lớp thể diện kiểu double:
  - point<double> q(3.5,2.3);
  - q.display();

Mô hình lớp

- Giới hạn của mô hình lớp
  - Mô hình lớp chỉ áp dụng cho các lớp dữ liệu mà có các hàm thành phần, các toán tử, hàm thiết lập được sử dụng trong mô hình lớp.
  - Ví dụ mô hình lớp point đã sử dụng các toán tử += và << do đó mô hình lớp point chỉ áp dụng được với các lớp mà có các toán tử đã sử dụng trong mô hình.

Mô hình lớp

- Cụ thể hoá mô hình lớp
  - Trong mô hình lớp có thể cụ thể hoá một số hàm thành phần hoặc cả lớp.
  - Cụ thể hoá hàm thành phần: Khi một số hàm thành phần của lớp không tuân theo mô hình chung thì phải cụ thể hoá các hàm đó.
  - Cụ thê hoá lớp: Khi có một kiểu dữ liệu mà định nghĩa lớp không tuân theo mô hình chung thì có thể cụ thể cả lớp cho kiểu dữ liệu đó.

Tóm tắt

- Mô hình hàm
  - Định nghĩa và sử dụng
  - Giới hạn của mô hình hàm
  - Cụ thể hoá mô hình hàm
- Mô hình lớp
  - Định nghĩa và sử dụng
  - Giới hạn của mô hình lớp
  - Cụ thể hoá mô hình lớp

Object Oriented Programming

Faculty of Information Technology

Vinh University

Mục đích

- Mục đích chuyên môn:

  Trang bị cho sinh viên phương pháp và kỹ thuật lập trình hiện đại, đó là phương pháp lập trình hướng đối tượng để phát triển những phần mềm có chất lượng, có tính mở và có khả năng đáp ứng những yêu cầu hay thay đổi của phần mềm.

- Mục đích năng lực:
  - Cơ sở: Cài đặt các bài toán lập trình hướng đối tượng trên C++.
  - Nâng cao: Xây dựng các ứng dụng trên .Net, đặc biệt là C#.

Nội dung

- Lý thuyết
  - Giới thiệu chung
  - Các mở rộng của C++
  - Lớp và đối tượng
  - Kế thừa
  - Mô hình
- Ứng dụng
  - Xây dựng một số ứng dụng trên C#.(Sinh viên thực hiện)

Tiêu chí đánh giá

- Dự lớp
- Kiểm tra điều kiện: 1 bài trên máy, 1 bài trên giấy
- Thi kết thúc học phần: Thi trên máy.

Tài liệu tham khảo

- [1]. Nguyễn Thanh Thuỷ, Tạ Tuấn Anh "***Lập trình HĐT với C++***", NXB KH,2002.

- [2]. Nguyễn Thanh Thuỷ, Tạ Tuấn Anh "***Bài tập lập trình HĐT với C++***", NXB KH,2002.

- [3]. Nguyễn Hùng,"***C++ kỹ thuật và ứng dụng***", Biên dịch, Scitec.

Tài liệu tham khảo (tiếp)

- [4]. Đoàn Văn Ban, "***Phân tích, thiết kế và lập trình Hướng đối tượng***", NXB Thống kê, 1997.

- [5]. Ivar Jacobson, "***Object – Oriented Software Engineering***", ACM Press, 1998.

- [6]. Sharam Hekmat, "***C++ Programming***", www.pragsoft.com.

Tài liệu tham khảo(tiếp)

- [7]. Eric Gunnerson, ”***A Programmer's Introduction to C#***”, Apress, 2000.

- [8]. Jesse Liberty, “***Programming C#***”, Publisher: O'Reilly, First Edition July 2001.

- [9]. Erik Brown, “***Windows Forms Programming with C#***”, Manning Publications Co, 2002.