



TRƯỜNG ĐẠI HỌC THỦY LỢI
Khoa CNTT – Bộ môn CNPM

NGUYÊN LÝ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG
CSE 224

Giảng viên: Cù Việt Dũng
Email: dungcv@tlu.edu.vn
ĐT: 0964.644.986



- Tên học phần: Nguyên lý lập trình hướng đối tượng
- Mã học phần: CSE224
- Số tín chỉ: 03
- Phân bổ thời gian:
 - Giờ giảng lý thuyết: **30** tiết
 - Giờ thực hành: **15** tiết
 - Giờ tự học của sinh viên: **90** tiết
- Giảng viên: Cù Việt Dũng
- Email: dungcv@tlu.edu.vn
- Phone: 0964 644 986.

- ❑ Cung cấp cho sinh viên một khối lượng kiến thức về nguyên lý cũng như kỹ năng lập trình hướng đối tượng và sử dụng ngôn ngữ lập trình C++ để minh họa:
 - Nguyên lý lập trình hướng đối tượng: lớp, thuộc tính, phương thức, constructor, destructor, kế thừa, overload, override, hàm ảo, đa hình.
 - Không gian tên, template, xử lý ngoại lệ

Các qui định



- ❑ Không làm việc riêng trong lớp học
- ❑ Tham gia đầy đủ các buổi học, trao đổi, đóng góp ý kiến tích cực trong buổi học
- ❑ Thảo luận nhóm một cách sôi nổi, hiệu quả.
- ❑ Hoàn thành các nội dung và bài kiểm tra theo đúng thời gian qui định
- ❑ **Chú ý: Không được vắng mặt hôm kiểm tra**



Hình thức đánh giá



- Điểm thảo luận
 - Điểm giữa kì
 - Thi cuối kì: Thực hành

50%

50%



- ❑ **Mở đầu:** Giới thiệu.
- ❑ **Chương 1:** Nhắc lại về C++
- ❑ **Chương 2:** Lớp và đối tượng
- ❑ **Chương 3:** Nạp chồng toán tử
- ❑ **Chương 4:** Nguyên lý kế thừa
- ❑ **Chương 5:** Khuôn mẫu (Template) và thư viện chuẩn (STL)
- ❑ **Chương 6:** Hàm ảo và đa hình



Thảo luận



TRƯỜNG ĐẠI HỌC THỦY LỢI
Khoa CNTT – Bộ môn CNPM

NGUYÊN LÝ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

CSE 224

Chương 1: Nhắc lại về C++

Giảng viên: Cù Việt Dũng

Email: dungcv@tlu.edu.vn

ĐT: 0964.644.986



Nội dung

- 1.1 Tổng quan về C++
- 1.2 Cấu trúc một chương trình C++
- 1.3 Hàm và nạp chồng hàm
- 1.4. Con trỏ
- 1.5. Cấu trúc

1.1 Tổng quan về C++

- Tác giả: Bjarne Stroustrup (Mỹ)
- Ý tưởng bắt đầu từ năm 1979
- Được giới thiệu năm 1985
- Phiên bản C++ 2.0 năm 1989
- Phiên bản mới nhất: C++17
- Môn học này chỉ học khoảng 10% kiến thức về C++ và các thư viện của nó
- Cần 3-5 năm để trở thành lập trình viên C++ ở mức độ chuyên nghiệp

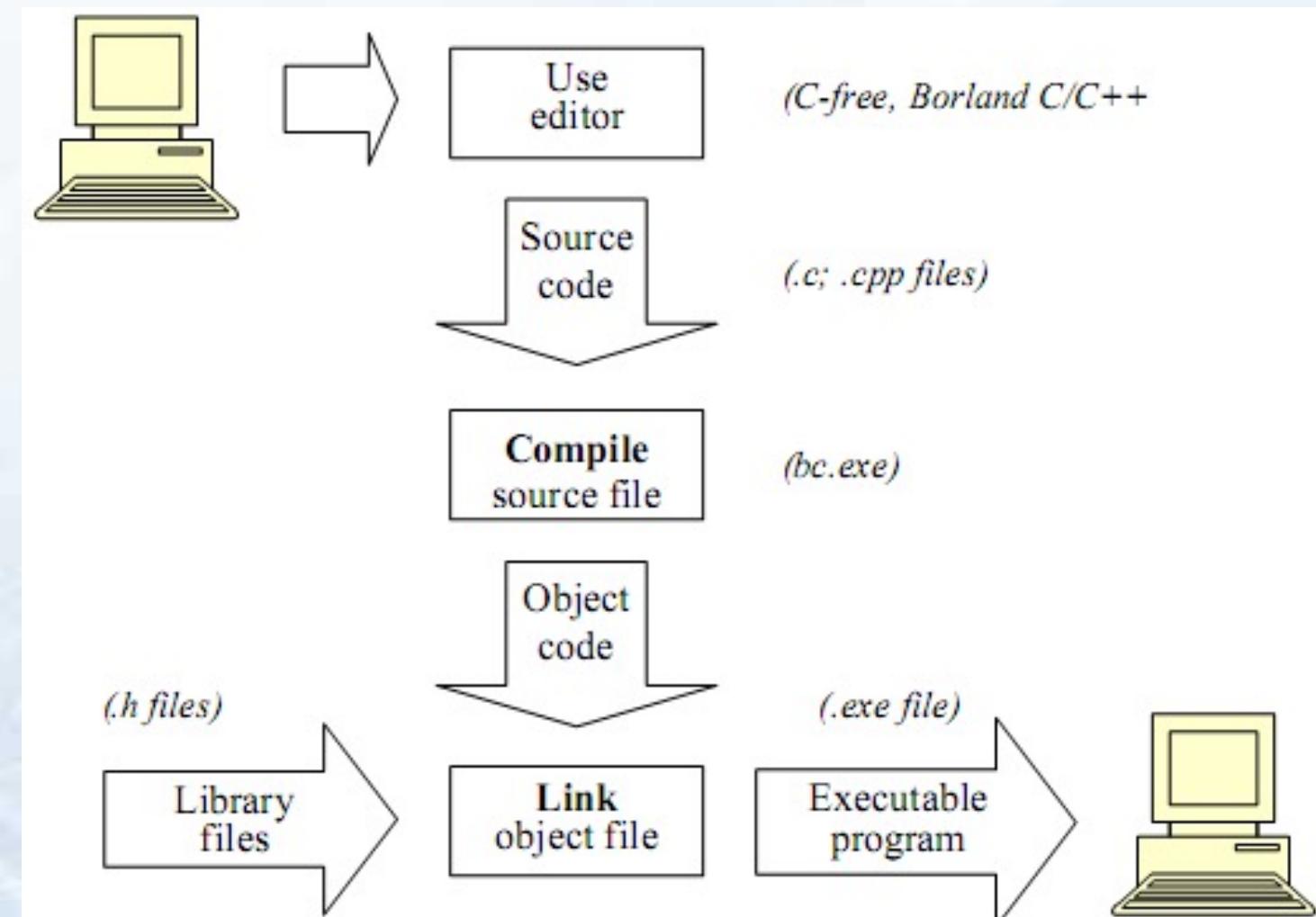




Các môi trường hỗ trợ lập trình (IDE)

- Borland C++
- Microsoft Visual Basic
- Microsoft Visual C++
- JBuilder
- Eclipse SDK
- Visual .Net
- ...

Các bước thực thi chương trình C++





1.2 Cấu trúc một chương trình C++





1.2 Cấu trúc một chương trình C++

```
/*Chuong trinh C++ */  
#include <iostream>  
  
using namespace std;  
  
int main()  
{  
    //Lenh cout<< de xuat ra man hinh  
    cout<<"Xin chao cac ban\n";  
    return 0;  
}
```

Chú thích
(Comment)

Chỉ thị tiền xử lý
(Preprocessor directive)

Chú thích
(Comment)

Lệnh
(Statement)

Chỉ thị tiền xử lý (*Preprocessor directive*)

- Các chỉ thị tiền xử lý là những dòng được đưa vào trong mã của chương trình phía sau dấu **#**
- Những dòng này không phải là lệnh của chương trình nhưng chỉ thi cho tiền xử lý
- Tiền xử lý kiểm tra mã lệnh trước khi biên dịch thực sự và **thực hiện tất cả các chỉ thị trước** khi thực thi mã lệnh của các câu lệnh thông thường



Chỉ thị tiền xử lý (*Preprocessor directive*)

Đặc điểm:

1. Mô tả trên một dòng, không có dấu ;
2. Trường hợp cần mô tả trên nhiều dòng dùng dấu \ ở cuối mỗi dòng



Khai báo thư viện `#include`

`#include <header>`
hoặc
`#include "file"`

Mục đích: Chỉ thị này sẽ thay thế toàn bộ nội dung của *header* hay một tập tin “*file*” *tự định nghĩa* thêm.

Các tập tin này thường chứa định nghĩa các hàm được sử dụng trong chương trình



Hàm main()

- Hàm main() là bắt buộc và được thực hiện đầu tiên khi thực thi chương trình C++
- Tập các lệnh trong hàm main() phải được đặt trong cặp dấu { }
- Chương trình sẽ thực hiện những lệnh theo thứ tự trong hàm main()



Lệnh/ khối lệnh

❑ Lệnh

Lệnh thực hiện một chức năng nào đó (khai báo, gán, xuất, nhập, ...) và được kết thúc bằng dấu chấm phẩy (;)

❑ Khối lệnh

Khối lệnh gồm nhiều lệnh và được đặt trong cặp dấu ngoặc { }

Lệnh/ khối lệnh

- Lệnh `cout<<“Xin chao cac ban\n”`; dùng để xuất ra màn hình dòng chữ “**Xin chao cac ban**”
- Lệnh `return 0`; dùng để kết thúc hàm main() – Kết thúc chương trình và trả về giá trị mã là 0

!!! Mỗi lệnh đều được kết thúc bằng dấu ;



Chú thích (comment)

```
/*Chuong trinh C++ */  
//Lenh cout<< de xuat ra man hinh
```

- Được lập trình viên ghi chú hay diễn giải trong chương trình
- Đây **không phải là lệnh**
- Chú thích một dòng: dùng // trước chú thích
- Chú thích cho nhiều dòng: dùng cặp dấu /* và */ để bao nội dung chú thích



1.3 Hàm và nạp chồng hàm

Hàm (chương trình con - subroutine) là một khối lệnh, **thực hiện trọng vẹn một công việc nhất định** (module), được đặt tên và được gọi thực thi nhiều lần tại nhiều vị trí

Khi nào sử dụng hàm?

1. Khi có một công việc giống nhau cần thực hiện ở nhiều vị trí
2. Khi cần chia nhỏ chương trình để dễ quản lý

Hàm

- ❑ Hàm có thể được gọi từ chương trình chính (hàm main) hoặc từ 1 hàm khác
- ❑ Hàm có giá trị trả về hoặc không
- ❑ Nếu hàm không có giá trị trả về gọi là thủ tục (*procedure*)



Hàm

- *Hàm thư viện*: là những hàm đã được xây dựng sẵn. Muốn sử dụng các hàm thư viện phải khai báo thư viện chứa nó trong phần khai báo #include
- *Hàm do người dùng định nghĩa*

Cấu trúc chung của hàm

<KDL trả về của hàm> TênHàm([ds tham số]);

Kiểu dữ liệu trả về của hàm (kết quả của hàm/ đầu ra), gồm:

- **void**: Không trả về giá trị
- **float / int / long / char */ kiểu cấu trúc / ...** : Trả về kết quả tính được với KDL tương ứng

Cấu trúc chung của hàm

- TênHàm: Đặt tên theo qui ước sao cho phản ánh đúng chức năng thực hiện của hàm
- Danh sách các tham số (nếu có): đầu vào của hàm

Trong một số trường hợp có thể là đầu vào và đầu ra của hàm nếu kết quả đầu ra có nhiều giá trị

Nạp chồng hàm

- Tên hàm giống nhau, danh sách các tham số khác nhau
- Hai định nghĩa hàm riêng biệt
- Tín hiệu hàm
 - Tên hàm & danh sách tham số
 - Phải là “duy nhất” cho mỗi định nghĩa hàm
- Cho phép thực hiện cùng một công việc trên các dữ liệu khác nhau



Nạp chồng hàm

```
1 #include <iostream>
2 using namespace std;
3
4 double average(double n1, double n2)
5 { return ((n1 + n2) / 2.0);}
6
7 double average(double n1, double n2, double n3)
8 { return ((n1 + n2 + n3) / 3.0);}
9
10 int main()
11 {
12     double x, y, z;
13     cout<<"Nhập giá trị x, y, z:";
14     cin>>x>>y>>z;
15     cout<<"Giá trị trung bình của "<<x<<" và "<<y<<" là: " <<average(x, y)<<endl;
16     cout<<"Giá trị trung bình của "<<x<<", "<<y<<" và "<<z<<" là: " <<average(x, y, z);
17     return 0;
}
```

```
Nhap gia tri x, y, z:3 4 5
Gia tri trung binh cua 3 va 4 la: 3.5
Gia tri trung binh cua 3, 4 va 5 la: 4
```

Nạp chồng hàm



```
// same name different arguments
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

```
// Error code
int test(int a) { }
double test(int b){ }
```



Bài tập áp dụng

Viết các hàm sau:

1. Đếm số ước số của số nguyên dương n.
2. Kiểm tra số nguyên n có phải là số nguyên tố không?
3. Đếm số chữ số là số nguyên tố
4. Hàm **main()** nhập vào một số nguyên n gồm k chữ số, gọi các hàm cần thiết để đếm xem n có bao nhiêu chữ số là số nguyên tố

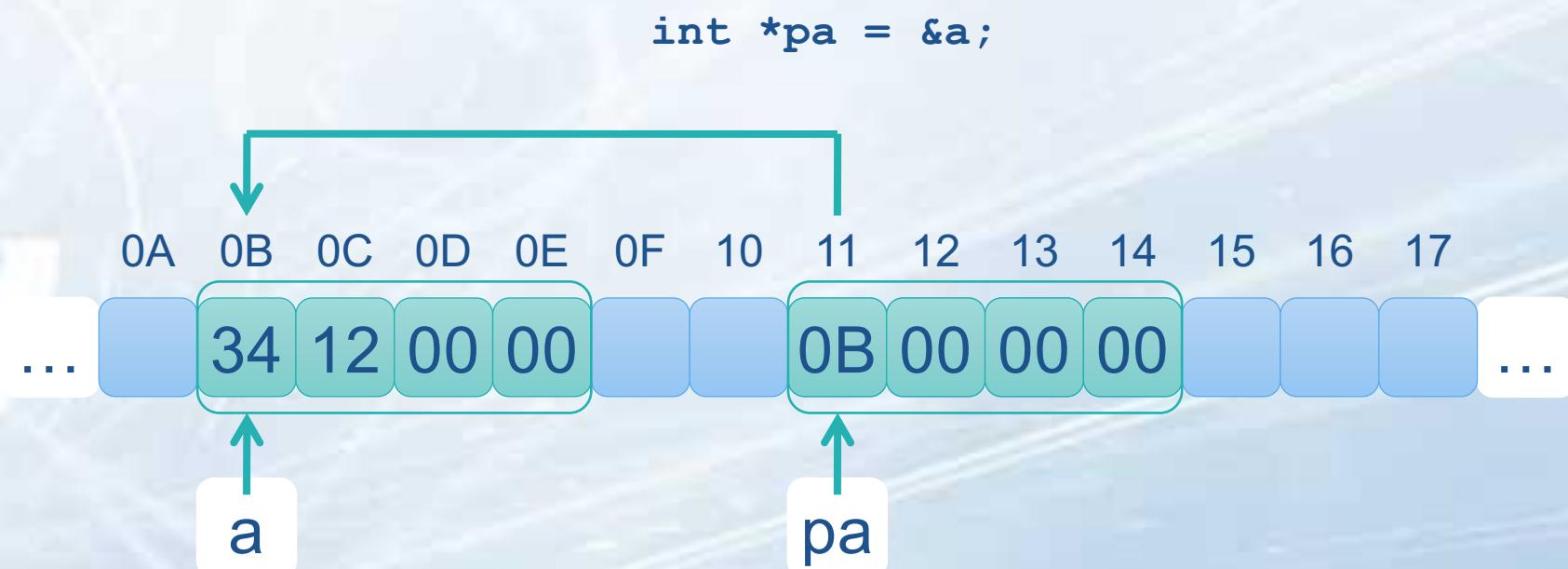


1.4. Con trỏ

Khái niệm con trỏ

□ Khái niệm

- Địa chỉ của biến là một con số.
- Ta có thể tạo biến khác để lưu địa chỉ của biến này → Con trỏ.



Khai báo con trỏ

□ Khai báo

- Giống như mọi biến khác, biến con trỏ muốn sử dụng cũng cần phải được khai báo

█ <kiểu dữ liệu> *<tên biến con trỏ>;

□ Ví dụ

█ `char *ch1, *ch2;
int *p1, p2;`

- ch1 và ch2 là biến con trỏ, trỏ tới vùng nhớ kiểu char (1 byte).
- p1 là biến con trỏ, trỏ tới vùng nhớ kiểu int (4 bytes) còn p2 là biến kiểu int bình thường.



Khai báo con trỏ

□ Ví dụ

<kiểu dữ liệu> *<tên kiểu con trỏ>;

```
double *p;  
int *p1;  
char *p2;
```

Con trỏ NULL

□ Khái niệm

- Con trỏ **NULL** là con trỏ không trỏ và đâu cả.
- Khác với con trỏ chưa được khởi tạo.

```
int n;  
int *p1 = &n;  
int *p2; // unreferenced local variable  
int *p3 = NULL;
```



Khởi tạo kiểu con trỏ

❑ Khởi tạo

- Khi mới khai báo, biến con trỏ được **đặt ở địa chỉ nào đó** (không biết trước).
→ Đặt địa chỉ của biến vào con trỏ (toán tử **&**)

❑ Ví dụ

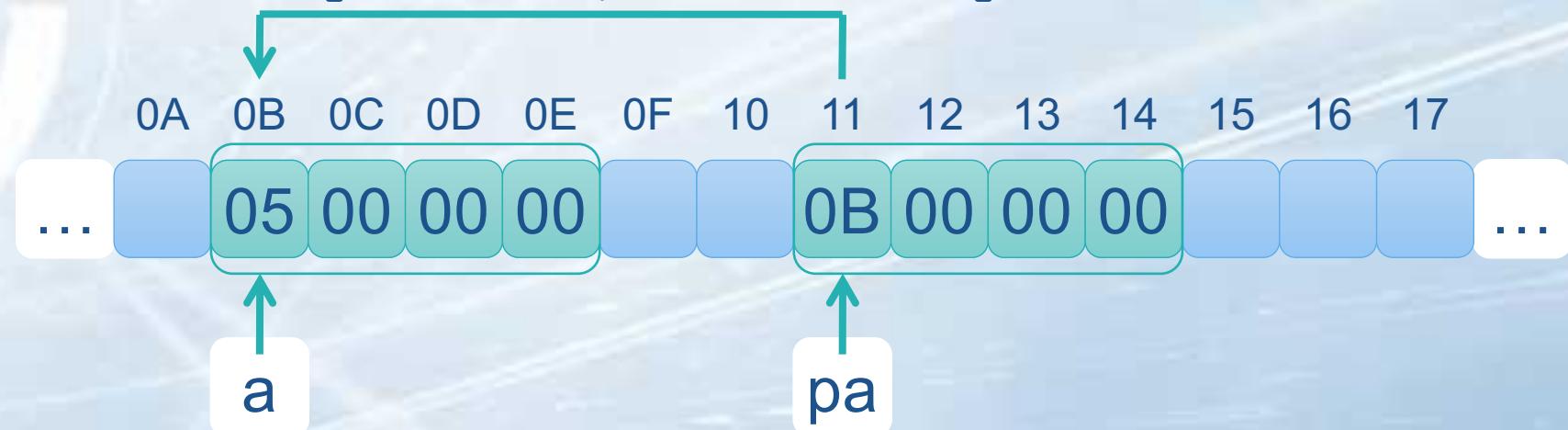
<biến con trỏ> = &<biến>;

```
int a, b;  
int *pa = &a, *pb;  
pb = &b;
```

Sử dụng con trỏ

- Truy xuất đến ô nhớ mà con trỏ trỏ đến
 - Con trỏ chứa **một số nguyên chỉ địa chỉ**.
 - Vùng nhớ mà nó trỏ đến, sử dụng toán tử *****.
- Ví dụ

```
int a = 5, *pa = &a;  
cout<< pa; // Giá trị biến pa  
cout<< *pa; // Giá trị vùng nhớ pa trỏ đến  
cout<< &pa; // Địa chỉ biến pa
```



Kích thước của con trỏ

□ Kích thước của con trỏ

```
char *p1;  
int *p2;  
float *p3;  
double *p4;  
...
```

□ Con trỏ **chỉ lưu địa chỉ** nên kích thước của mọi con trỏ là như nhau

1.5. Cấu trúc



Kiểu struct

- **Ý nghĩa:** là cấu trúc cho phép bên trong chứa các trường dữ liệu mà có kiểu dữ liệu có thể khác nhau
- Các thao tác cơ bản:
 - Định nghĩa
 - Khai báo biến
 - Truy nhập vào các trường
 - Gán giá trị



Kiểu struct – Định nghĩa

```
struct tên_kiểu {  
    Khai báo các trường;  
};
```

VD:

```
struct date {  
    int ngay;  
    int thang;  
    int nam;  
};
```

```
struct nguoi {  
    char ten[30];  
    int tuoi;  
    char gioitinh; // ‘M’ cho nam, ‘F’ cho nữ  
    date ngaysinh;  
};
```

Truy nhập vào các trường

□ Có 2 cách truy nhập:

- Cách 1: dùng biến thông thường, sử dụng cú pháp “**tên_bien.tên_trường**”
- Cách 2: dùng biến con trỏ, sử dụng cú pháp “**tên_bien->tên_trường**”

```
struct lophoc{  
    char sohieu[30] ;  
    char chuyennganh;  
    int soluong;  
};  
lophoc lh;  
lophoc * p = & lh;  
  
cin>>lh.sohieu;  
cin>>p->chuyennganh;  
cin>>p->soluong;
```

Kiểu struct - Gán giá trị

- Hai biến cùng một kiểu struct có thể được gán cho nhau. Việc gán này sẽ cho thay cho việc gán lần lượt tất cả các trường của hai biến này cho nhau.

VD:

```
lophoc lh1, lh2;  
lh1=lh2;
```

```
lh1.sohieu = lh2.sohieu;  
lh1.chuyennganh = lh2.chuyennganh;  
lh1.soluong = lh2.soluong;
```



Bài tập áp dụng

- Viết chương trình quản lý một danh sách nhân viên với các chức năng:
 - Nhập vào một danh sách N nhân viên (N là hằng số cho trước)
 - In ra nội dung danh sách đó



NGUYÊN LÝ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

CSE 224

Chương 2: Lớp và đối tượng

Giảng viên: Cù Việt Dũng

Email: dungcv@tlu.edu.vn

ĐT: 0964.644.986

NỘI DUNG



- 2.1 Định nghĩa lớp
- 2.2 Hàm tạo và hàm hủy
- 2.3 Thành phần tĩnh
- 2.4 Hàm bạn và lớp bạn

NỘI DUNG:



- 2.1 Định nghĩa lớp
- 2.2 Hàm tạo và hàm hủy
- 2.3 Thành phần tĩnh
- 2.4 Hàm bạn và lớp bạn



Định nghĩa lớp

- Định nghĩa lớp là tạo ra một kiểu dữ liệu trừu tượng mới để mô tả các đặc trưng của đối tượng trong thực tế.
- **Định nghĩa lớp gồm hai bước:**
 - *Khai báo lớp*: khai báo các dữ liệu và hàm thành phần (phương thức) tạo nên lớp
 - *Định nghĩa lớp*: định nghĩa cụ thể các hàm thành phần của lớp



Khai báo lớp

class tên-lớp-được-định-nghĩa

{

private: *//khai báo vùng che giấu riêng của lớp*

khai báo các dữ liệu

khai báo các phương thức

protected: *//khai báo vùng bảo vệ của lớp*

khai báo các dữ liệu

khai báo các phương thức

public: *//khai báo vùng dùng chung của lớp*

khai báo các dữ liệu

khai báo các phương thức

};



Khai báo lớp

❑ **Dữ liệu:** xác định các thành phần dữ liệu để mô tả lớp \Leftrightarrow gọi là trường, thuộc tính để xác định đối tượng

- dữ liệu được mô tả thông qua tên và kiểu xác định \Leftrightarrow ***khai báo biến dữ liệu***
- kiểu: chấp nhận được khai báo các kiểu gồm các kiểu cơ bản(int, float,...) hay con trỏ đến đối tượng lớp
- ví dụ:

```
class date{  
    private:  
        int ngay;  
        int thang;  
        int nam;  
    ....  
};
```

Khai báo lớp



❑ **Phương thức:** xác định các phương thức hay tác vụ thực hiện xử lý trên dữ liệu của lớp, được khai báo như khai báo hàm

<kiểu-trả-lại> Tên-phương-thức(ds đối số);

❑ Ví dụ:

```
class date{
```

```
....
```

```
public:
```

```
    int get_ngay( );
    int get_thang( );
    int get_nam( );
    int sosanh( date &d );
```

```
};
```

Khai báo lớp



- ❑ Các dữ liệu và hàm thành phần của lớp được mô tả với 3 mức khác nhau:
 - **private**: sở hữu riêng, dùng để khai báo các thành viên là riêng của chỉ lớp đó và không thể truy cập từ các lớp khác \Leftrightarrow được dùng với mục đích che giấu thông tin của lớp
 - **protected**: chế độ bảo vệ, khai báo các thành phần của riêng lớp đó và các lớp kế thừa nó có thể truy cập được còn các lớp khác không truy cập được
 - **public**: dùng chung, dùng khai báo các thành phần có thể truy cập từ mọi nơi \Leftrightarrow truy cập được từ bên ngoài lớp
- ❑ Chú ý: Nếu không có từ khóa **private** thì mặc định hiểu là thuộc vùng **private**

Ví dụ:



```
class date
{
private:
    int ngay;
    int thang;
    int nam;
public:
    int get_ngay();
    int get_thang();
    int get_nam();
    void in( );
    void nhanh( );
};
```

2. Định nghĩa hàm thành phần

- ❑ Định nghĩa hàm thành phần là thao tác định nghĩa cụ thể các hàm – thao tác của lớp được thực hiện như thế nào?
- ❑ Các hàm thành phần của lớp được định nghĩa theo hai cách:
 - *Định nghĩa từ bên ngoài khai báo lớp*
 - *Định nghĩa bên trong khai báo lớp*



Định nghĩa bên ngoài khai báo lớp

- ❑ Là thực hiện định nghĩa bên ngoài vùng khai báo **class** \Leftrightarrow định nghĩa sau dấu “ ; ” của khai báo **class**
- ❑ Định nghĩa bên ngoài phải có thêm tiếp đầu ngữ chỉ ra **tên lớp** chứa hàm cần định nghĩa.
- ❑ Dạng tổng quát:

Kiểu-trả-lại **tên-lớp** :: tên-hàm (ds đối số)

{

Định nghĩa nội dung hàm

}

Ví dụ



```
#include "iostream.h"
class date
{
    int ngay,thang,nam;
public:
    void nhap();
    void in();
};
```

```
void date :: nhap( )
{
    cout<<"Nhập ngày tháng năm:";
    cin>>ngay>>thang>>nam;
}
```

```
void date :: in( )
{
    cout<<"Ngày:"<<ngay<<" / "<<thang<<" / "<<nam;
}
```



Định nghĩa trong lớp

- ❑ Định nghĩa ngay tại vị trí các hàm thành phần trong phần mô tả và khai báo lớp
- ❑ Thường dùng với các hàm đơn giản và ít dòng lệnh

Ví dụ



```
#include <iostream.h>
class date
{
    int ngay,thang,nam;
public:
    void nhap()
    {
        cout<<"Nhập ngày tháng năm:";
        cin>>ngay>>thang>>nam;
    }
    void in()
    {
        cout<<"Ngày:"<<ngay<<" / "<<thang<<" / "<<nam;
    }
};
```

Tạo lập đối tượng

- ❑ Để sử dụng lớp đối tượng phải tạo ra các đối tượng \Leftrightarrow khai báo biến kiểu class đã được định nghĩa
- ❑ Ví dụ:
`date ngaysinh;`
- ❑ Để truy cập đến từng thành phần đối tượng thông qua tên biến kiểu class đã được khai báo và toán tử dấu chấm (.) theo dạng sau:
`Tên-biến-đối-tượng.dữ-liệu-thành-phần;`
`Tên-biến-đối-tượng.hàm-thành-phần(ds đối số);`
- ❑ Ví dụ: `ngaysinh.ngay`
`ngaysinh.thang`
`ngaysinh.nhap();`
`ts = ngaysinh.get_thang();`

Truy cập các thành viên của đối tượng



- ❑ Đối với các thành viên dữ liệu của lớp đối tượng đang định nghĩa thì:
 - Gán dữ liệu cho đối tượng: thực hiện thay đổi từng thành phần dữ liệu của đối tượng
 - thay đổi các dữ liệu private của lớp: chỉ có các phương thức của lớp mới có quyền thay đổi giá trị các thành phần riêng của lớp
 - thay đổi các dữ liệu trong thành phần public: có thể được thực hiện bằng bất cứ thành phần nào của lớp

Con trỏ This



❑ Mỗi một lớp trong nó luôn có một con trỏ mặc định là con trỏ **this**, con trỏ để mô tả chính đối tượng lớp đang định nghĩa.

❑ **Các hàm thành phần của lớp luôn có tham số đầu tiên là con trỏ this.**

❑ Ví dụ:

class PS

{

public:

 int ts, ms;

 void nhap();

 // void nhap(PS *this);

 PS cong(PS b);

 //PS cong(PS *this , PS b);

};

Con trỏ this



- Truy cập đến thành phần con trỏ this

this->tên_thành_phần ⇔ tên_thành_phần

Ví dụ:

```
void PS::nhap()
{
    cout<<“Tu:”; cin>>ts;
    cout<<“Mau:”; cin>>ms;
}
```



```
void PS::nhap()
{
    cout<<“Tu:”;
    cin>>this->ts;
    cout<<“Mau:”;
    cin>>this->ms;
}
```

NỘI DUNG:



- 2.1 Định nghĩa lớp
- 2.2 Hàm tạo và hàm hủy**
- 2.3 Thành phần tĩnh
- 2.4 Hàm bạn và lớp bạn

Constructor và Destructor



- ❑ là phương thức được định nghĩa đặc biệt được sử dụng khi khởi tạo một đối tượng mới hay loại bỏ đối tượng.
- **Constructor** : cho phép tự động tạo ra một đối tượng mới khi khai báo
- **Destructor** : tự động phá bỏ đối tượng khi không cần dùng đến

Constructor



- ❑ Là hàm thành phần đặc biệt của lớp, làm nhiệm vụ tạo lập các đối tượng theo yêu cầu.
- ❑ Tên Constructor được **đặt trùng với tên lớp** đang định nghĩa

❑ Ví dụ:

```
class sophuc
{
    private:
        double x;           //phan thuc
        double y;           //phan ao
    public:
        //dinh nghia ham toan tu cho phep tao doi tuong so phuc
        sophuc( void );
        sophuc( double a, double b );
        void nhap();
        void in();
};
```

Tính chất của Constructor



- ❑ Phải khai báo trong vùng public
- ❑ Tên của constructor phải trùng với tên của lớp
- ❑ Tự động được thực hiện khi khai báo đối tượng
- ❑ Một lớp có thể có nhiều hàm constructor
- ❑ Constructor không có kiểu trả về
- ❑ Không thể kế thừa, nhưng lớp dẫn xuất vẫn có thể gọi constructor của lớp cơ sở

Phân loại Constructor

❑ Constructor *mặc định* \Leftrightarrow Constructor không có tham số \Rightarrow tạo đối tượng chỉ cần đặt tên

Ví dụ: **sophuc** *a,b,c*;

❑ Constructor *có tham số*: là constructor có các tham số để cho phép khởi tạo một bộ giá trị nào đó cho các dữ liệu thành phần của đối tượng, kiểu tham số cho constructor có thể là kiểu bất kỳ ngoại trừ **kiểu lớp đang định nghĩa**.

Ví dụ: **sophuc** *tg(10,-2);*

sophuc *tong(2,-3), hieu(0,0);*

- ❑ Là phương thức(method) thực hiện giải phóng bộ nhớ đã cấp cho đối tượng khi không cần sử dụng đối tượng.
- ❑ Mỗi lớp chỉ có duy nhất một Destructor
- ❑ Destructor là method rỗng không có tham số và là thành phần của lớp có cùng tên lớp và thêm tiếp đầu ngữ “~”
- ❑ Ví dụ: **~date();**

Ví dụ



```
#include "iostream.h"
class date
{
    int ngay,thang,nam;
public:
    date( ){ ngay=0; thang=0; nam=0; }
    date( int d, int m, int y ){ ngay = m; thang = n; nam = y }
    ~date( ) { }
    void nhap()
    {
        cout<<"Nhập ngày tháng năm:";
        cin>>ngay>>thang>>nam;
    }
    void in()
    {
        cout<<"Ngày:"<<ngay<<" / "<<thang<<" / "<<nam;
    }
};
```

Ví dụ



```
void main()
{
    date ngaysinh(01,04,2020), ngay_vcq;
    cout<<“ngay sinh cua ban la \n”;
    ngaysinh.in( );
    cout<<“Nhập ngày vào cơ quan \n”;
    ngay_vcq.nhap( );
    cout<<“Bạn đã nhập ngày vào cơ quan là \n”;
    ngay_vcq.in( );
    getch( );
}
```

Constructor sao chép



- Là Constructor cho phép tạo ra một đối tượng mới từ một đối tượng đã có nhưng hoàn toàn độc lập với đối tượng đã có đó.

VD: PS **d(2,3);**

PS **u(d);**

- Khi lớp không có thuộc tính kiểu con trỏ hoặc tham chiếu thì ta chỉ dùng hàm tạo sao chép mặc định.
- Khi lớp có các thuộc tính con trỏ hoặc tham chiếu thì phải định nghĩa hàm tạo sao chép mới.

Tên_lớp (const Tên_lớp &đối_tượng)

{

// Các lệnh dùng các thuộc tính của đối tượng
// khởi gán cho các thuộc tính của đối tượng mới

.....

}

* ví dụ:



```
class PS
{
public:
    int ts, ms;
    PS( PS &x); //Hàm tạo sao chép
    void nhap();
    void in();
};

//Định nghĩa hàm tạo sao chép
PS :: PS(const PS &x)
{
    ts = x.ts;
    ms = x.ms;
}
```

NỘI DUNG:



- 2.1 Định nghĩa lớp
- 2.2 Hàm tạo và hàm hủy
- 2.3 Thành phần tĩnh
- 2.4 Hàm bạn và lớp bạn

Hàm thành phần tĩnh



- ❑ Được khai báo với từ khóa static ở đâu.
- ❑ Đặc tính của hàm thành phần tĩnh:
 - *Hàm khai báo static chỉ có thể truy nhập tới những thành phần tĩnh trong lớp*
 - *Hàm thành phần tĩnh có thể được gọi với tên của lớp thay cho tên của đối tượng*

- ❑ Được dùng khi không muốn thay đổi dữ liệu của đối tượng lớp ở trong một số phương thức (hàm) nào đó.
- ❑ **Sử dụng:** thêm từ khóa **const** vào trước khai báo của tham số hàm
- ❑ **ví dụ:**
friend sophuc cong(**const** sophuc &c1, **const** sophuc &c2);
friend sophuc tru(**const** sophuc &c1, **const** sophuc &c2);

NỘI DUNG:



- 2.1 Định nghĩa lớp
- 2.2 Hàm tạo và hàm hủy
- 2.3 Thành phần tĩnh
- 2.4 **Hàm bạn và lớp bạn**

Phân loại hàm thành phần



- ❑ Hàm thành phần của lớp gồm các loại sau:
 - Hàm thành phần riêng của lớp \Leftrightarrow được khai báo trong vùng **private**
 - Hàm thành phần thân thiện \Leftrightarrow được khai báo với từ khóa **friend**
 - Hàm thành phần tĩnh \Leftrightarrow được khai báo với từ khóa **static**

Hàm thành phần private



- ❑ Hàm che giấu không cho phép các đối tượng khác truy cập đến
- ❑ Hàm được khai báo trong vùng private
- ❑ Hàm private chỉ gọi được thông qua các hàm khác trong cùng lớp
- ❑ Đối tượng trong cùng lớp cũng không truy cập được đến hàm private khi khai báo ngoài vùng định nghĩa lớp

Hàm thành phần friend



- ❑ Hàm friend là hàm được định nghĩa cho phép nhiều lớp cùng sử dụng chung
- ❑ Có quyền truy cập đến các thành viên **private** và **protected**
- ❑ Khai báo: thêm từ khóa **friend** vào trước kiểu của hàm thành phần
- ❑ Cú pháp khai báo:
friend type tên-hàm(parameter);

Hàm thành phần friend



- ❑ Hàm friend có thể định nghĩa ở mọi nơi và không cần dùng từ khóa friend hay toán tử $:: \Leftrightarrow$ Hàm friends là một hàm được định nghĩa thông thường



Đặc điểm hàm friend

- ❑ Không nằm trong miền xác định của lớp nơi được khai báo
- ❑ Khi truy cập đến hàm không cần gắn với đối tượng của lớp ⇔ truy nhập đến các hàm friend thực hiện như các hàm khai báo thông thường
- ❑ Thông thường đối số của các hàm friend là các đối tượng đang định nghĩa
- ❑ Hàm friend không sử dụng con trỏ this

Ví dụ

- ❑ Xây dựng lớp phân số với các thao tác cần định nghĩa như sau:
 - Cộng, trừ hai phân số
 - Nhân, chia hai phân số (*hàm friend*)

Ví dụ



```
#include...
//Mo ta lop

class ps
{
    int ts, ms;      //tp du lieu
public:
    //dinh nghia ham nhap xuat
    void nhap( );
    void in( );
    //dinh nghia ham cong, tru, nhan, chia
    ps cong( ps b);
    ps tru( ps b);

    friend ps nhan(ps a, ps b);
    friend ps chia(ps a, ps b);
};
```

Ví dụ



```
//Định nghĩa hàm thành phần  
void ps::in()  
{  
    if(ms==1) cout<<ts;  
    else  
    {  
        if(ts==0) cout<<"0";  
        else cout<<ts<<"/"<<ms;  
    }  
}
```

Ví dụ



```
void ps::nhap()
{
    cout<<"\n";
    cout<<"Nhập tọa độ: "; cin>>ts;
    cout<<"Nhập màu sắc: "; cin>>ms;
    if (ms<0)
    {
        ts=-ts;
        ms=-ms;
    }
}
```

Ví dụ



```
ps ps::cong( ps b)
{
    ps kq;
    kq.ts = ts*b.ms+b.ts*ms;
    kq.ms = ms*b.ms;
    return kq;
}
ps ps::tru( ps b)
{
    ps kq;
    kq.ts = ts*b.ms - b.ts*ms;
    kq.ms = ms*b.ms;
    return kq;
}
```

Ví dụ



```
ps nhan( ps a, ps b)
{
    ps kq;
    kq.ts = a.ts*b.ts;
    kq.ms = a.ms*b.ms;
    return kq;
}
ps chia( ps a, ps b)
{
    ps kq;
    kq.ts = a.ts*b.ms;
    kq.ms = a.ms*b.ts;
    return kq;
}
```

```
void main()
{
    ps x, y, kq;
    cout<<"Nhap phan so thu nhat: \n";
    x.nhap();
    cout<<"\n\n Nhap phan so thu hai:\n";
    y.nhap();
    kq = x.cong(y);
    cout<<"\n Tong:";
    kq.in();
    kq = nhan(x, y);
    cout<<"\n Tich:";
    kq.in();
    getch();
}
```

Bài tập



```
class Matrix
{
private:
    int m; // dòng
    int n; // cột
    double elements[100][100];
public:
    Matrix();
    ~Matrix();
    Matrix(const Matrix & a);
    void nhap();
    void xuat();
    int Cong(const Matrix & a); // return 1 nếu cộng dc
    void Nhan(const double & k); // Nhân với 1 số K
    int Nhan(const Matrix & a); // return 1 nếu nhân đc
    friend Vector multiply(const Matrix &a, const Vector &b);
};
```



NGUYÊN LÝ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

CSE 224

Chương 3: Nạp chồng toán tử

Giảng viên: Cù Việt Dũng

Email: dungcv@tlu.edu.vn

ĐT: 0964.644.986

NỘI DUNG



- 3.1 Nạp chồng toán tử
- 3.2 Nạp chồng toán tử bằng hàm thành viên
- 3.3 Nạp chồng toán tử bằng hàm bạn

NỘI DUNG



- 3.1 Nạp chồng toán tử
- 3.2 Nạp chồng toán tử bằng hàm thành viên
- 3.3 Nạp chồng toán tử bằng hàm bạn



Định nghĩa

- Nạp chồng toán tử là khả năng định nghĩa các phương thức của lớp dưới dạng các toán tử (**phép toán**)
- Các toán tử **cùng tên** thực hiện được nhiều chức năng khác nhau được gọi là **nạp chồng (overload) toán tử**
- Ví dụ:

PS operator - ();

friend PS operator - (PS x , PS y);

* Khai báo tổng quát:

<return - type> operator # (d/s đổi số);

trong đó:

- ✓ return-type: kiểu trả lại kết quả của hàm toán tử
- ✓ #: tên toán tử cần định nghĩa (+,-,*,/,...)

• Nạp chồng toán tử được định nghĩa trong vị trí public của phần khai báo lớp.

Các toán tử được nạp chồng

+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&		++
--	->*	,	->	[]	0	new	delete



Các toán tử không được nạp chồng

Bài tập



Viết chương trình xây dựng lớp Phân số. Yêu cầu định nghĩa:

- Hàm tạo không tham số, hàm tạo có tham số
- Nhập và xuất phân số
- Tính tổng hai phân số
- Tính hiệu hai phân số (*friend*)
- Áp dụng nhập vào hai phân số và in ra kết quả



Định nghĩa toán tử >> và <<

- ❑ Cho phép nạp chồng hai toán tử nhập (>>) và xuất(<<) cho các đối tượng của lớp bất kỳ
- ❑ Định nghĩa hai toán tử dưới dạng hàm friend
- ❑ Khi đó có thể sử dụng dòng **cin>>** và **cout<<** để nhập và xuất dữ liệu cho đối tượng lớp



Định nghĩa toán tử <<

❑ Khai báo:

friend ostream & operator << (ostream &out, class-name c);

❑ Định nghĩa:

ostream & operator << (ostream &out, class-name c)

{

//định nghĩa thao tác in các giá trị của “c” ra màn hình. Lưu ý
dùng dòng out<< để xuất thay vì dùng cout<<

 return out;

}

Định nghĩa toán tử <<



❑ Trong đó:

- **ostream**: tham chiếu chỉ đến dòng xuất,
- **out**: tên dòng xuất tạm thời sử dụng thay cho cout
- **class-name**: là tên của lớp đối tượng cần định nghĩa toán tử xuất

Ví dụ



```
class phanso
{
public:
    int mau, tu;
    ....
friend ostream& operator<<(ostream &out, phanso a);
    ....
};

ostream& operator<<(ostream &out, phanso a)
{
    out<<"Phan so:"<<a.tu<<" / "<<a.mau;
    return out;
}
```

Định nghĩa toán tử >>



❑ Khai báo:

friend istream & operator >> (istream &in, class-name &c);

❑ Định nghĩa:

istream & operator >> (istream &in, class-name &c)

{

//định nghĩa thao tác nhập các giá trị cho đối tượng “c”. Lưu ý
dùng dòng in>> để nhập thay vì dùng cin>>

 return in;

}

Định nghĩa toán tử >>



- ❑ Trong đó:
 - **istream**: tham chiếu chỉ đến dòng nhập,
 - **in**: tên dòng nhập tạm thời sử dụng thay cho cin
 - **class-name**: là tên của lớp đối tượng cần định nghĩa toán tử nhập

Ví dụ



```
class phanso
{
public:
    int mau, tu;
    ....
friend istream& operator>>(istream &in, phanso &a);
    ....
};

istream& operator>>(istream &in, phanso &a)
{
    cout<<"Nhập tu:";           in>>a.tu;
    cout<<"Nhập mau:";          in>>a.mau;
    return in;
}
```

Bài tập áp dụng



Viết chương trình xây dựng lớp số phức. Yêu cầu:

- Định nghĩa toán tử nhập >> và xuất <<
- Định nghĩa toán tử +, - để cộng, trừ hai số phức
- Định nghĩa toán tử *, / để nhân, chia hai số phức(friend)
- Định nghĩa toán tử == để so sánh hai số phức(friend)
- Áp dụng nhập vào hai số phức và in ra màn hình các kết quả

Toán tử chuyển đổi kiểu



- Toán tử chuyển đổi kiểu được dùng để chuyển đổi một đối tượng của lớp thành đối tượng lớp khác hoặc thành đối tượng của một kiểu dữ liệu đã có sẵn
- Hàm chuyển đổi kiểu phải là hàm thành viên không tĩnh (*không static*) và không hàm bạn (friend)

Toán tử chuyển đổi kiểu



❑ cú pháp:

operator <kiểu-cân-chuyển>();

❑ ví dụ:

class date

{

public:

operator int(); //chuyển kiểu date về kiểu int

};

Ví dụ



Xây dựng lớp Date. Yêu cầu:

- Định nghĩa hàm toán tử nhập và xuất
- Định nghĩa toán tử chuyển đổi để chuyển một giá trị Date thành kiểu số nguyên
- Áp dụng nhập vào một ngày bất kỳ và in ra số nguyên tương ứng của ngày đó

Ví dụ



```
//Dinh nghia kieu du lieu moi co ten la String
typedef char *string;
//Dinh nghia mang chua so ngay
int ngay[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};
//Mo ta lop
class N_Date
{
    int d,m,y;//Ngay - Thang - Nam
public:
    //Toan tu chuyen doi kieu,
    operator int();
    //Dinh nghia phuong thuc nhap va xuat
    friend ostream& operator<<(ostream &out, N_Date x);
    friend istream& operator>>(istream &in, N_Date &x);
};
```

Ví dụ



```
istream& operator>>(istream &in, N_Date &x)
{
    cout<<"Ngay:";    in>>x.d;
    cout<<"Thang:";   in>>x.m;
    cout<<"Nam:";    in>>x.y;
    return in;
}
```

Ví dụ



```
ostream& operator<<(ostream &out, N_Date x)
{
    out<<x.d<<" / " <<x.m<<" / " <<x.y;
    return out;
}
```

Ví dụ



```
N_Date::operator int()
{
    int s=0,i;
    if( y== 1900)
    {
        s=s+d;
        for(i=0;i<m;i++)      s = s + ngay[i];
    }
    else
    {
        s=s+d;
        s=s+(y-1)*365;
        for(i=1900;i<y;i++)
            if((i%4==0) &&(i%100!=0)) s=s+1;
        for(i=0;i<m;i++)
        {
            s=s+ngay[i];
            if((i%4==0) &&(i%100!=0) && (i==2)) s=s+1;
        }
    }
    return s;
}
```

Ví dụ



```
void main()
{
    N_Date x;
    long n;
    cout<<"Nhập ngày x: \n";
    cin>>x;
    n = long(x);
    cout<<"\n Ngày x = "<<x;
    cout<<"\n Số nguyên tuong ứng: "<<n;
    getch();
}
```

Toán tử []



- ❑ Là toán tử cho phép truy cập đến từng thành phần của đối tượng (**chuỗi ký tự, mảng**)
- ❑ là **Toán tử hai ngôi**, có dạng: $a[b]$
 - a : đối tượng cần truy cập
 - b : chỉ số vị trí phần tử cần truy cập
- ❑ Toán tử này phải là thành viên của lớp

NỘI DUNG



- 3.1 Nạp chồng toán tử
- 3.2 Nạp chồng toán tử bằng hàm thành viên
- 3.3 Nạp chồng toán tử bằng hàm bạn

- **Hàm thành viên:** hàm này không có đối số cho toán tử một ngôi hay có một đối số cho toán tử hai ngôi \Leftrightarrow tương tự hàm thành phần thông thường
- **Hàm không thành viên - Hàm bạn:** hàm có một đối số cho toán tử một ngôi và hai đối số cho toán tử hai ngôi. Hàm được khai báo thêm từ khóa *friend* trước tên hàm \Leftrightarrow tương tự hàm bạn

Quy tắc sử dụng



Loại	Khai báo	Sử dụng
Thành phần	type operator #();	a.operator #(); hoặc #a;
	type operator #(type b);	a.operator #(b); hoặc a#b;
Thân thiện	friend type operator #(type a);	#a
	friend type operator #(type a, type b);	a#b

Trong đó: # là toán tử cần định nghĩa

Ví dụ



Loại	Khai báo	Sử dụng
Thành phần	PS operator -();	$kq = a.operator-();$ hoặc $kq = -a;$
	PS operator -(PS b);	$kq = a.operator -(b);$ hoặc $kq = a-b;$
friend	friend PS operator -(PS a);	$kq = -a$
	friend PS operator - (PS a, PS b);	$kq = a - b$

Trong đó: # là toán tử cần định nghĩa

Các nguyên tắc cơ bản



- ❑ Có thể định nghĩa nạp chồng trên các kiểu có sẵn hoặc trên các kiểu mới (*kiểu do người dùng định nghĩa*)
- ❑ Toán tử gán được sử dụng mặc định cho mọi lớp mà không cần định nghĩa
- ❑ Toán tử lấy địa chỉ (&) cũng được sử dụng mặc định cho các đối tượng
- ❑ Khi đa năng hóa (), [], -> hoặc = thì method nạp chồng phải là hàm thành viên



TRƯỜNG ĐẠI HỌC THỦY LỢI
Khoa CNTT – Bộ môn CNPM

NGUYÊN LÝ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

CSE 224

Chương 4: Nguyên lý kế thừa

Giảng viên: Cù Việt Dũng

Email: dungcv@tlu.edu.vn

ĐT: 0964.644.986

NỘI DUNG



- ❑ 4.1 Các kiểu kế thừa
- ❑ 4.2 Hàm tạo và hàm hủy trong kế thừa
- ❑ 4.3 Định nghĩa lại hàm thành viên

NỘI DUNG



- ❑ 4.1 Các kiểu kế thừa
- ❑ 4.2 Hàm tạo và hàm hủy trong kế thừa
- ❑ 4.3 Định nghĩa lại hàm thành viên



Giới thiệu

- Kế thừa là một cơ chế sử dụng lại rất mạnh, **cho phép định nghĩa lớp mới từ các lớp đã có bằng cách sử dụng lại**
- Quy ước:
 - Lớp mới \Leftrightarrow lớp dẫn xuất
 - Lớp đã có \Leftrightarrow lớp cơ sở
- Lớp dẫn xuất sẽ kế thừa các thuộc tính hay các phương thức từ các lớp cơ sở



Định nghĩa lớp dẫn xuất

❑ Cú pháp:

class tên-lớp-dẫn-xuất : **mode** tên-lớp-cơ-sở

{

....; //các thành phần lớp dẫn xuất, lớp mới



Định nghĩa lớp dẫn xuất

❑ **mode**: xác định chế độ kế thừa cho phép xác định các điều khiển, các hàm thành viên trong lớp cơ sở được xuất hiện như thế nào trong lớp dẫn xuất. Có thể nhận giá trị:

- **private**: tất cả các thành phần **public** hay **protected** của lớp cơ sở trở thành phần riêng **private** của lớp dẫn xuất ⇔ các thành phần **public** của lớp cơ sở chỉ truy cập được thông qua hàm thành phần của lớp dẫn xuất



Định nghĩa lớp dẫn xuất

- **public:** các thành phần chung (**public, protected**) của lớp cơ sở trở thành phần chung (**public, protected**) của lớp dẫn xuất \Leftrightarrow các đối tượng của lớp dẫn xuất có thể truy xuất đến thành phần **public** của lớp cơ sở
- **protected:** tất cả các thành phần **public** và **protected** của lớp cơ sở thành thành phần **protected** trong lớp dẫn xuất

Tổng hợp cơ chế kế thừa



Lớp cơ sở	Thùa kế public	Thùa kế private	Thùa kế protected
	Kết quả trong lớp dẫn xuất		
private	—	—	—
public	public	private	protected
protected	protected	private	protected

Bài tập áp dụng



SINHVIEN
<u>Dữ liệu:</u> - Ma sinh viên - Họ tên - Tuổi
<u>Thao tác:</u> -Nhập -Xuất

DIEM TONG KET
<u>Dữ liệu:</u> -Mon Toan -Mon Ly -Mon Hoa
<u>Phương thức</u>
- Nhập
- In
-Tính điểm trung bình
-Xếp loại

Áp dụng:

- Tạo một danh sách điểm tổng kết để quản lý điểm của Sinh viên
- In ra danh sách các sinh viên có điểm trung bình lớn hơn 8
- Đếm số sinh viên xếp loại khá



```
#include...
//Xay dung lop SINHVIEN
class SINHVIEN
{
protected:
    int MSV;
    char Hoten[20];
    int Tuoi;
public:
    void Nhap();
    void Xuat();
```



```
//dinh nghia thao tac nhap lop SINHVIEN
void SINHVIEN::Nhap()
{
    cout<<"Nhap ma sinh vien:"; cin>>MSV;
    cout<<"Nhap ho ten:"; cin>>Hoten;
    cout<<"Nhap tuoi:"; cin>>Tuoi;
}
```



```
//dinh nghia thao tac xuat lop SINHVIEN
void SINHVIEN::Xuat()
{
    cout<<"\n"<<setw(6)<<MSV;
    cout<<setw(15)<<Hoten;
    cout<<setw(4)<<Tuoi;
}
```



```
//Xay dung lop DIEMTONGKET  
class DTK : public SINHVIEN  
{  
    float toan, ly, hoa;  
public:  
    void Nhap();  
    void Xuat();  
    float Diem_TB();  
    *char Xeploai();  
};
```



```
//dinh nghia thao tac nhap lop DTK
void DTK::Nhap()
{
    SINHVIEN::Nhap();
    cout<<"Nhap diem toan:"; cin>>toan;
    cout<<"Nhap diem ly:"; cin>>ly;
    cout<<"Nhap diem hoa:"; cin>>hoa;
}
```



```
//dinh nghia thao tac xuat lop DTK
void DTK::Xuat()
{
    SINHVIEN::Xuat();
    cout<<setw(5)<<toan;
    cout<<setw(5)<<ly;
    cout<<setw(50)<<hoa;
}
```



```
//dinh nghia thao tac Tinh_DTB lop DTK
float DTK::Tinh_DTB()
{
    return (toan + ly + hoa)/3;
}
```



```
//dinh nghia thao tac Xeploai lop DTK
*char DTK::Xeploai ()
{
    if ( Tinh_DTB() >= 8 ) return "Gioi";
    else if ( Tinh_DTB() >= 7 ) return "Kha";
    if (Tinh_DTB()      >= 5) return "Trung
binh";
    else return "Yeu";
}
```



```
void main()
{
    DTK ds[50];      //Tao danh sach de luu DTK
    int n,i;

    cout<<"\n Nhap so Sinh vien:"; cin>>n;
    cout<<"\n NHAP THONG TIN TUNG SINH VIEN \n";
    for ( i=0 ; i<n ; i++ )
        ds[i].nhap();
```



```
cout<<"\n DS SINH VIEN DA NHAP\n";  
for ( i=0 ; i<n ; i++ )  
    ds[i].xuat();
```

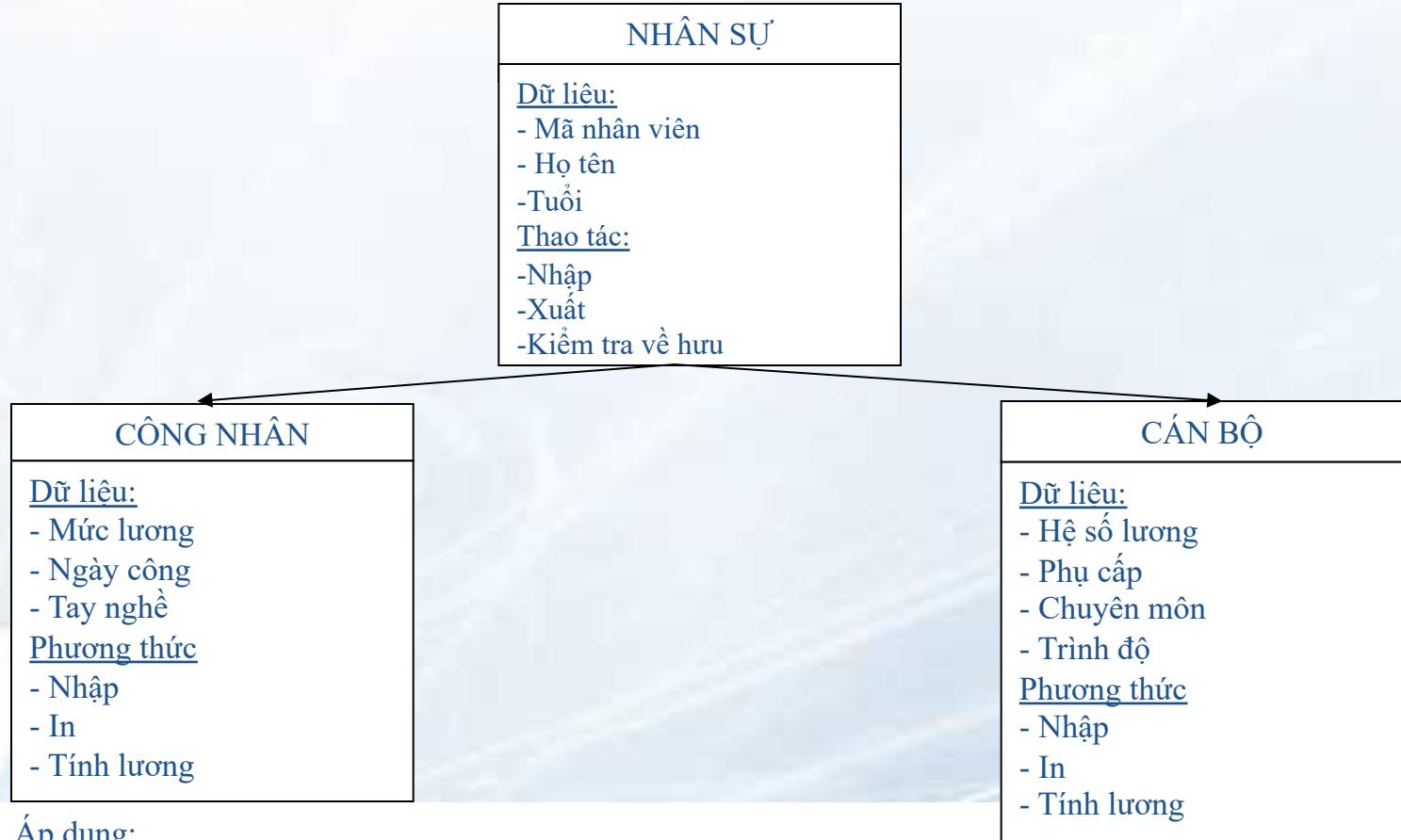
```
cout<<"\n DS SINH VIEN CO DTB > 8\n";  
for ( i=0 ; i<n ; i++ )  
    if (ds[i].Tinh_DTB() > 8 )  
        ds[i].xuat();
```



```
cout<<"\n DS SINH VIEN XEP LOAI KHA \n";
for ( i=0 ; i<n ; i++ )
    if ( strcmp( ds[i].Xeploai() , "Kha" ) == 0 )
        ds[i].xuat();

getch();
}
```

•Ví dụ áp dụng



Áp dụng:

- Tạo ra danh sách để quản lý công nhân và quản lý cán bộ
- In ra màn hình danh sách các công nhân đã đủ điều kiện về hưu
- In ra màn hình danh sách các cán bộ chưa đủ điều kiện về hưu
- In ra tiền lương cao nhất trong công nhân và tiền lương thấp nhất trong cán bộ

II. Phân loại kế thừa

1. Kế thừa đơn

- Là một lớp chỉ kế thừa từ một lớp đã có

- Ví dụ:

```
class B{  
    int a;  
public:  
    int b;  
    int get_a( );  
};  
class C : public B {  
    int c;  
public:  
    void mul( void ) { c=b*get_a( ); };  
};
```





2. Kế thừa đa mức

- Kế thừa đa mức là một lớp dẫn xuất kế thừa một lớp cơ sở nhưng thông qua lớp trung gian ở giữa

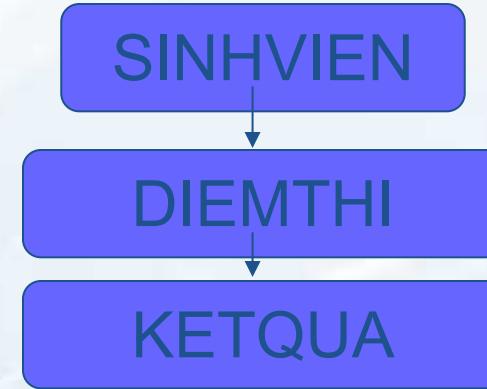
- Ví dụ:

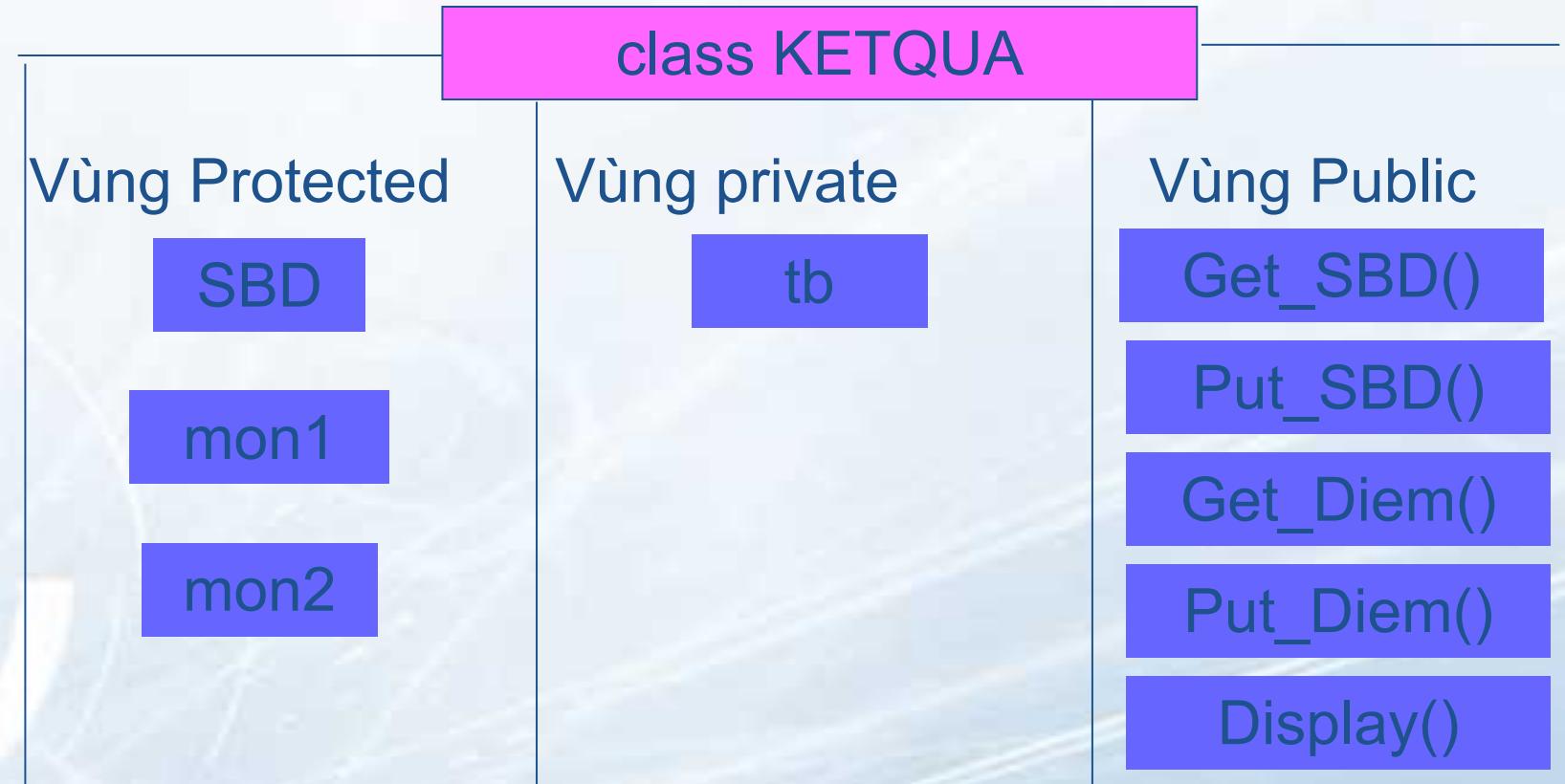
```
class SINHVIEN {  
    char *hoten;  
protected:  
    int SBD;  
public:  
    void get_SBD( int );  
    void put_SBD( void );  
};
```





```
class DIEMTHI : public SINHVIEN {  
protected:  
    float mon1, mon2;  
public:  
    void get_Diem( float , float );  
    void put_Diem( void );  
};  
class KETQUA : public DIEMTHI {  
    float tb;  
public:  
    void Display( void );  
};
```

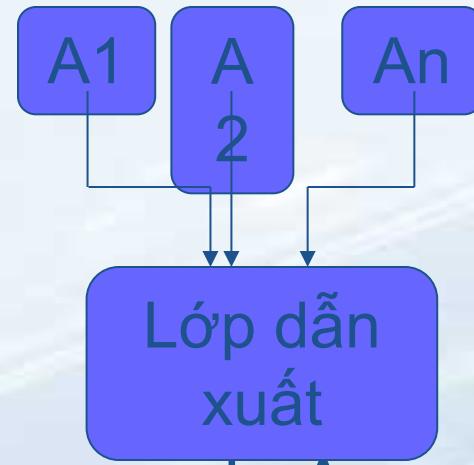






3. Kế thừa bội

- Là một lớp dẫn xuất kế thừa của nhiều lớp cơ sở cùng lúc \Leftrightarrow kết hợp đặc trưng của các lớp để tạo lớp mới



•Cú pháp:

class Lớp-dẫn-xuất : mode A1, mode A2, ..., mode An

{

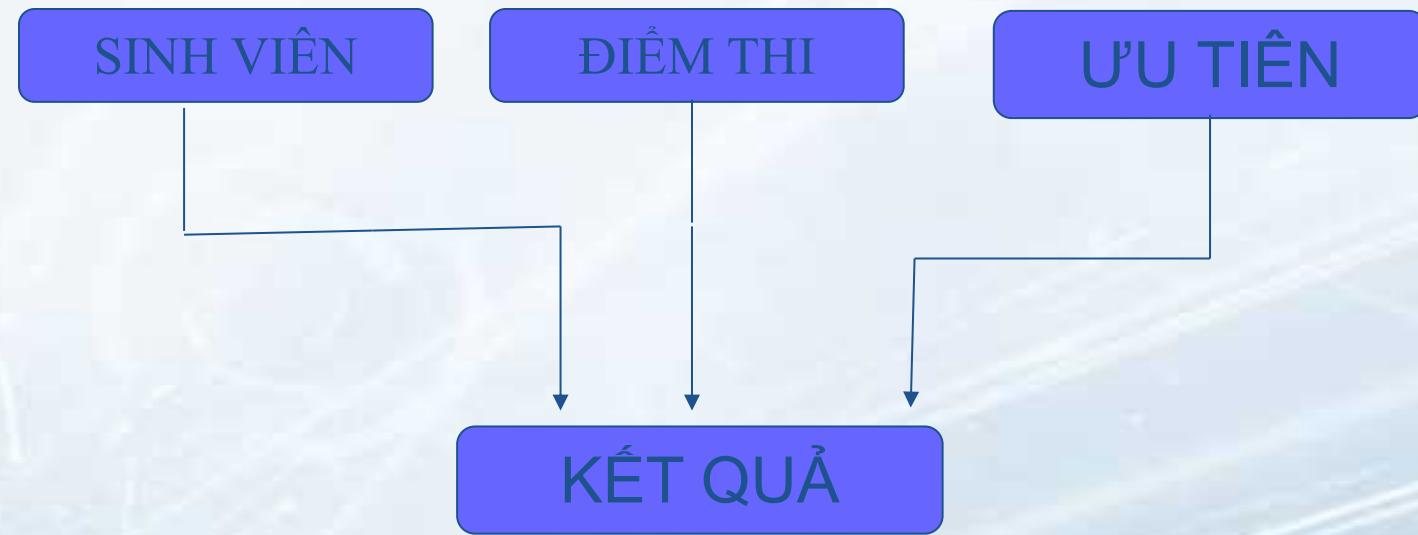
.....; //định nghĩa lớp mới

};

Chú ý: mỗi lớp sẽ có một chế độ kế thừa khác nhau



✓ Ví dụ:





4. Kế thừa kép:

- ✓ Là kế thừa bằng kết hợp nhiều loại kế thừa để tạo ra lớp dẫn xuất mới





IV. Cấu tử và hủy tử trong lớp dẫn xuất

- ❑ Trong lớp dẫn xuất kế thừa các thành viên của lớp cơ sở **ngoại trừ cấu tử và hủy tử**
- ❑ Các cấu tử và hủy tử của lớp dẫn xuất vẫn có thể gọi các **cấu tử và hủy tử** của lớp cơ sở để khởi gán
 - Khi xây dựng cấu tử của lớp dẫn xuất thì phải thực hiện khởi gán các thành phần của lớp cơ sở trước (gọi cấu tử của lớp cơ sở trước), sau đó mới thực hiện khởi gán các thành phần trong lớp dẫn xuất
 - Các hủy tử được gọi theo thứ tự ngược lại => các hủy tử của lớp dẫn xuất thực hiện trước, sau đó đến các hủy tử của lớp cơ sở



* Một số nguyên tắc

- Nếu trong lớp cơ sở không có câu tử có tham số thì trong lớp dẫn xuất không bắt buộc xây dựng câu tử trong lớp dẫn xuất
- Nếu trong lớp cơ sở chỉ có câu tử có tham số thì trong lớp dẫn xuất bắt buộc phải xây dựng câu tử trong lớp dẫn xuất
- Cả lớp cơ sở và lớp dẫn xuất đều có tham biến => câu tử lớp cơ sở được thực hiện sau đó đến câu tử lớp dẫn xuất
- Kế thừa bội thì các câu tử được thực hiện lần lượt theo thứ tự khai báo



* Ví dụ

- Xây dựng lớp POINT với hai thành phần dữ liệu x,y. *Yêu cầu xây dựng hàm cấu tử và hủy tử*
- Xây dựng lớp TAMGIACT kế thừa từ lớp POINT với thành phần thể hiện tọa độ của tam giác. *Yêu cầu xây dựng cấu tử và hủy tử tương ứng cho hình tròn*



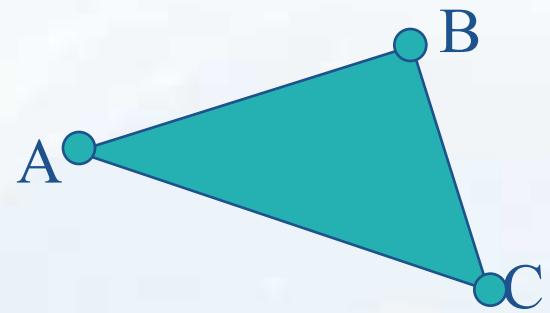
```
class DIEM
{
protected:
    int x;
    int y;
public:
    DIEM( int xx, int yy);
    void nhap( );
    void in( );
};

DIEM::DIEM( int xx, int yy)
{
    x = xx;
    y = yy;
}
```



```
class HINHTG : public DIEM
{
private:
    DIEM B;                                //Tọa độ điểm B
    int Cx, Cy;                            // Tọa độ điểm C
public:
    TAMGIAC( int Axx, int Ayy, int Bxx, int Byy, int Cxx, int Cyy );
    void nhap();
    void in();
};

TAMGIAC::TAMGIAC( int Axx, int Ayy, int Bxx, int Byy, int Cxx,
int Cyy ) : DIEM(Axx, Ayy) , B(Bxx, Byy)
{
    Cx = Cxx;
    Cy = Cyy;
}
```





*Chú ý:

❑ Khi thực hiện kế thừa các thành phần dữ liệu của lớp dẫn xuất gồm các loại:

- thành phần mới khai báo (Cx, Cy)
- thành phần kế thừa từ lớp cơ sở (*Tọa độ đỉnh A – Ax, Ay*)
- thành phần có kiểu là đối tượng của lớp đã định nghĩa (*đỉnh B – Bx, By*)

❑ Tùy theo loại thành phần dữ liệu khác nhau, cách xây dựng hàm tạo là khác nhau:

- nếu thành phần mới: dùng câu lệnh gán trực tiếp trong thân hàm cấu tử
- nếu thành phần kế thừa từ lớp cơ sở: dùng hàm cấu tử của lớp cơ sở để khởi gán và viết trên dòng tên hàm cấu tử của lớp dẫn xuất $\Leftrightarrow \text{DIEM}(Axx, Ayy)$
- nếu thành phần có kiểu là đối tượng của lớp đã có: dùng trực tiếp tên của đối tượng được khai báo để khởi gán và cùng viết trên dòng tên hàm cấu tử $\Leftrightarrow \text{B(Bxx, Byy)}$;



V. Toán tử gán cho lớp dẫn xuất

- ✓ Khi trong lớp dẫn xuất có tồn tại thuộc tính dưới dạng con trỏ (kể cả thuộc tính kế thừa từ lớp cơ sở) thì bắt buộc phải xây dựng toán tử gán không được sử dụng toán tử gán mặc định



*Ví dụ định nghĩa toán tử gán của lớp dẫn xuất:

```
class A
{
.....
    A operator =(A & h)
    {
        //thực hiện định nghĩa toán tử gán của lớp A
    }
    A* get_A() { return this; }
};

class B:public A
{ .....
    B& operator=(B&h)
    {
        A *u1, *u2;
        u1= this->get_A();
        u2 = h.get_A();
        *u1= *u2;
    }
};
```

Cách xây dựng



Bước 1: xây dựng toán tử gán cho lớp cơ sở.

Bước 2: Xây dựng phương thức (trong các lớp cơ sở) để nhận địa chỉ của đối tượng ân của lớp:

```
Tên_lớp_cơ_sở* get_DT()
{
    return this;
}
```

Bước 3: xây dựng toán tử gán cho lớp dẫn xuất \Leftrightarrow dùng phương thức trên để nhận địa chỉ của đối tượng lớp cơ sở mà lớp dẫn xuất kế thừa, sau đó thực hiện lệnh gán trên 2 đối tượng này

* Ví dụ:

```
class DIEM
{
    private:
        int x;
        int y;
    public:
        DIEM operator = ( DIEM &h )
        {
            x = h.x;
            y = h.y;
        }
        DIEM *get_DIEM( )
        {
            return this;
        }
};
```





```
class HCN : public DIEM
{
    private:
        int Cx, Cy;
    public:
        HCN operator = ( HCN &h)
        {
            DIEM *u1, *u2;
            u1 = this -> get_DIEM( );
            u2 = h.get_DIEM( );
            *u1 = *u2;
            Cx = h.Cx;
            Cy = h.Cy;
        }
}
```



VI. Hàm tạo sao chép của lớp dẫn xuất

- ❑ Tương tự như toán tử gán, khi các thuộc tính (kể cả các thuộc tính kế thừa) là con trỏ cần xây dựng hàm tạo sao chép để tạo ra đối tượng mới giống và độc lập với đối tượng đã cho.
- ❑ Cách xây dựng:

B1: Xây dựng toán tử gán cho lớp dẫn xuất

B2: Xây dựng hàm tạo sao chép cho lớp dẫn xuất theo mẫu:

Tên_lớp_dẫn_xuất(Tên_lớp_dẫn_xuất & h)

{

***this = h;**

}

VII. Sự nhập nhằng trong đa kế thừa



```
class OptionList {  
public:  
    // .....  
    void Highlight (int part);  
};
```

```
class Window {  
public:  
    // .....  
    void Highlight (int part);  
};
```

```
class Menu : public OptionList,  
             public Window  
{ ..... };
```

Gọi
hàm
của lớp
nào?

```
void main() {  
    Menu m1(...);  
    m1.Highlight(10);  
    ....  
}
```

Hàm cùng tên

Chỉ rõ hàm
của lớp nào

xử lý

```
void main() {  
    Menu m1(...);  
    m1.OptionList::Highlight(10);  
    m1.Window::Highlight(20);  
    ....  
}
```



- ❑ Trong đa kế thừa xảy ra trường hợp hai lớp cơ sở của một lớp dẫn xuất có cùng tên thành phần (dữ liệu và phương thức) => Trình biên dịch sẽ không có khả năng hiểu thành phần được sử dụng khi lớp dẫn xuất thực hiện lời gọi \Leftrightarrow **dẫn đến sự nhập nhằng**
- ❑ Để tránh nhập nhằng và để cho máy hiểu được thì phải chỉ rõ thành phần được truy cập (sử dụng) thuộc lớp cơ sở nào?

*Ví dụ:



```
class A1
{
    ...
public:
    void Nhap();
};

class A2
{
    ...
public:
    void Nhap();
};
```

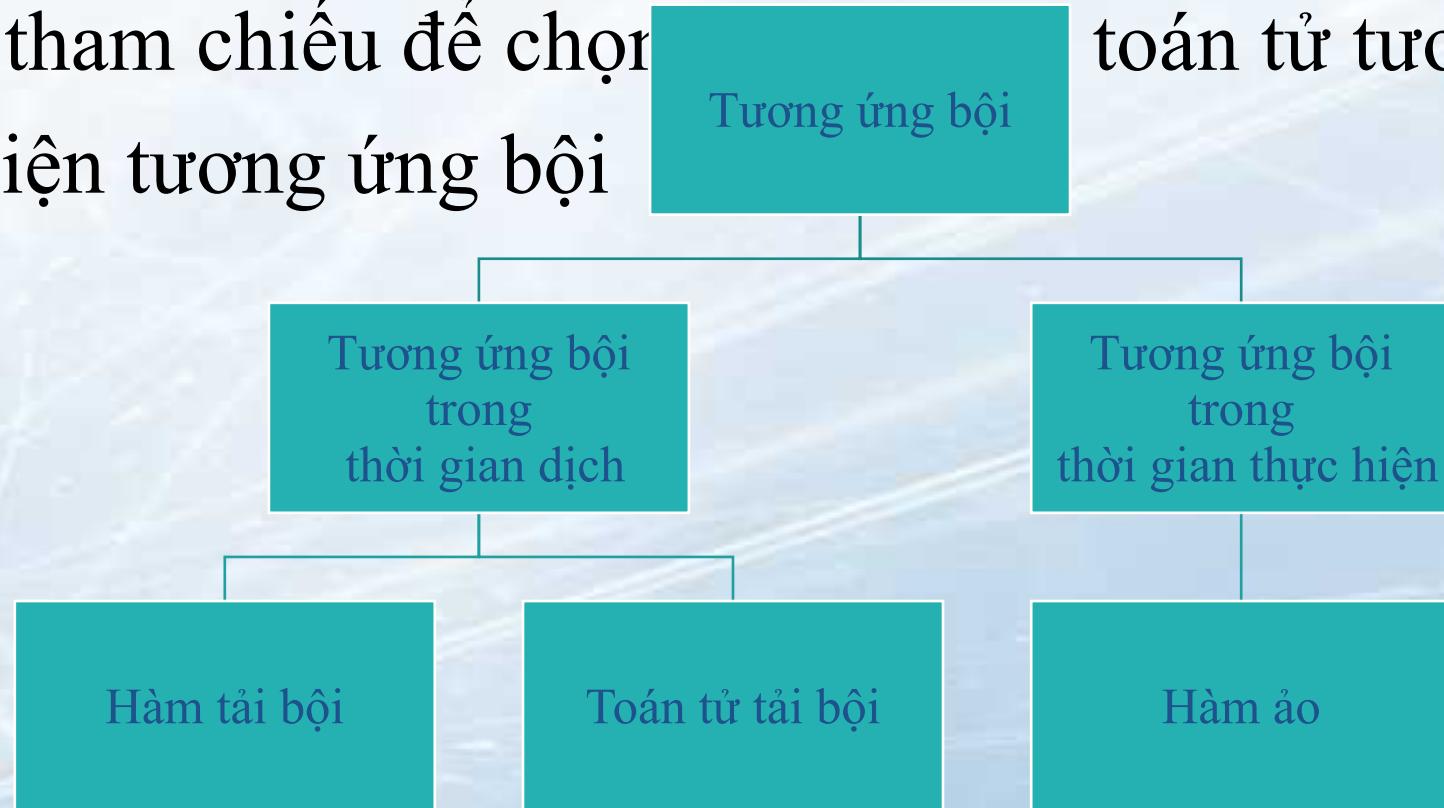
```
class B : public A1, public A2
{
    ...
};
```

Khi sử dụng:
B b1;
b1.Nhap(); (*xảy ra nhập
nhằng*)



X. Tương ứng bội và hàm ảo

- ❑ Tương ứng bội là khả năng sử dụng một tên dưới nhiều dạng khác nhau.
- ❑ Khi gọi, hàm hay toán tử tải bội sẽ được phân tích, so sánh về kiểu, số lượng tham chiếu để chọn toán tử tương ứng
- ❑ Cơ chế thực hiện tương ứng bội





□ Liên kết tĩnh (*static binding*):

- Xác định khi biên dịch chương trình.
- Dùng **hàm thành viên**.
- Gọi hàm của lớp cơ sở

```
class ContactDir {  
    //...  
public:  
    int Lookup (const char *name);  
    //...  
};
```

```
class SortedDir : public ContactDir {  
public:  
    SortedDir(const int max) : ContactDir(max) {}  
    int Lookup(const char *name);  
};
```

```
void main() {  
    ContactDir c1(10);  
    SortedDir *p; p = &c1;
```

```
    cout<<p->Lookup("ABC"); •••  
    ....  
}
```

Gọi
hàm
nào ?



❑ Liên kết động (*dynamic binding*)

- Xác định khi thực thi chương trình.
- Dùng **hàm ảo** (virtual function).
- Gọi hàm của lớp **dẫn xuất**
- Thể hiện tính **đa hình** của OOP.

```
class ContactDir {  
    //...  
public:  
    virtual int Lookup (const char *name);  
};
```

```
class SortedDir : public ContactDir {  
    //....  
public:  
    int Lookup(const char *name);  
};
```

```
void main() {  
    ContactDir c1(10);  
    SortedDir *p1; p1 = &c1;  
    cout<<p->Lookup("ABC");  
  
    SortedDir c2(20);  
    ContactDir *p2; p2 = &c2;  
    cout<<p->Lookup("ABC");  
}
```

Gọi hàm
của lớp
nào ?

Kết quả
trên
màn hình
là gì?



- ✓ **Hàm ảo** là hàm sử dụng cùng một tên trong cả lớp cơ sở và lớp dẫn xuất.
- ✓ Khai báo hàm ảo:
 - ✓ thêm từ khóa **virtual** trước khai báo hàm trong lớp cơ sở



❑ Ví dụ dùng hàm ảo:





```
class Hinh  
{  
public:  
    Hinh();  
    virtual double Dientich( ) const=0;  
};
```



```
class Hinhtron :: public Hinh
{
private:
    double R;
public:
    Hinhtron( );
    Hinhtron( double );
    double Dientich()
    {
        return (3.14*R*R);
    }
}
```



```
class Hinhvuong:: public Hinh
{
private:
    double a,b;
public:
    Hinhvuong();
    Hinhvuong(double, double);
    double Dientich()
    {
        return (a*b);
```



❑ Đặc trưng của hàm ảo:

- *hàm ảo không thể là các hàm thành viên tĩnh (không là static)*
- *một hàm ảo có thể được khai báo là friend trong một lớp nhưng các hàm friend trong một lớp thì không thể là hàm ảo*
- *không cần phải thêm từ khóa virtual khi định nghĩa phương thức ảo trong lớp dẫn xuất*

❑ Chú ý:

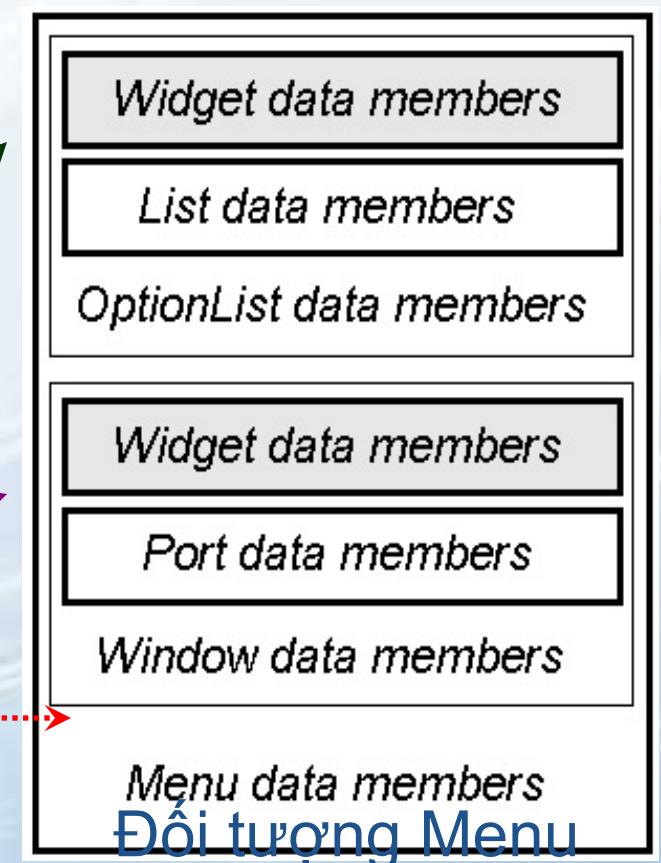
- *hàm cấu tử không thể là hàm ảo nhưng cấu tử có thể gọi tới các hàm ảo khác*
- *Hàm hủy tử có thể là hàm ảo*
- *toán tử có thể là hàm ảo*

* Đặt vấn đề:



● Sự mơ hồ - dư thừa dữ liệu

```
class OptionList  
    : public Widget, List  
{ /*...*/ };  
  
class Window  
    : public Widget, Port  
{ /*...*/ };  
  
class Menu  
    : public OptionList,  
      public Window  
{ /*...*/ };
```



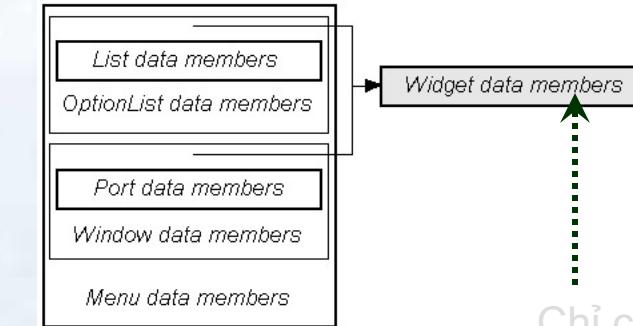


❑ Cách xử lý: dùng lớp cơ sở ảo.

```
class OptionList
    : virtual public Widget,
      public List
    { /*...*/ };

class Window
    : virtual public Widget,
      public Port
    { /*...*/ };

class Menu
    : public OptionList,
      public Window
    { /*...*/ };
```



Chỉ có 1
đối tượng Widget

```
Menu::Menu (int n, Rect &bounds) :
    Widget(bounds), OptionList(n), Window(bounds)
{ //... }
```



- ❑ Trong kế thừa bội có thể xảy ra một lớp dẫn xuất cùng kế thừa tới một lớp cơ sở nhưng nhiều lần => **hiểu nhầm**
- ❑ Để tránh máy không hiểu nhầm thì phải khai báo lớp cơ sở được kế thừa nhiều lần là lớp cơ sở ảo => Thêm từ khóa virtual khi xây dựng lớp mới kế thừa từ lớp ảo

```
class A
{
    ...
};

class B : virtual public A
{
    ....
}
```

```
class C : virtual public A
{
    ....
};

class D : public B, public C
{
    ....
}
```



XI. Lớp cơ sở trừu tượng

- ✓ Lớp trừu tượng để mô tả một cách tổng quát của lớp trong hệ thống
- ✓ Lớp trừu tượng thường dùng để xác định cái gì phải làm những không chỉ rõ công việc đó được thực hiện như thế nào?
- ✓ Là lớp chỉ được dùng làm cơ sở cho lớp khác, được dùng để định nghĩa khái niệm tổng quát
- ✓ Không có đối tượng nào của lớp trừu tượng được tạo ra



TRƯỜNG ĐẠI HỌC THỦY LỢI
Khoa CNTT – Bộ môn CNPM

NGUYÊN LÝ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

CSE 224

Chương 5: Hàm ảo và tính đa hình

Giảng viên: Cù Việt Dũng

Email: dungcv@tlu.edu.vn

ĐT: 0964.644.986

NỘI DUNG



- 5.1 Đa hình trong kế thừa
- 5.2 Thực thi hàm ảo
- 5.3 Con trỏ và hàm ảo

NỘI DUNG



- 5.1 Đa hình trong kế thừa
- 5.2 Thực thi hàm ảo
- 5.3 Con trỏ và hàm ảo

Sự nhập nhằng trong đa kế thừa



```
class OptionList {  
public:  
    // .....  
    void Highlight (int part);  
};
```

```
class Window {  
public:  
    // .....  
    void Highlight (int part);  
};
```

```
class Menu : public OptionList,  
            public Window  
{ ..... };
```

Hàm cùng tên

Chỉ rõ hàm
của lớp nào

Gọi
hàm
của lớp
nào?

```
void main() {  
    Menu m1(...);  
    m1.Highlight(10);  
    ....  
}
```

xử lý

```
void main() {  
    Menu m1(...);  
    m1.OptionList::Highlight(10);  
    m1.Window::Highlight(20);  
    ....  
}
```



Sự nhập nhằng trong đa kế thừa

- ❑ Trong đa kế thừa xảy ra trường hợp hai lớp cơ sở của một lớp dẫn xuất có cùng tên thành phần (dữ liệu và phương thức) => Trình biên dịch sẽ không có khả năng hiểu thành phần được sử dụng khi lớp dẫn xuất thực hiện lời gọi \Leftrightarrow **dẫn đến sự nhập nhằng**
- ❑ Để tránh nhập nhằng và để cho máy hiểu được thì phải chỉ rõ thành phần được truy cập (sử dụng) thuộc lớp cơ sở nào?

*Ví dụ:



```
class A1
{
    ...
    public:
        void Nhap();
};

class A2
{
    ...
    public:
        void Nhap();
};
```

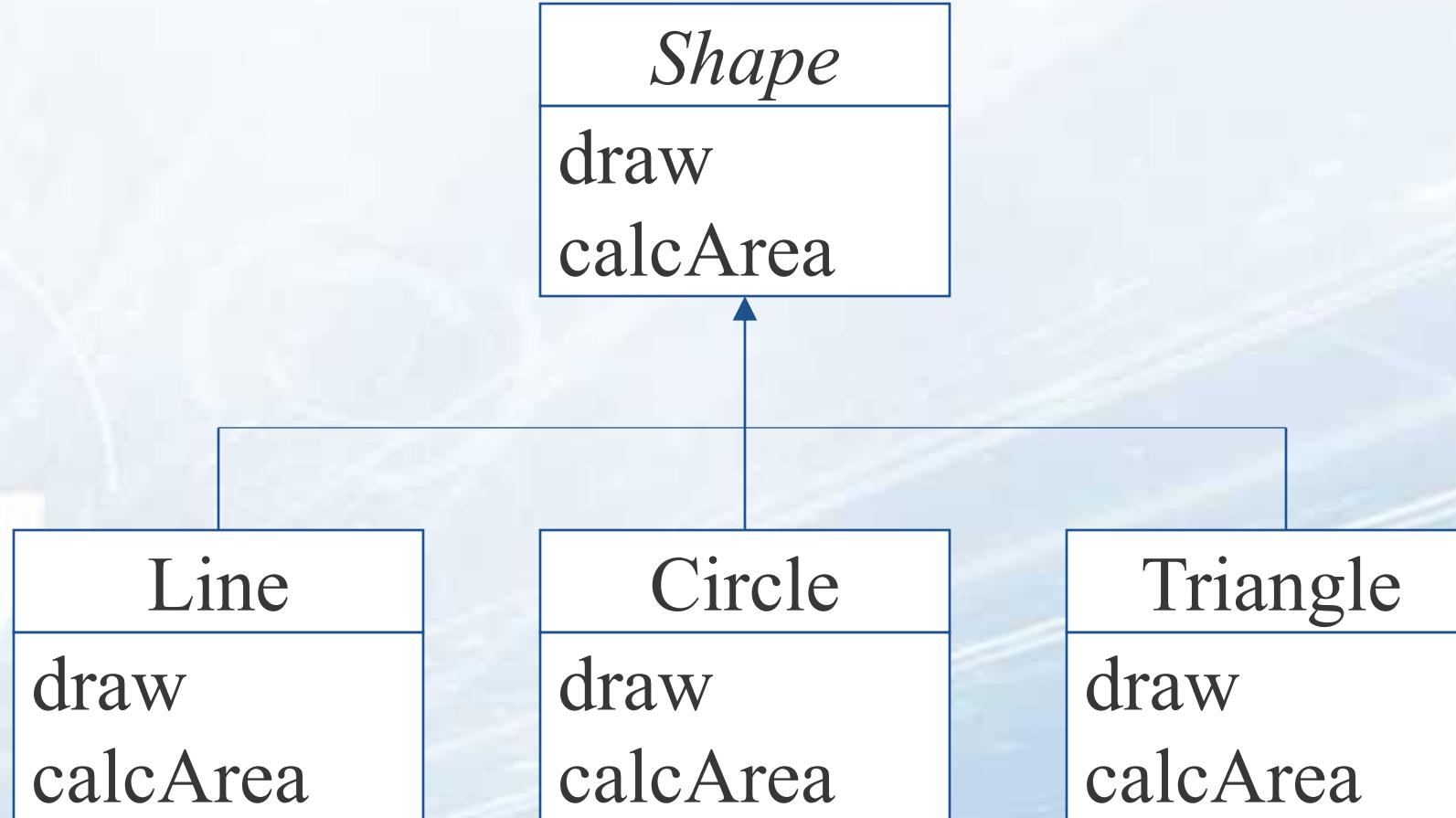
```
class B : public A1, public A2
{
    ...
};
```

Khi sử dụng:
B b1;
b1.Nhap(); (*xảy ra nhập nhằng*)
↔
b1.A1::Nhap();
b1.A2::Nhap();



Shape Hierarchy

Phát triển một hàm có thể vẽ các loại hình hình học khác nhau từ một mảng





Shape Hierarchy

```
class Shape {  
    ...  
protected:  
    char _type;  
public:  
    Shape() {}  
    void draw() { cout << "Shape\n"; }  
    int calcArea() { return 0; }  
    char getType() { return _type; }  
}
```

... Shape Hierarchy



```
class Line : public Shape {  
    ...  
public:  
    Line(Point p1, Point p2) {  
        ...  
    }  
    void draw(){ cout << "Line\n"; }  
}
```

... Shape Hierarchy



```
class Circle : public Shape {  
    ...  
public:  
    Circle(Point center, double radius) {  
        ...  
    }  
    void draw() { cout << "Circle\n"; }  
    int calcArea() { ... }  
}
```

... Shape Hierarchy



```
class Triangle : public Shape {  
    ...  
public:  
    Triangle(Line l1, Line l2,  
              double angle)  
    { ... }  
    void draw(){ cout << "Triangle\n"; }  
    int calcArea() { ... }  
}
```



Drawing a Scene

```
int main() {  
    Shape* _shape[ 10 ] ;  
    Point p1(0, 0) , p2(10, 10) ;  
    shape[1] = new Line(p1, p2) ;  
    shape[2] = new Circle(p1, 15) ;  
    ...  
    void drawShapes( shape, 10 ) ;  
    return 0 ;  
}
```



Function drawShapes()

```
void drawShapes(Shape* _shape[],  
                int size) {  
    for (int i = 0; i < size; i++) {  
        _shape[i]->draw();  
    }  
}
```

Sample Output



Shape

Shape

Shape

Shape

...



Function drawShapes()

```
void drawShapes (  
    Shape* _shape[], int size) {  
    for (int i = 0; i < size; i++) {  
        // Determine object type with  
        // switch & accordingly call  
        // draw() method  
    }  
}
```



Sử dụng Switch

```
switch ( _shape[i]->getType() )  
{  
    case 'L':  
        static_cast<Line*>(_shape[i])->draw();  
        break;  
    case 'C':  
        static_cast<Circle*>(_shape[i])  
            ->draw();  
        break;  
    ...  
}
```

Sử dụng If



```
if ( _shape[i]->getType() == 'L' )  
    static_cast<Line*>(_shape[i])->draw();  
else if ( _shape[i]->getType() == 'C' )  
    static_cast<Circle*>(_shape[i])->draw();  
...  
...
```

Sample Output



Line

Circle

Triangle

Circle

...



Function printArea

Hãy xem xét một hàm in diện tích của mỗi hình dạng từ một mảng đầu vào

```
void printArea(  
    Shape* _shape[], int size) {  
    for (int i = 0; i < size; i++) {  
        // Print shape name.  
        // Determine object type with  
        // switch & accordingly call  
        // calcArea() method.  
    }  
}
```



Required Switch Logic

```
switch ( _shape[i]->getType() )  
{  
    case 'L':  
        static_cast<Line*>(_shape[i])  
            ->calcArea();      break;  
    case 'C':  
        static_cast<Circle*>(_shape[i])  
            ->calcArea();      break;  
    ...  
}
```

...Phân tích code



- ❑ Đoạn mã “switch” ở trên giống như trong hàm drawArray ()
- ❑ Hơn nữa, chúng ta có thể cần vẽ các hình dạng hoặc tính toán diện tích ở nhiều vị trí trong code

Other Problems



- ❑ Lập trình viên có thể quên kiểm tra
- ❑ Có thể quên kiểm thử tất cả các trường hợp có thể xảy ra
- ❑ Khó bảo trì

Giải pháp?



- ❑ Để giải quyết, chúng ta cần cơ chế tự động lựa chọn -> **tính đa hình**
- ❑ Trong mô hình OO, tính đa hình có nghĩa là các đối tượng khác nhau có thể hoạt động theo những cách khác nhau cho cùng một thông điệp (kích thích)
- ❑ Do đó, người gửi tin nhắn không cần biết chính xác loại người nhận

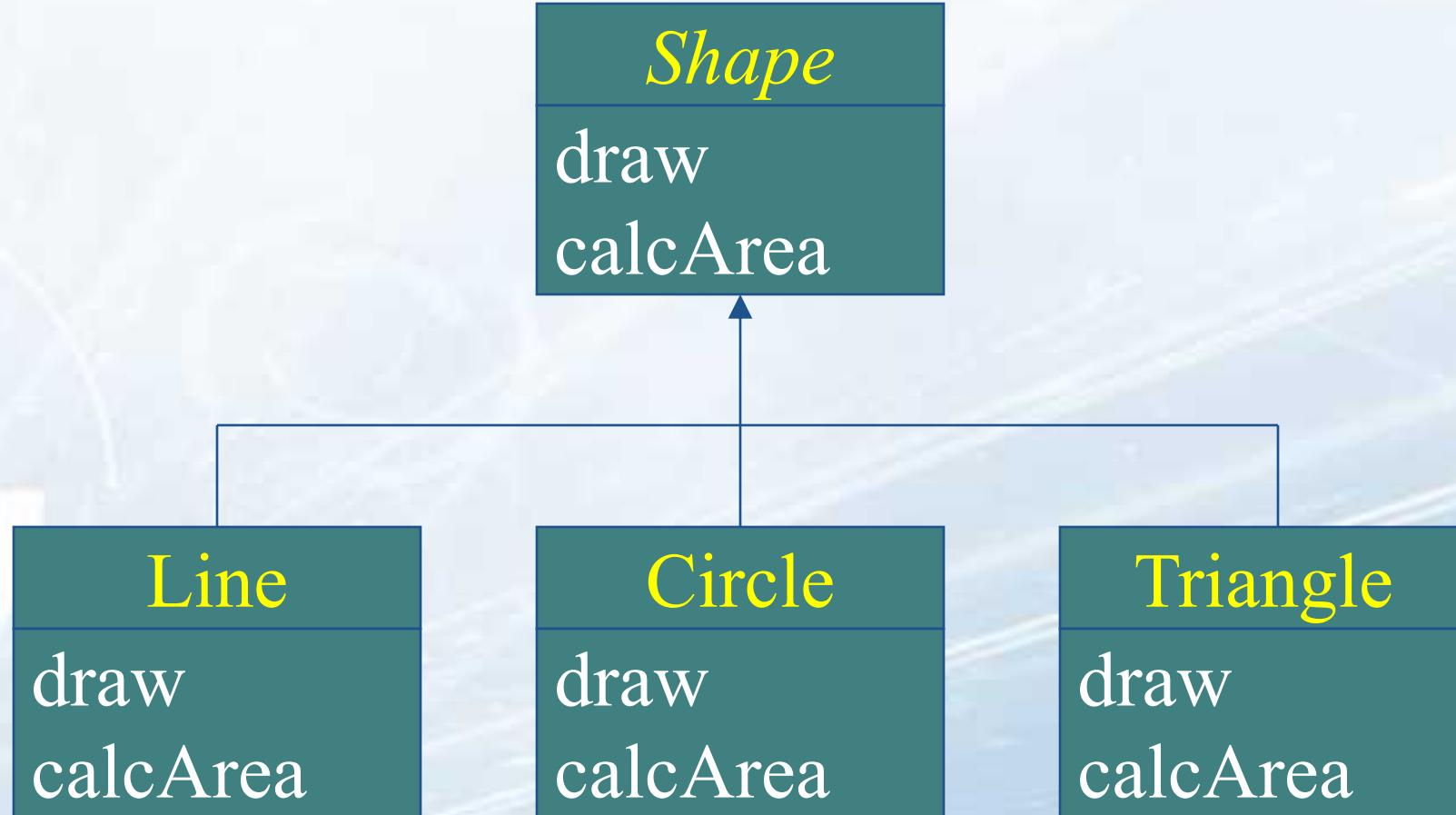


Virtual Functions

- ❑ Mục tiêu của 1 lệnh gọi hàm ảo được xác định tại thời điểm chạy
- ❑ Trong C++, khai báo một hàm ảo bằng cách đặt trước tiêu đề hàm với từ khóa “virtual”

```
class Shape {  
    ...  
    virtual void draw();  
}
```

Shape Hierarchy



...Shape Hierarchy Revisited



```
class Shape {  
    ...  
    virtual void draw();  
    virtual int calcArea();  
}  
  
class Line : public Shape {  
    ...  
    virtual void draw();  
}
```

Không định nghĩa

... Shape Hierarchy Revisited



```
class Circle : public Shape {  
    ...  
    virtual void draw();  
    virtual int calcArea();  
}  
  
class Triangle : public Shape {  
    ...  
    virtual void draw();  
    virtual int calcArea();  
}
```

Function drawShapes()



```
void drawShapes(Shape* _shape[],  
                int size) {  
    for (int i = 0; i < size; i++) {  
        _shape[i]->draw();  
    }  
}
```

Sample Output



Line

Circle

Triangle

Circle

...



Function printArea

```
void printArea(Shape* _shape[],  
    int size) {  
    for (int i = 0; i < size; i++) {  
        // Print shape name  
        cout<< _shape[i]->calcArea();  
        cout << endl;  
    }  
}
```

Static vs Dynamic Binding



- ❑ Liên kết tĩnh có nghĩa là hàm đích cho một cuộc gọi được chọn tại thời điểm biên dịch
- ❑ Liên kết động có nghĩa là hàm đích cho một cuộc gọi được chọn vào thời gian chạy

Liên kết tĩnh



□ Liên kết tĩnh (*static binding*):

- Xác định khi biên dịch chương trình.
- Dùng **hàm thành viên**.
- Gọi hàm của lớp cơ sở

```
class Pet {  
    Pet(string n) { Name = n; }  
    public:  
        string getSound(){  
            return "Say it";  
        }  
};
```

```
class Cat : public Pet {  
    public:  
        Cat(string n) : Pet(n) {}  
        string getSound() {  
            return "Meow! Meow!";  
        }  
};
```

```
void main() {  
    Pet *a_pet = new Cat("Kitty");  
}
```

```
cout << a_pet-> getSound();
```

Gọi
hàm
nào ?

Liên kết động



□ Liên kết động (*dynamic binding*)

- Xác định khi thực thi chương trình.
- Dùng **hàm ảo** (virtual function).
- Gọi hàm của lớp **dẫn xuất**
- Thể hiện tính **đa hình** của OOP.

```
class Pet {  
    Pet(string n) { Name = n; }  
public:  
    string getSound(){  
        return "Say it";  
    }  
    void makeSound() {  
        cout << Name << " says: ";  
        cout<< getSound() << endl;  
    }  
};
```

```
class Cat : public Pet {  
public:  
    Cat(string n) : Pet(n) {}  
    string getSound() {  
        return "Meow! Meow!";  
    }  
};
```

```
void main() {  
    Pet *a_pet = new Cat("Kitty");  
    cout<< a_pet-> getSound();  
    a_pet->makeSound();  
}
```

Gọi
hàm
của
lớp
nào ?

Kết quả
trên
màn hình
là gì?



Static vs Dynamic Binding

```
Line _line;  
_line.draw();      // Always Line::draw called  
Shape* _shape = new Line();  
_shape->draw(); // Shape::draw called if draw() is  
not virtual
```

```
Shape* _shape = new Line();  
_shape->draw(); // Line::draw called if draw() is  
virtual
```

NỘI DUNG



- 5.1 Đa hình trong kế thừa
- **5.2 Thực thi hàm ảo**
- 5.3 Con trỏ và hàm ảo



Hàm ảo

- ✓ **Hàm ảo** là hàm sử dụng cùng một tên trong cả lớp cơ sở và lớp dẫn xuất.
- ✓ Khai báo hàm ảo:
 - ✓ thêm từ khóa **virtual** trước khai báo hàm trong lớp cơ sở

Ví dụ



❑ Ví dụ dùng hàm ảo:



Ví dụ



```
class Hinh
{
public:
    Hinh();
    virtual double Dientich() {return 0;};
};
```

Ví dụ



```
class Hinhtron :: public Hinh
{
private:
    double R;
public:
    Hinhtron( );
    Hinhtron( double );
    double Dientich( )
    {
        return (3.14*R*R);
    }
};
```



```
class Hinhvuong:: public Hinh
{
private:
    double a,b;
public:
    Hinhvuong();
    Hinhvuong(double, double);
    double Dientich()
    {
        return (a*b);
    }
};
```



Đặc trưng hàm ảo

❑ Đặc trưng của hàm ảo:

- *hàm ảo không thể là các hàm thành viên tĩnh (không là static)*
- *một hàm ảo có thể được khai báo là friend trong một lớp nhưng các hàm friend trong một lớp thì không thể là hàm ảo*
- *không cần phải thêm từ khóa virtual khi định nghĩa phương thức ảo trong lớp dẫn xuất*

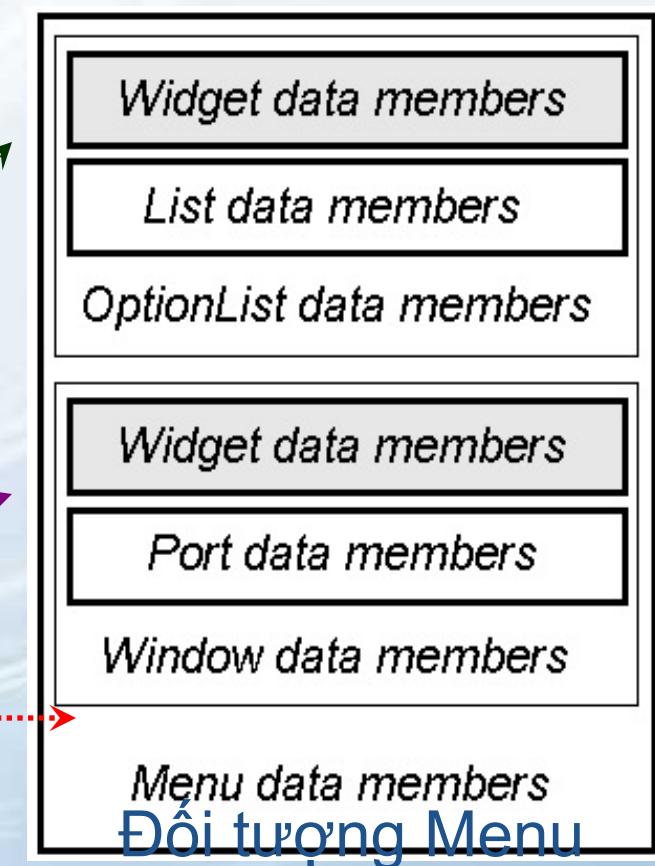
❑ Chú ý:

- *hàm cấu tử không thể là hàm ảo nhưng cấu tử có thể gọi tới các hàm ảo khác*
- *Hàm hủy tử có thể là hàm ảo*
- *toán tử có thể là hàm ảo*



● Sự mơ hồ - dư thừa dữ liệu

```
class OptionList  
    : public Widget, List  
{ /*...*/ };  
  
class Window  
    : public Widget, Port  
{ /*...*/ };  
  
class Menu  
    : public OptionList,  
      public Window  
{ /*...*/ };
```

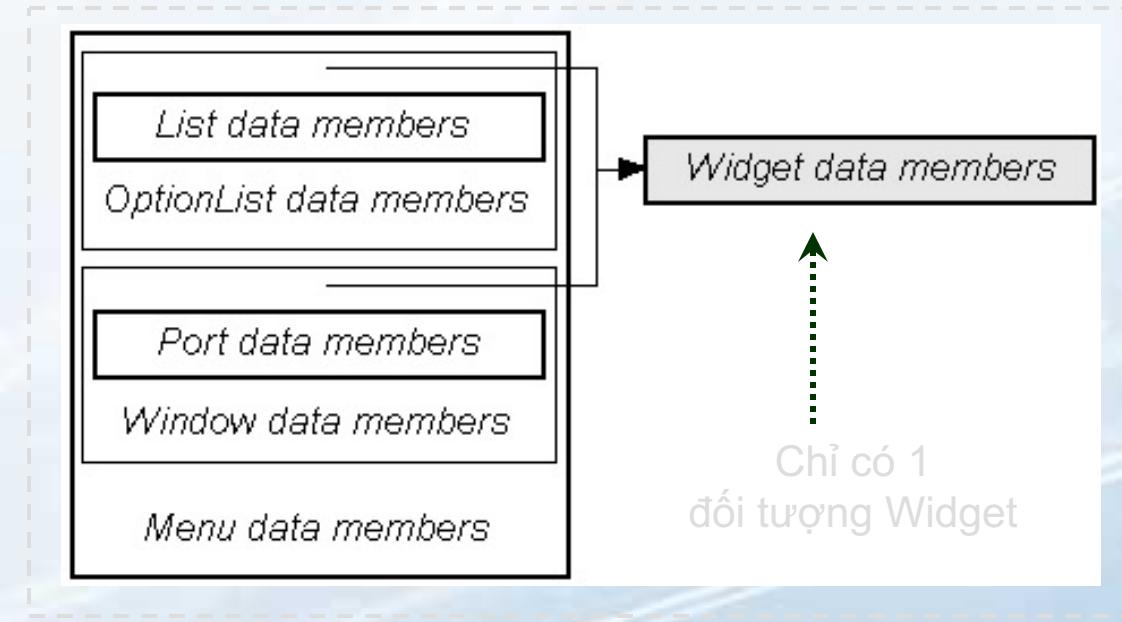


Đặt vấn đề



❑ Cách xử lý: dùng lớp cơ sở ảo.

```
class OptionList  
    : virtual public Widget,  
      public List  
    { /*...*/ };  
  
class Window  
    : virtual public Widget,  
      public Port  
    { /*...*/ };  
  
class Menu  
    : public OptionList,  
      public Window  
    { /*...*/ };
```



```
Menu::Menu (int n, Rect &bounds) :  
    Widget(bounds), OptionList(n), Window(bounds)  
{ //... }
```



- Trong kế thừa bội có thể xảy ra một lớp dẫn xuất cùng kế thừa tới một lớp cơ sở nhưng nhiều lần => **hiểu nhầm**
- Để tránh máy không hiểu nhầm thì phải khai báo lớp cơ sở được kế thừa nhiều lần là lớp cơ sở ảo => Thêm từ khóa virtual khi xây dựng lớp mới kế thừa từ lớp ảo

```
class A
{
    ...
};

class B : virtual public A
{
    ....
}
```

```
class C : virtual public A
{
    ....
};

class D : public B, public C
{
    ....
}
```



TRƯỜNG ĐẠI HỌC THỦY LỢI
Khoa CNTT – Bộ môn CNPM

NGUYÊN LÝ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

CSE 224

Chương 6: Khuôn mẫu (Template) và thư viện chuẩn (STL)

Giảng viên: Cù Việt Dũng

Email: dungcv@tlu.edu.vn

ĐT: 0964.644.986

Nội dung



- 6.1 Nhắc lại về vector
- 6.2 Làm việc với c-string và string
- 6.3 Khuôn mẫu hàm
- 6.4 Khuôn mẫu lớp

Nội dung



- 6.1 Nhắc lại về vector
- 6.2 Làm việc với c-string và string
- 6.3 Khuôn mẫu hàm
- 6.4 Khuôn mẫu lớp

Cơ bản về vector



- ❑ Dùng để lưu trữ tập dữ liệu CÙNG KIỂU, giống mảng, Nhưng vector có thể thay đổi kích thước trong lúc chạy chương trình (không giống như mảng có kích thước cố định)
- ❑ Thư viện: #include <vector>
- ❑ Ví dụ khai báo
 - *vector<int> A;*
 - vector<int> B(10);*
 - vector<int> C(10,2);*

Một số hàm thành viên của vector



<http://www.cplusplus.com/reference/vector>

Phương thức	Mục đích
v.assign(n,e)	Gán tập giá trị mới cho vector, thay thế nội dung hiện tại của nó đồng thời thay đổi kích thước
v[i] hoặc v.at[i]	Tham chiếu đến phần tử thứ i của vector
v.clear()	Xóa toàn bộ vector
v.pop_back()	Xóa phần tử cuối cùng của vector
v.erase(position)	Xóa phần tử ở vị trí position
v.erase(first, last)	Xóa các phần tử trong khoảng từ first tới last
v.push_back(e)	Thêm phần tử e vào cuối của vector
v.resize(new_size)	Thay đổi kích thước của vector



Nội dung

- 6.1 Nhắc lại về vector
- **6.2 Làm việc với c-string và string**
- 6.3 Khuôn mẫu hàm
- 6.4 Khuôn mẫu lớp



C-string và lớp string

- C-Strings: một kiểu mảng cho chuỗi ký tự
- Các công cụ thao tác ký tự (char)
- Character I/O, cin/getline
- Hàm thành viên: *get, put*
- Một số hàm khác: *pushback, peek, ignore ...*
- Lớp String chuẩn
- Xử lý chuỗi ký tự với lớp String

Hai cách biểu diễn chuỗi (string)



➤ C-strings

- Một mảng với các phần tử có kiểu cơ sở char
- Chuỗi được kết thúc với kí tự null, “\0”
- Là phương thức cũ được kế thừa từ C

➤ Lớp String

- Sử dụng khuôn mẫu (template)



C-strings

- Một mảng các phần tử với kiểu cơ sở char
- Mỗi phần tử của mảng là một ký tự
- Ký tự mở rộng “\0” (được gọi là ký tự rỗng)
- Là dấu hiệu kết thúc một chuỗi ký tự
- Chúng ta đã sử dụng C-strings!
- Ví dụ: literal “Hello” được lưu trữ như một C-string



Biến c-strings

- ❑ Khai báo: char s[10];
- ❑ Khai báo một biến c-string để lưu trữ 9 ký tự
- ❑ Và ký tự cuối cùng là ký tự kết thúc(null- ‘\0’)
- ❑ Chỉ có một điểm khác với mảng chuẩn:
 - C-strings phải chứa ký tự null !
- ❑ Khởi tạo một c-strings: char st[10] = “Hi Mom!”;
st[0] st[1] st[2] st3] st[4] st[5] st[6] st[7] st[8] st[9]..

H	i		M	o	m	!	\o	?	?
---	---	--	---	---	---	---	----	---	---
- ❑ Không cần thiết phải điền đầy đủ (kích thước) mảng
- ❑ Đặt ký tự “\0” ở cuối
- ❑ Có thể bỏ qua kích thước mảng



Thao tác với c-string

- ❑ Một c-string LÀ một mảng => có thể truy cập thành viên thông qua chỉ số (index)
Ví dụ: char ourString[5] = "Hi";
ourString[0] là "H"
ourString[1] là "i"
ourString[2] là "\0"
ourString[3] là không xác định (unknown)
ourString[4] là không xác định (unknown)
- ❑ Chú ý: nếu thực hiện phép gán ourString[2] = 'b';
Ghi đè ký tự "\0" (null) bởi ký tự "b"
- ❑ Nếu ký tự null bị ghi đè, c-string không còn hoạt động như c- string nữa! => kết quả không dự đoán được

Nhập dữ liệu



```
char mystring[10]= “Hello!”;
```

```
char newstring [20];
```

➤ cin

```
std::cin>>newstring;
```

➤ getline

```
std::cin.getline(newstring,12);
```



Toán tử gán

- ❑ C-strings không giống những biến khác
- ❑ Không thể sử dụng phép gán hoặc so sánh
- ❑ Chỉ có thể sử dụng toán tử “=” lúc khởi tạo một c-string!
char aString[10]; aString = “Hello”; // KHÔNG HỢP LỆ
- ❑ Phải sử dụng hàm thư viện cho phép gán:
 - strcpy(aString, "Hello");
- ❑ Một hàm được xây dựng sẵn trong <cstring>
- ❑ Đặt giá trị của aString bằng với “Hello”
- ❑ KHÔNG kiểm tra kích thước!



Toán tử so sánh

- Không thể sử dụng toán tử ‘==’ với c-string

```
char aString[ 10] = “Hello”;
```

```
char anotherString[ 10] = “Goodbye”;
```

```
aString == anotherString; //Không hợp lệ
```

- Phải sử dụng thư viện hàm:

```
if (strcmp(aString, anotherString))
```

```
    cout <<“Strings are NOT same.”;
```

```
else cout << "Strings are same.";
```



Một số hàm trong cstring

FUNCTION	DESCRIPTION	CAUTIONS
<code>strcpy(Target_String_Var, Src_String)</code>	Copies the C-string value <i>Src_String</i> into the C-string variable <i>Target_String_Var</i> .	Does not check to make sure <i>Target_String_Var</i> is large enough to hold the value <i>Src_String</i> .
<code>strcpy(Target_String_Var, Src_String, Limit)</code>	The same as the two-argument <code>strcpy</code> except that at most <i>Limit</i> characters are copied.	If <i>Limit</i> is chosen carefully, this is safer than the two-argument version of <code>strcpy</code> . Not implemented in all versions of C++.
<code>strcat(Target_String_Var, Src_String)</code>	Concatenates the C-string value <i>Src_String</i> onto the end of the C-string in the C-string variable <i>Target_String_Var</i> .	Does not check to see that <i>Target_String_Var</i> is large enough to hold the result of the concatenation.



Một số hàm trong cstring

FUNCTION	DESCRIPTION	CAUTIONS
<code>strcat(Target_String_Var, Src_String, Limit)</code>	The same as the two argument <code>strcat</code> except that at most <i>Limit</i> characters are appended.	If <i>Limit</i> is chosen carefully, this is safer than the two-argument version of <code>strcat</code> . Not implemented in all versions of C++.
<code>strlen(Src_String)</code>	Returns an integer equal to the length of <i>Src_String</i> . (The null character, '\0', is not counted in the length.)	
<code>strcmp(String_1, String_2)</code>	Returns 0 if <i>String_1</i> and <i>String_2</i> are the same. Returns a value < 0 if <i>String_1</i> is less than <i>String_2</i> . Returns a value > 0 if <i>String_1</i> is greater than <i>String_2</i> (that is, returns a nonzero value if <i>String_1</i> and <i>String_2</i> are different). The order is lexicographic.	If <i>String_1</i> equals <i>String_2</i> , this function returns 0, which converts to false. Note that this is the reverse of what you might expect it to return when the strings are equal.
<code>strcmp(String_1, String_2, Limit)</code>	The same as the two-argument <code>strcat</code> except that at most <i>Limit</i> characters are compared.	If <i>Limit</i> is chosen carefully, this is safer than the two-argument version of <code>strcmp</code> . Not implemented in all versions of C++.



Một số hàm trong cstring

- Hàm **get()**: Đọc một ký tự một lần
- Hàm **put()**: Hiển thị một ký tự một lần
- Hàm **putback()**: Giảm vị trí hiện tại trong stream lùi về một ký tự
- Hàm **peek()**: Trả về ký tự tiếp theo, nhưng không loại bỏ nó khỏi luồng input
- Hàm **ignore()**: Bỏ qua input, cho đến khi gặp ký tự được chỉ định

Ví dụ:

```
cin.ignore(1000, "\n"); bỏ qua nhiều nhất 1000 ký tự cho đến khi  
gặp “\n”
```



Lớp string

- Được định nghĩa trong thư viện <string>

```
#include <string>
using namespace std;
```

- Biến string và các biểu thức được xử lý giống như những kiểu đơn giản khác
- Có thể gán, so sánh, cộng string s1, s2, s3;
 - $s3 = s1 + s2;$
 - $s3 = "Hello Mom!"$
- Lưu ý: c-string “Hello Mom!” được tự động chuyển thành kiểu string!

Nội dung



- 6.1 Nhắc lại về vector
- 6.2 Làm việc với c-string và string
- 6.3 Khuôn mẫu hàm
- 6.4 Khuôn mẫu lớp



Giới thiệu

- Hàm dưới đây in ra một mảng các phần tử số nguyên:

```
void printArray(int* array, int size)
{
    for ( int i = 0; i < size; i++ )
        cout << array[ i ] << ", ";
}
```

Giới thiệu



- ❑ Nếu chúng ta muốn in một mảng ký tự thì sao?

```
void printArray(char* array,int size)
{
    for ( int i = 0; i < size; i++ )
        cout << array[ i ] << ", ";
}
```



Giới thiệu

❑ Nếu chúng ta muốn in một mảng các phần tử thực thì sao?

```
void printArray(double* array,  
                int size)  
{  
    for ( int i = 0; i < size; i++ )  
        cout << array[ i ] << ", ";  
}
```

Giới thiệu



□ Bây giờ nếu chúng ta muốn thay đổi cách hàm in mảng. ví dụ:

1 , 2 , 3 , 4 , 5

to

1 - 2 - 3 - 4 - 5



Giới thiệu

❑ Bây giờ chúng ta xem xét lớp **Array** là lớp bao một mảng các số nguyên.

```
class Array {  
    int* pArray;  
    int size;  
public:  
    ...  
};
```



Giới thiệu

- ❑ Điều gì sẽ xảy ra nếu chúng ta muốn sử dụng một lớp **Array** bao bọc mảng các số thực?

```
class Array {  
    double* pArray;  
    int size;  
public:  
    ...  
};
```



Giới thiệu

- ❑ Điều gì sẽ xảy ra nếu chúng ta muốn sử dụng một lớp **Array** bao bọc mảng các bool?

```
class Array {  
    bool* pArray;  
    int size;  
public:  
    ...  
};
```

Giới thiệu



- ❑ Bây giờ nếu chúng ta muốn thêm một phương thức **Sum** vào lớp **Array**, chúng ta phải thay đổi cả ba lớp.



Generic Programming (GP)

- ❑ GP đề cập đến các chương trình chứa các trừu tượng tổng quát
- ❑ Một lập trình trừu tượng tổng quát (function, method, class) có thể được tham số hóa với một kiểu
- ❑ Sự trừu tượng như vậy có thể hoạt động với nhiều loại dữ liệu khác nhau

Ưu điểm



- ❑ Khả năng tái sử dụng
- ❑ Khả năng viết
- ❑ Khả năng bảo trì



Templates

- ❑ Trong C++, lập trình tổng quát được thực hiện bằng cách sử dụng khuôn mẫu (templates)
- ❑ Có 02 dạng:
 - Function Templates
 - Class Templates
- ❑ Trình biên dịch tạo ra các bản sao theo loại cụ thể khác nhau từ một khuôn mẫu duy nhất

Function Templates



- ❑ Một function template có thể được tham số hóa để hoạt động trên các loại dữ liệu khác nhau



Khai báo

```
template< class T >  
void funName( T x );  
// OR
```

```
template< typename T >  
void funName( T x );  
// OR
```

```
template< class T, class U, ... >  
void funName( T x, U y, ... );
```



Example – Function Templates

- ❑ khuôn mẫu hàm sau in ra một mảng có hầu hết mọi loại phần tử :

```
template< typename T >
void printArray( T* array, int size )
{
    for ( int i = 0; i < size; i++ )
        cout << array[ i ] << ", ";
}
```



...Example – Function Templates

```
int main() {  
    int iArray[5] = { 1, 2, 3, 4, 5 };  
    printArray( iArray, 5 );  
    // Instantiated for int[]  
  
    char cArray[3] = { 'a', 'b', 'c' };  
    printArray( cArray, 3 );  
    // Instantiated for char[]  
    return 0;  
}
```

Kiểu tham số tường minh



- ❑ function template có thể không có bất kỳ tham số nào

```
template <typename T>  
T getInput() {  
    T x;  
    cin >> x;  
    return x;  
}
```

Kiểu tham số tường minh



```
int main() {  
    int x;  
    x = getInput();           // Error!  
  
    double y;  
    y = getInput();           // Error!  
}
```

Kiểu tham số tường minh



```
int main() {  
    int x;  
    x = getInput< int >();  
  
    double y;  
    y = getInput< double >();  
}
```

User Specializations



- ❑ Template có thể không xử lý thành công với tất cả các kiểu
- ❑ Cần được cung cấp cho (các) kiểu cụ thể

Example – User Specializations



```
template< typename T >
bool isEqual( T x, T y ) {
    return ( x == y );
}
```

... Example – User Specializations



```
int main {  
    isEqual( 5, 6 ); // OK  
    isEqual( 7.5, 7.5 ); // OK  
    char s1[]="abc";char s2[]="abc" ;  
    cout<<isEqual( s1, s2 )<<endl;  
    // Logical Error!  
    return 0;  
}
```

... Example – User Specializations



```
template< >
bool isEqual< char*>(
    char* x, char* y ) {
    return ( strcmp( x, y ) == 0 );
}
```



... Example – User Specializations

```
int main {  
    isEqual( 5, 6 );  
        // Target: general template  
    isEqual( 7.5, 7.5 );  
        // Target: general template  
  
    isEqual( "abc", "xyz" );  
        // Target: user specialization  
    return 0;  
}
```

Đa kiểu tham số



```
template< typename T, typename U >
T my_cast( U u ) {
    return (T)u;
}

int main() {
    double d = 10.5674;
    int j = my_cast( d );      //Error
    int i = my_cast< int >( d );
    return 0;
}
```



User-Defined Types

- ❑ Bên cạnh các kiểu nguyên thủy, các kiểu do người dùng xác định cũng có thể được chuyển làm đối số kiểu cho các mẫu
- ❑ Trình biên dịch thực hiện kiểm tra kiểu tĩnh để chẩn đoán lỗi kiểu



...User-Defined Types

- Xem xét lớp String không có nạp chòng toán tử “==”

```
class String {  
    char* pStr;  
  
    ...  
    // Operator "==" not defined  
};
```



... User-Defined Types

```
template< typename T >
bool isEqual( T x, T y ) {
    return ( x == y );
}

int main() {
    String s1 = "xyz", s2 = "xyz";
    isEqual( s1, s2 ); // Error!
    return 0;
}
```

...User-Defined Types



```
class String {  
    char* pStr;  
  
    ...  
  
    friend bool operator ==(  
        const String&, const String& );  
};
```

... User-Defined Types



```
bool operator ==( const String& x,  
                    const String& y ) {  
    return strcmp(x.pStr, y.pStr) == 0;  
}
```

... User-Defined Types



```
template< typename T >
bool isEqual( T x, T y ) {
    return ( x == y );
}

int main() {
    String s1 = "xyz", s2 = "xyz";
    isEqual( s1, s2 ); // OK
    return 0;
}
```

Overloading vs Templates



- ❑ Khác kiểu dữ liệu, hoạt động tương tự nhau
=> Cân nạp chồng

- ❑ Các kiểu dữ liệu khác nhau, hoạt động giống hệt nhau
=> Cân khuôn mẫu

Overloading vs Templates



- Toán tử ‘+’ là được nạp chồng với kiểu toán hạng khác nhau.
- Một khuôn mẫu hàm duy nhất có thể tính tổng của mảng nhiều kiểu dữ liệu



Overloading vs Templates

```
String operator +( const String& x,
                     const String& y ) {
    String tmp;
    tmp.pStr = new char[strlen(x.pStr) +
                        strlen(y.pStr) + 1];
    strcpy( tmp.pStr, x.pStr );
    strcat( tmp.pStr, y.pStr );
    return tmp;
}
```



Overloading vs Templates

```
String operator +( const char * str1,  
                    const String& y ) {  
  
    String tmp;  
    tmp.pStr = new char[ strlen(str1) +  
                        strlen(y.pStr) + 1 ];  
    strcpy( tmp.pStr, str1 );  
    strcat( tmp.pStr, y.pStr );  
    return tmp;  
}
```

Overloading vs Templates



```
template< class T >
T sum( T* array, int size ) {
    T sum = 0;

    for (int i = 0; i < size; i++)
        sum = sum + array[i];

    return sum;
}
```

Print an Array



```
template< typename T >
void printArray( T* array, int size )
{
    for ( int i = 0; i < size; i++ )
        cout << array[ i ] << ", ";
}
```



...Generic Algorithms

- Thuật toán này hoạt động tốt cho các mảng thuộc bất kỳ loại nào
- Chúng tôi có thể làm cho nó hoạt động cho bất kỳ vùng tống quát chung nào hỗ trợ hai hoạt động
- Toán tử tham khảo (*)
 - Toán tử tăng dần (++)
 - Toán tử con trỏ (*)



...Generic Algorithms

```
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
    while ( start != beyond &&
            *start != x )
        start++;
    return start;
}
```



...Generic Algorithms

```
int main() {  
    int iArray[5];  
    iArray[0] = 15;  
    iArray[1] = 7;  
    iArray[2] = 987;  
    ...  
    int* found;  
    found = find(iArray, iArray + 5, 7);  
    return 0;  
}
```



Nội dung

- 6.1 Nhắc lại về vector
- 6.2 Làm việc với c-string và string
- 6.3 Khuôn mẫu hàm
- 6.4 Khuôn mẫu lớp

Class Templates



- ❑ Một khuôn mẫu lớp đơn cung cấp hàm hoạt động trên các kiểu dữ liệu khác nhau
- ❑ Tận dụng tái sử dụng các lớp.
- ❑ Định nghĩa khuôn mẫu lớp như sau:
 - template< class T > class Xyz { ... }; or
 - template< typename T > class Xyz { ... };

Example – Class Template



- ❑ Lớp **Vector** có thể lưu trữ các phần tử dữ liệu của nhiều kiểu khác nhau
- ❑ Không dùng khuôn mẫu, chúng ta cần phân tách lớp Vector cho mỗi kiểu dữ liệu.

...Example – Class Template

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





...Example – Class Template

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

Bàn đến việc định nghĩa method của class template

- Trong lớp: đ/n như lớp thông thường
- Ngoài lớp: cần nhắc lại cho trình biên dịch C++ biết các tham số kiểu template <class T> và tên class được viết lại point<T>



...Example – Class Template

Trong lớp

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



...Example – Class Template

```
#include <iostream.h>
//tạo khuôn mẫu lớp
template <class T> class point {
    T x, y;
public:
    // Định nghĩa hàm thành phần bên trong khuôn mẫu lớp
    point(Tabs=0, Tord=0) {
        x=abs; y=ord;
    }
    void display();
};

// Định nghĩa hàm thành phần bên ngoài khuôn mẫu lớp
template <class T> void point<T>::display() {
    cout<<"Toa do: "<<x<<" "<<y<<"\n";
}
```



Sử dụng Class Template

- Một khi khuôn mẫu lớp point được định nghĩa, thì khai báo như sau :
point<int> ai;

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

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

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

...Example – Class Template



```
int main() {  
    point<int> ai(3,5);  
    ai.display();  
    point<char> ac('d','y');  
    ac.display();  
    point<double> ad(3.5, 2.3);  
    ad.display();  
    return 0;  
}
```



Các tham số khuôn mẫu lớp

□ Có 2 loại:

- Tham số kiểu
- Tham số biểu thức

□ Có nhiều điểm giống khuôn mẫu hàm nhưng các ràng buộc đối với các kiểu tham số lại khác nhau

Các tham số khuôn mẫu lớp



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



Các tham số khuôn mẫu lớp

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

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

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

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

Các tham số khuôn mẫu lớp



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

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



Các tham số khuôn mẫu lớp

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

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

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



Các tham số khuôn mẫu lớp

- Tham số biểu thức trong khuôn mẫu lớp: tham số thực tế tương ứng với tham số biểu thức phải là một hằng số
- Giả sử chúng ta định nghĩa lớp Arr để thao tác trên mảng chứa đối tượng có kiểu bất kỳ.
 - Tham số 1: tham số kiểu
 - Tham số 2: tham số biểu thức để xác định số lượng phần tử.



Tham số biểu thức

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

- Tham số kiểu được xác định bởi từ khóa class
- Tham số biểu thức kiểu int. Phải chỉ rõ giá trị trong khai báo các lớp thể hiện
 - Arr <int, 4>

```
class Arr {  
    int elements[4];  
public:  
    ...  
};
```



Các tham số khuôn mẫu lớp

- Có thể khai báo tùy ý các tham số biểu thức trong khuôn mẫu hàm.
- Các tham số này có thể xuất hiện ở bất kỳ nơi nào trong định nghĩa khuôn mẫu lớp
- Khi cụ thể 1 lớp có tham số biểu thức, các tham số thực tế tương ứng truyền vào phải là hằng phù hợp với kiểu dữ liệu đã khai báo



Cụ thể hóa khuôn mẫu lớp

- ❑ khuôn mẫu lớp định nghĩa họ các lớp trong đó mỗi lớp chứa đồng thời định nghĩa của chính nó và các hàm thành phần
- ❑ Tất cả các hàm thành phần cùng tên sẽ được thực hiện theo cùng một giải thuật
- ❑ Nếu muốn 1 hàm thành phần thích ứng tình huống cụ thể nào đó có thể viết 1 đ/n khác

Ví dụ



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

Ví dụ



```
void point<char>::display() {
    cout<<"Toa do: "<<(int)x<<" "<<(int)y<<"\n"
}
void main() {
    clrscr();
    point <int> ai(3,5); ai.display();
    point <char> ac('d','y');
    ac.display();
    point <double> ad(3.5, 2.3);
    ad.display();
    getch();
}
```

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



Chú ý

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

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

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



Nhận xét

Nhận xét

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

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

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

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



Nhận xét

- ☐ Xét lệnh gán giữa 02 đối tượng: hai lớp thể hiện tương ứng với cùng 1 kiểu nếu các tham số kiểu tương ứng nhau 1 cách chính xác và các tham số biểu thức có cùng giá trị (trừ kẽ thửa)

Arr <int, 12> a1;

Arr <float, 12> a2;

Arr <int, 12> a1;

Arr <int, 16> a2;

- ☐ Thì không được viết $a2 = a1$
- ☐ Kẽ cả khai báo