

# POLYMORPHISM

# POLYMORPHISM

- Polymorphism in C++ is exhibited by the ability of a pointer or reference to a base class type to behave in different ways when it is used to manipulate objects of different subtypes of that base class.
- 多形就是當你可以用基底class的指標或參考來進行處理時，卻可以依照其 subtype 來進行不一樣的事情（其實多形是很難用解釋的）



# (USING A GRAPHICS APPLICATION)

## 以繪圖應用程式為例處理為例

- Suppose we have a base class called “shape”
- Many shapes box, circle, rectangle.... Inherit shape.
- In this type of application, it is inevitable you encounter such fragment of code to repaint all objects)  
(在寫這一類的應用程式時，我們一定會寫到一個 function，這個 function 是要重繪所有的繪圖物件，假設我們這些物件都放在一個
- Suppose all the objects are put in a `vector<shape *>container`
- ```
void paintall(vector<shape *> & container) {  
    for each shape object x in container  
        x->draw();  
}
```
- In this example, we hope each object can draw its specific shapes.
- 在這個例子我們希望 shape 物件 x 可以依照其原來的類別進行 draw() 的行為
- 我們可不希望前一堂課的按照 pointer 類別呼叫 method 的狀況出現



# VIRTUAL FUNCTIONS

```
class Pet
{
public:
// Constructors, Destructors
Pet () {}
virtual ~Pet() {}

//General methods
virtual void speak();
void breath();

};
```

- 任何想要被繼承的 class overridden 的 method 都要宣告成 virtual
- 一個 function 一旦宣告成 virtual，則在繼承的 class 中也是 virtual。但是可以忽略不寫，不過建議要寫
- Polymorphism only works for pointer and reference with function that declared as virtual (多型只對 pointer, reference, 以及宣告為 virtual 的 function 有效)
- If a class has any method declared as virtual, its destructor should be declared as virtual. (如果一個 class 有任何一個 method 宣告為 virtual 則 destructor 也應該宣告為 virtual，這為了確保正確的 destructor 做正確的清除)



# COMMENTS

- The method, `breath()`, is not declared as virtual. Why? Because I am assuming that all animals breath and that this method will not be overridden in base classes.
- Suppose that later some programmer defines a "fish" subclass. Then, perhaps, `breath` needs to be overridden to allow for underwater breathing. For now, I'll assume it's adequate.
- Why not declare all methods as virtual?
  - This could be done, but involves additional overhead. The compiler creates a virtual function table that is used in the polymorphism mechanism. Avoiding unnecessary details, the more functions that are declared as virtual, the greater the overhead to create and maintain these tables.
  - But, by declaring all methods virtual, redesign may be avoided if the classes need to change.
- Some C++ programmers declare all methods virtual if any need be. Some don't. In some programming languages, such as Java, all methods are virtual by default. I don't have a definitive answer on this one.



```

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

class Pet {
public:
// Constructors, Destructors
Pet () {}
virtual ~Pet() {}

//General methods
virtual void speak();
void breath() { cout << "Gasp" << endl;}

};

void Pet::speak() { cout << "Growl" << endl;}

class Rat: public Pet{
public:
Rat() {}
~Rat() {}

void speak();
};

void Rat::speak() { out << "Rat noise" << endl;}

class Cat: public Pet {
public:
Cat() {}
~Cat() {}
void speak();
};

```

```

void Cat::speak() {cout << "Meow" << endl;}

void chorus(Pet pet, Pet *petPtr, Pet
&petRef)
{
pet.speak();
petPtr->speak();
petRef.speak();
}

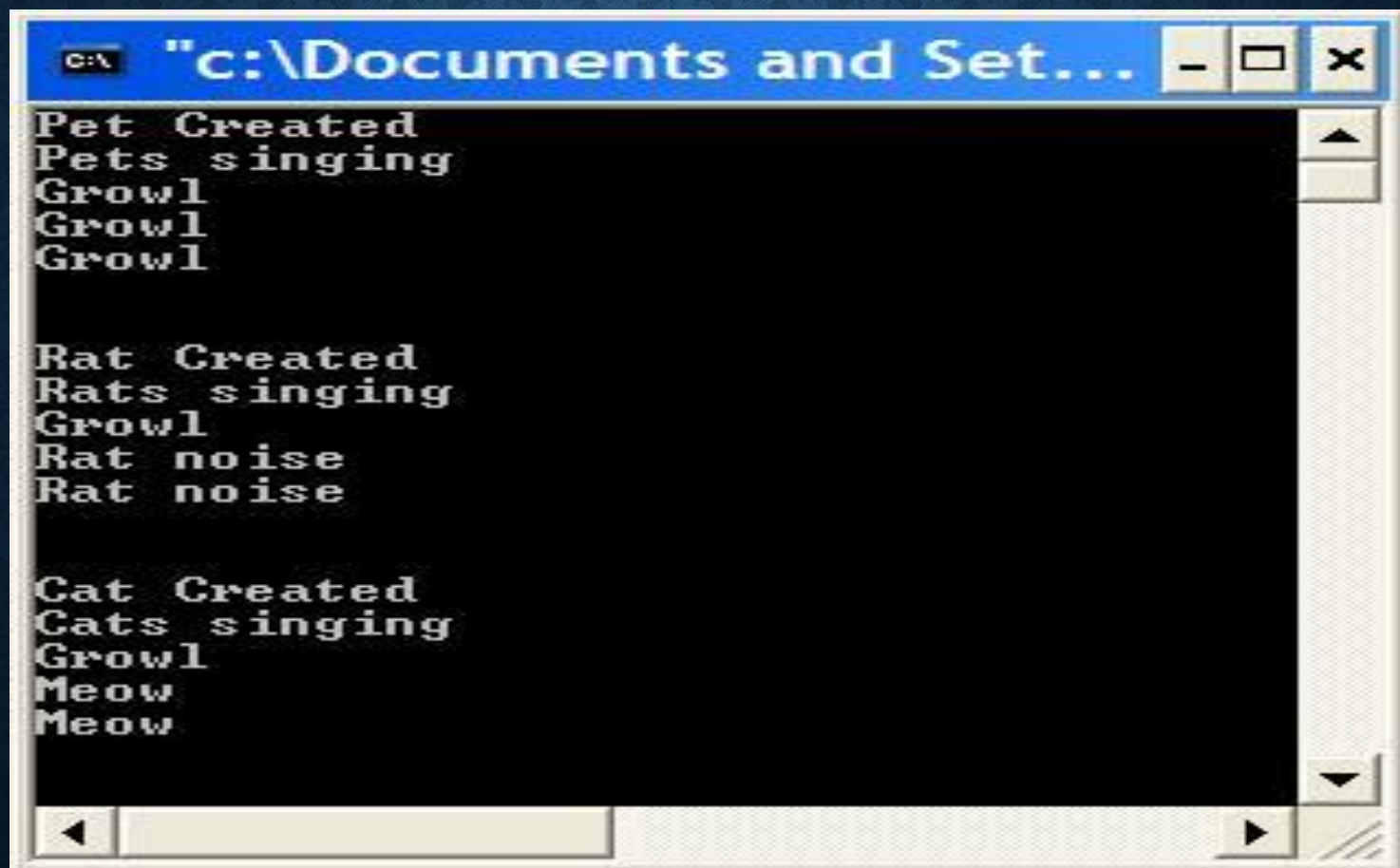
int main()
{
Pet *ptr; //Pointer to base class

ptr = new Pet;
cout << "Pet Created" << endl;
cout << "Pets singing" << endl;
chorus(*ptr,ptr,*ptr);
cout << endl << endl;
delete ptr; //Prevent memory leaks

ptr = new Rat;
cout << "Rat Created" << endl;
cout << "Rats singing" << endl;
chorus(*ptr,ptr,*ptr);
cout << endl << endl;
delete ptr; //Prevent memory leaks

ptr = new Cat;
cout << "Cat Created" << endl;
cout << "Cats singing" << endl;
chorus(*ptr,ptr,*ptr);
cout << endl << endl;
delete ptr; //Prevent memory leaks
return 0;
}

```





# OBJECT SLICING

- You may recall that when you pass an argument by value, a copy is made.
- In this example, the compiler makes space for a Pet object and then copies the Rat object argument into it. Since the Rat object is larger, it doesn't fit. This is called *slicing*. Only the Pet part of Rat gets copied. When speak is called, it is Pet's speak method that is executed.



# POLYMORPHISM

- When passed a pointer of type Pet holding the address of a Rat object, we get Rat behavior - "rat noise" is printed. This is polymorphism.
- Polymorphism is seen when a pointer or reference of a base class points to a subclass object.
- Methods must be declared as virtual for polymorphism to work.

# AN ARRAY OF POINTERS

```
#define MAXCAPACITYINROOM 50
int main()
{
    int quit = 0;
    int choice;
    int totalNumber;
    Pet
    *house[MAXCAPACITYINROOM];
    //Array of pointers to base class

    int i = 0;
    while (!quit && i <
MAXCAPACITYINROOM)
    {
        cout << "Enter (0) to quit" << endl;
        cout << "Enter (1) for Rat" << endl;
        cout << "Enter (2) for Cat" << endl;
        cin >> choice;
```

```
        if (choice == 0) {
            quit = 1;
        }
        else if (choice == 1) {
            house[i++] = new Rat();
        } else if (choice == 2) {
            house[i++] = new Cat();
        } else {
            cout << "Invalid Choice, Reenter"
            << endl;
        }
    }
    totalNumber = i;

    for (i = 0; i < totalNumber; i++)
    {
        house[i]->speack();
    }

    return 0;
}
```



# COMMENTS

- Why have I chosen an array of pointers of the base class type, "Pet", to store my pets as I enter them into the program. Because I do not know at compile time the type of each subclass object (Rat or Cat) and need to be able to store either.
- I need each to behave as the subclass it is. I want polymorphism. That is, I want each object to have the appropriate behavior according to its subclass.
- An array of objects would not work. Remember, polymorphism is supported through pointers and references.

# THE PRINCIPLE OF POLYMORPHISM



# OOP (THE FOUR BASIC ELEMENTS)



物件與 message



Dynamic binding  
(via polymorphism)



inheritance



encapsulation

# POLYMORPHISM

- 在物件導向的語言中，polymorphism (多型) 是 inheritance, message passing 所自然結合產生的結果
- What is polymorphism?

函式在執行的時候，面對不同的型別或物件，能有自動相對應的操作及功能

與overloading 多出來的功能，是在於可以真正的以一段同樣的程式碼對不同的 資料型別或物件進行操作



# MAKING POLYMORPHISM WITHOUT OOP

```
Color NewColor;                // 目前設定的繪圖顏色
void main() {
    int Cont = 1;               // 迴圈是否繼續執行的指示
    int Event = 0;              // 使用者的動作會變成 "event", 0 表示沒有動作
    if(Initial()) {             // 進入繪圖模式和其他的設定
        while(Cont) {          // 主要的迴圈
            Event = GetEvent();  // 取得使用者的動作, 如按下一個功能鍵....
            switch(Event) {      // 判別使用者的動作是什麼
                case Circle:
                case Pie:
                case Ellipse:
                case EllipsePie:
                    gCircle(Event, NowColor);
                                // 若是有關圓形的功能, 就進入圓形的操作控制當中
                    break;

                case Rect:
                case Box:
                    gRect(Event, NowColor);
                                // 如果是有關方形的功能, 就進入方形的操作控制當中
                    break;

                .....

                case Exit:
                    Cont = !gExit();
                                // 如果是按下離開程式的按鍵, 則要確定使用者是否想要離開
                    break;
            }
        }
    }
    End();                      // 做離開程式的一些工作, 如回到文字模式, 釋放所有
                                // 使用的記憶體.....等等
}
```

# SWITCH PATTERNS OFTEN APPEAR IN SEVERAL PLACES

```
Switch(xxx) {  
  Case x: ...  
  Case y:...  
  Case z: ....
```

```
Switch(xxx) {  
  Case x: ...  
  Case y:...  
  Case z: ....
```

```
Switch(xxx) {  
  Case x: ...  
  Case y:...  
  Case z: ....
```

```
Switch(xxx) {  
  Case x: ...  
  Case y:...  
  Case z: ....
```



# FUNCTION POINTER

- `int (*f)(int, int);`
- `int (*f[])(int,int);`

```

Color NewColor; // 目前的繪圖顏色
int Cont = 1; // 控制迴圈的變數

void main() {
    int E = 0; // 存放使用者動作的變數
    struct { // 由函式指標組成的結構
        int (*Draw)(void (*())(), void (*())()); // 操作控制的函式
        void (*Move)(); // 移動圖形時所用的函式
        void (*Act)(); // 繪製圖形時所用的函式
    } F[] = {
        ✓ {gRegion, XorCircle, PutCircle}, // 操作控制是用滑鼠設定區域、移動、改變
  // 區域大小的函式；移動圖形是用 XOR 方式
  // 繪製的圓形；繪製的圖形是圓
        ✓ {gRegion, XorCircle, PutPie}, // 填滿的圓形
        ✓ {gRegion, XorEllipse, PutEllipse}, // 橢圓形
        {gRegion, XorEllipse, PutEllipsePie}, // 填滿的橢圓形
        {gRegion, XorRect, PutRect}, // 方形
        {gRegion, XorRect, PutBar}, // 填滿的方形
        {gExit, NULL, NULL} // 離開，會設定 Cont
    };

    if(Initial()) { // 初始的設定
        while(Cont) { // 主迴圈
            E = GetEvent(); // 取得使用者的動作
            (*F[E].Draw)(F[E].Move, F[E].Act); // 根據動作編號動作
        }
        End(); // 程式結束的處理
    }
}

```

this are functions



- Inheritance alone cannot implement polymorphism.
- We need a new mechanism called dynamic binding to make polymorphism to work.
- 我們需要一種新的機制，使我們能夠用同樣的操作方式，完成不同實質內容的動作
- 也就是動態連結（dynamic binding）

# RECALL THE PREVIOUS EXAMPLE: MAKING POLYMORPHISM WITH OOP

```
class Circle : Public Shape{  
    public:  
        virtual void Draw() { .....}  
        virtual void Move() { ....}  
        virtual void Act() { ....}  
};
```

```
void main () {  
    Vector <Shape *> ShapePool ;  
    ShapePool.insert(new Circle);  
    ShapePool.insert(new Pie);  
    .....  
    if (initial()) {  
        while (cont) {  
            E = GetEvent();  
            ShapePool[E]->Draw();  
        }  
    }  
}
```

How can this be achieved?



# HOW POLYMORPHISM IS IMPLEMENTED?

- Lets' see the binding of compiler and linker

## c 原始程式

```

void Do_something()
{
    /* ... */
}
void Sit_on_it()
{
    /* ... */
}
void Think_it()
{
    /* ... */
}
main()
{
    float x, y, z;
    Sit_on_it();
    Think_it();
    Do_something();
    x = y + z;
}

```

## 編譯後的組合語言碼

```

START main
Do_something: ←
.
.
RET
Sit_on_it: ←
.
.
RET
Think_it: ←
.
.
RET
main:
CALL Sit_on_it
CALL Think_it
CALL Do_something
PUSH y
PUSH z
CALL Float_plus
MOVE R0, x
END

```

logic addr

執行時段  
函式庫

```

.
.
.
Float_plus:
.
.
RET
...

```

linking

圖 9-1 : C 的程式檔內靜態繫結



## 編譯動作

## 連結動作

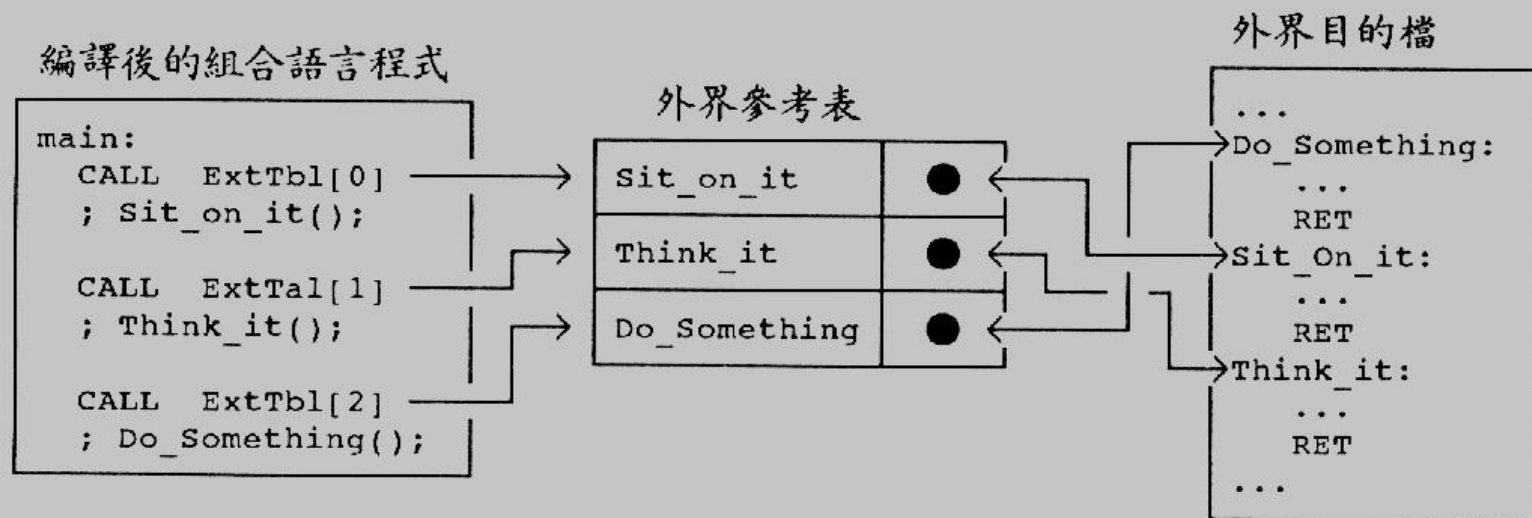


圖 9-2 : C 的外界參考靜態繫結 (連結)

# DYNAMIC BINDING

子：

```
class Shape {  
    public:  
        double x0, y0;  
        Shape(double x, double y);    // 建構元  
        virtual double area();        // 計算表面積  
        virtual void draw();          // 繪製圖形  
};
```

Shape 類別中，`virtual` 關鍵字生成了 `area()` 與

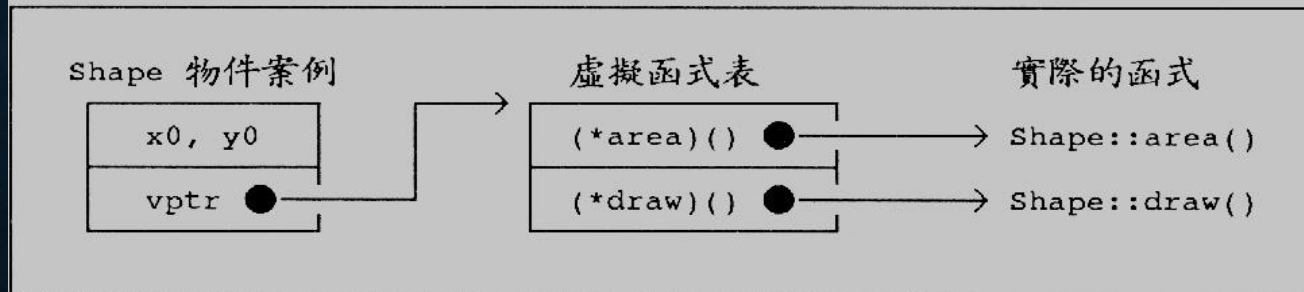


圖 9-5：基底類別物件與虛擬函式表



```

class Circle : public Shape {
public:
    double radius;
    Circle(double x, double y, double r); // 建構元
    void draw();                          // 自己的畫圓函式
   // 取代 Shape::draw()
};

```

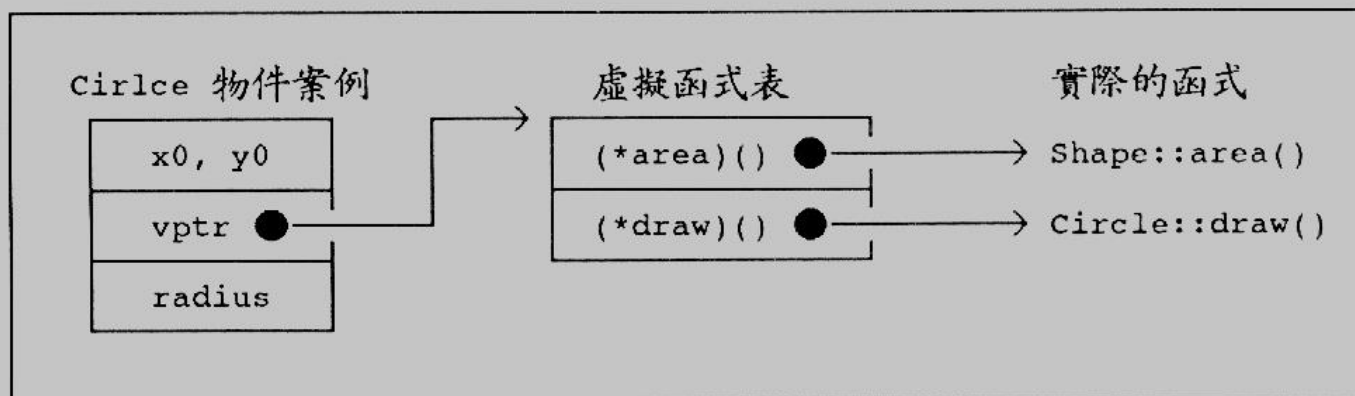


圖 9-6 : 衍生類別物件與虛擬函式表

# COMPILER TRANSLATION

Shape \*p ;

Shape A ;

Circle B ;

p = &A ;

p->draw();

**(p->vptr[1])();**

p = &B ;

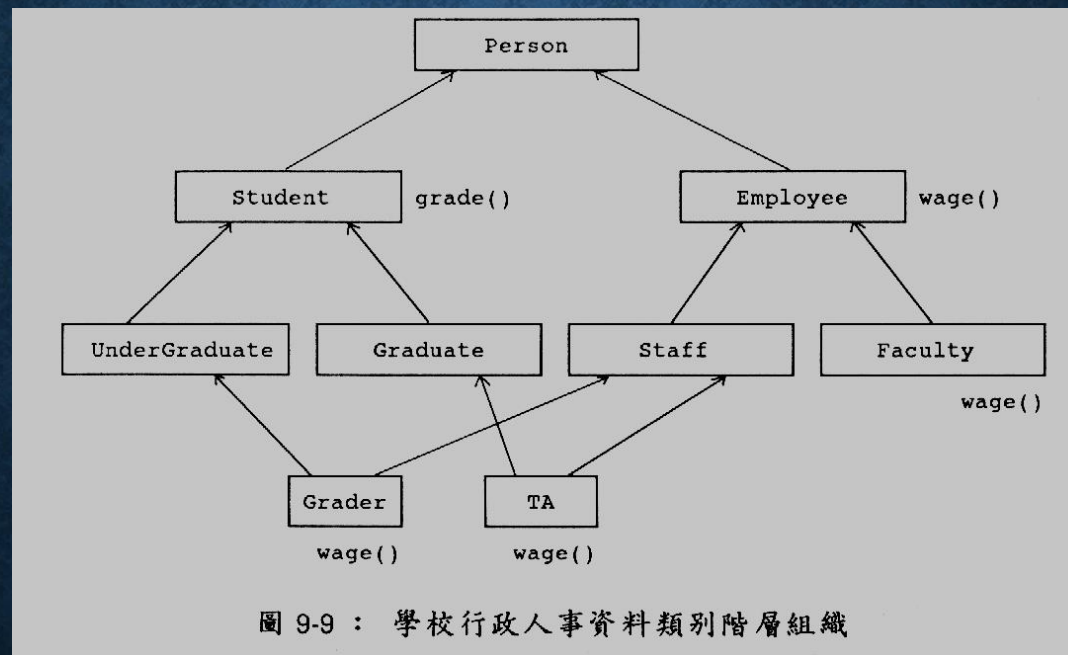
p->draw()

**(p->vptr[1])();**



# OOA

- Let's go back to OOA again
- In a school wage system you want to analyze it and obtain an inheritance hierarchy as follows



# FACT !

- Actually, writing OO programs is a process, not just programming
- Most programmers learn OOP (C++ ,Java) first, without knowing that writing OO programs require a planning and design first.
- It is a process not a programming task
- Its goal : to make the system highly reusable and maintainable



# WHY OO PROGRAMS ARE EASILY MAINTAINABLE IF IT IS DONE PERFECTLY?

```
class Circle : Public Shape{
    public:
        virtual void Draw() { ....}
        virtual void Move() { ....}
        virtual void Act() { ....}
};

void main () {
    Vector <Shape *> ShapePool ;
    ShapePool.insert(new Circle);
    ShapePool.insert(new Pie);
    .....
    if (initial()) {
        while (cont) {
            E = GetEvent();
            ShapePool[E]->Draw();
        }
    }
}
```

Take the following examples again. When you want to extend the system to another shape called Polyghon You simply write a class to extend Shap and add a new line in main () or some function called init() and then you do not need to change other parts of the program

# VIRTUAL BASE CLASS (ABSTRACT CLASS)

- In C++, class that has member functions are pure virtual function, such as
  - `virtual void p() = 0 ;`
- 這些 class 稱為 abstract class, 沒有 instance
  - 也就是你在程式中不會從 abstract class new 出 instances

```
class Human {  
    public:  
        virtual Money work() = 0 ;  
};
```



# THUMB RULE

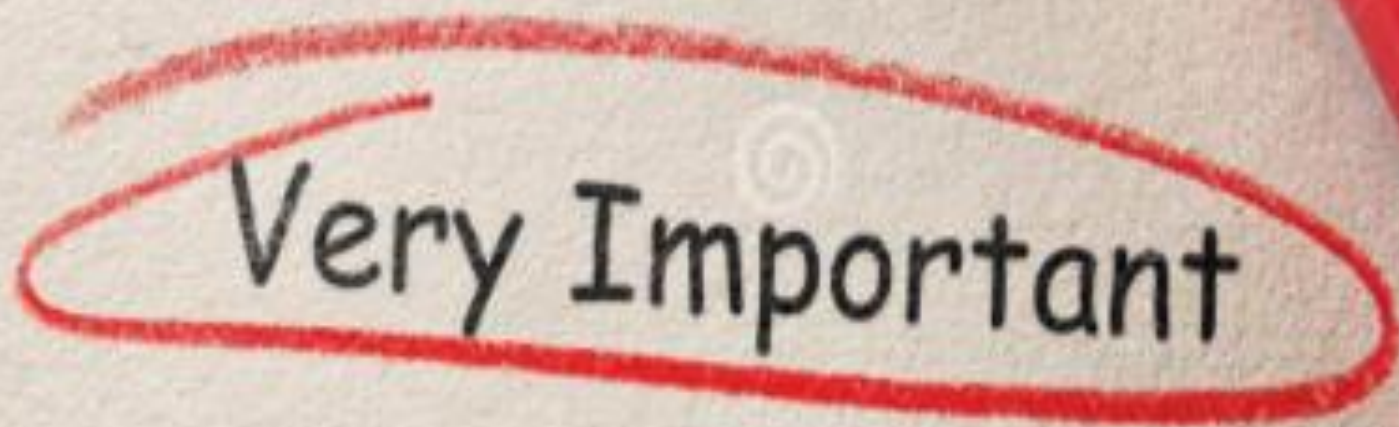
- If a member function can be overridden by subclass, declare it as virtual
- 假設某個member function 以後有可能會被derived class修改，就在他的前面加 virtual 就對了。
- 假如你喜歡用多重繼承，那麼只要這個class可能有兩個以上的derived class 則你應該在繼承他的時候在前面加上 virtual

# QUESTION

- Why not making an abstract class to have all the methods to be abstract and enforce its subclasses to implement each methods? Such as

```
class Human {  
    public:  
        virtual void walk() = 0 ;  
        virtual void breath() = 0 ;  
        virtual void speak()=0;  
};
```





Very Important

# ONE CRITICAL POINT TO DETERMINE IF YOU OO PROGRAM IS REALLY DOING RIGHT

```
class base {
```

```
    int a ;
```

```
    int b ;
```

```
}
```

```
class something : base {
```

```
    int c ;
```

```
}
```

Somewhere in the code

....

...

...

something s = new something();

....

s.