純虛擬函式、抽象類別（Abstract class）

節錄自

http://openhome.cc/Gossip/CppGossip/PureVirtualFunction.html

以下C++檔案，請參考 ch15.rar裡的子目錄t15\_11(Circle)

C++預設函式成員都不是虛擬函式，如果要將某個函式成員宣告為虛擬函式，則要加上"virtual"關鍵字，然而C++提供一種語法定義「純虛擬函式」 （Pure virtual function），指明某個函式只是提供一個介面，要求繼承的子類別必須重新定義該函式，定義純虛擬函式除了使用關鍵字"virtual"之外，要在函 式定義之後緊跟著'='並加上一個0，例如：

**class Some {**
**public:
    // 純虛擬函式
    virtual void someFunction() = 0;**
**....**
**};**

一個類別中如果含有純虛擬函式，則該類別為一「抽象類別」（Abstract class），該類別只能被繼承，而不能用來直接生成實例，如果試圖使用一個抽象類別來生成實例，則會發生編譯錯誤。

以下舉個實際的例子，先假設您設計了兩個類別：ConcreteCircle與HollowCircle：

**class ConcreteCircle {
public:
    void radius(double radius) {
         \_radius = radius;
    }
    double radius() {
        return \_radius;
    }
    void render() {
         cout << "畫一個半徑 "
              << \_radius
              << " 的實心圓"
              << endl;
    }
private:
    double \_radius;
};**

**class HollowCircle {**
**public:**
**void radius(double radius) {**
**\_radius = radius;**
**}**
**double radius() {**
**return \_radius;**
**}**
**void render() {**
**cout << "畫一個半徑 "**
**<< \_radius**
**<< " 的空心圓"**
**<< endl;**
**}**
**private:**
**double \_radius;**
**};**

顯然的，這兩個類別除了render()方法的實作內容不同之外，其它的定義是一樣的，而且這兩個類別所定義的顯然都是「圓」的一種類型，您可以定義一個 抽象的AbstractCircle類別，將ConcreteCircle與HollowCircle中相同的行為與定義提取（Pull up）至抽象類別中：

* AbstractCircle.h

#ifndef ABSTRACTCIRCLE
#define ABSTRACTCIRCLE

**class** AbstractCircle {
**public**:
 **void** radius(**double** radius) {
 \_radius = radius;
 }
 **double** radius() {
 **return** \_radius;
 }
 // 宣告虛擬函式
 **virtual** **void** render() = 0;

**protected**:
 **double** \_radius;
};

#endif

注意到在類別宣告了虛擬函式render()，所以AbstractCircle是個抽象類別，它只能被繼承，繼承了AbstractCircle的類別 必須實作render()函式，接著您可以讓ConcreteCircle與HollowCircle類別繼承AbstractCircle方法並實作 render()函式：

* HollowCircle.h

#include <iostream>
#include "AbstractCircle.h"
**using** **namespace** std;

**class** HollowCircle : **public** AbstractCircle {
**public**:
 **void** render() {
 cout << "畫一個半徑 "
 << \_radius
 << " 的空心圓"
 << endl;
 }
};

* ConcreteCircle.h

#include <iostream>
#include "AbstractCircle.h"
**using** **namespace** std;

**class** ConcreteCircle : **public** AbstractCircle {
**public**:
 **void** render() {
 cout << "畫一個半徑 "
 << \_radius
 << " 的實心圓"
 << endl;
 }
};

由於共同的定義被提取至AbstractCircle類別中，並於衍生類別中繼承了下來，所以在ConcreteCircle與HollowCircle 中無需重覆定義，只要定義個別對render()的處理方式就行了，而由於ConcreteCircle與HollowCircle都是 AbstractCircle的子類別，因而可以使用AbstractCircle上所定義的虛擬操作介面，來操作子類別實例上的方法，如下所示：

* main.cpp

#include <iostream>
#include "AbstractCircle.h"
#include "ConcreteCircle.h"
#include "HollowCircle.h"
**using** **namespace** std;

**void** render(AbstractCircle &circle) {
 circle.render();
}

**int** main() {
 ConcreteCircle concrete;
 concrete.radius(10.0);
 render(concrete);

 HollowCircle hollow;
 hollow.radius(20.0);
 render(hollow);

 **return** 0;
}

執行結果：

畫一個半徑 10 的實心圓
畫一個半徑 20 的空心圓

什麼是**多態性 (polymorphism)**呢?
多態性就是當你對一個擁有特殊性質的物件進行函式呼叫, 出來的結果可能來自兩個不同的類別.

<http://www.study-area.org/coobila/tutorial_430.html>

請參考以下範例 : t15\_12.cpp

#include <iostream>

using namespace std;

class rectangle

{

 public:

 virtual void output()=0; //虛擬函式, 等一下會介紹

 void output2();

};

class square : public rectangle

{

 public:

 void output(); //square中的兩個函式由於和rectangle名稱相同, 因此會取代繼承來的函式

 void output2();

};

void rectangle::output()

{

 cout << "這是一個長方形" << endl;

}

void rectangle::output2()

{

 cout << "這是一個長方形, 非正方形" << endl;

}

void square::output()

{

 cout << "這是一個正方形" << endl;

}

void square::output2()

{

 cout << "這是一個正方形, 也算是長方形" << endl;

}

int main()

{

 rectangle \*one; //這是一個rectangle的指標物件

 square two; //這是一個square物件

 one = &two; //從這邊就比較奇怪一點,

 //把rectangle指標指向一個square物件的記憶體位置?

 one -> output(); //虛擬函式被square物件給取代了, 呼叫square::output(), 展現多態性

 one -> output2(); //呼叫rectangle::output2(), 非多態

 two.output(); //剩下這兩個都是呼叫square下的函式

 two.output2();

}

one本身是一個rectangle的指標, 但是由於output()函式本身是**虛擬函式 (virtual function)**, 因此當我們將一個sqaure的物件位置指定給一個rectangle的指標 (這個物件就擁有這兩個類別的部分特性了, 夠特殊吧), 就會被取代.

當從one指標執行output()的時候, 他會執行實際物件內容的output(), 也就是square::output().

可以運用的到呢?
假如說你今天設計了一個類別, 然後程式已經正常運行了. 但是過了一陣子你又寫了新的類別去繼承原有的類別 - 多態性讓你可以用一種指標或是參考資料型態來使用不同類別的物件 (但多半都是繼承同樣的父類別).