C++程式數計：繼承與多型

**目錄**

[C++程式數計：繼承與多型 1](#_Toc205821948)

[1. 繼承（Inheritance） 1](#_Toc205821949)

[2. 三種繼承方式與成員存取關係 2](#_Toc205821950)

[2.1 基本繼承語法 2](#_Toc205821951)

[2.2 範例：public 繼承的基本範例 2](#_Toc205821952)

[2.3 範例：protected 繼承的基本範例 3](#_Toc205821953)

[2.4 範例：private 繼承的基本範例 4](#_Toc205821954)

[3. 多重繼承（Multiple Inheritance） 6](#_Toc205821955)

[3.1 衍生類別得建構子與解構子 7](#_Toc205821956)

[3.2 範例：單一繼承下的建構子與解構子 8](#_Toc205821957)

[3.3 範例：多重繼承下的建構子與解構子 8](#_Toc205821958)

[4. 多型（Polymorphism） 9](#_Toc205821959)

[4.1 範例：靜態綁定(就沒有做報多形) 9](#_Toc205821960)

[5. 虛擬函數(virtual function) 10](#_Toc205821961)

[5.1 純虛擬函數(virtual function) 11](#_Toc205821962)

[5.2 範例：使用 virtual 完成多型 11](#_Toc205821963)

[6. 基礎抽象類別 12](#_Toc205821964)

[7. 虛擬基礎類別(virtula base class) 13](#_Toc205821965)

[7.1 範例：虛擬基礎類別(virtula base class) 13](#_Toc205821966)

# 繼承（Inheritance）

繼承是物件導向程式設計中非常重要的觀念。我們可以根據既有的類別建立新類別，新類別稱為 衍生類別（Derived Class），舊的稱為 基礎類別（Base Class）。衍生類別會保留基礎類別的大部分功能，也可以擴充自己的特性。

繼承好處

* 程式碼重複利用（Reuse）
* 易於維護與修改
* 階層式設計與擴充彈性

無法被繼承的：

* 建構子，解構子
* 多載
* 朋友類別

# 三種繼承方式與成員存取關係

* private 成員永遠無法被子類別存取（只能透過 成員函數間接取得）。
  + private 繼承：僅限內部使用，外部無法使用父類別成員。
* public 繼承保留了最大程度的存取權限，是最常用的繼承方式。
* protected 繼承：仍允許子類別繼承並存取，但不能給物件使用。

|  |  |  |  |
| --- | --- | --- | --- |
| **成員屬性 \ 繼承方式** | **public 繼承** | **protected 繼承** | **private 繼承** |
| public 成員 | 保留 public | 變成 protected | 變成 private |
| protected 成員 | 保留 protected | 保留 protected | 變成 private |
| private 成員 | ❌ 無法繼承 | ❌ 無法繼承 | ❌ 無法繼承 |

## 基本繼承語法

// : 是繼承關係宣告符號

class 父類別名稱 **{**

// 基礎資料與功能

**};**

class 子類別名稱 **:** public 父類別名稱 **{**

// 子類別新增資料與功能

**};**

## 範例：public 繼承的基本範例

#include <iostream>

using namespace std;

// 父類別 Car

class car {

public:  // 設為 public，讓子類別 freighter 可繼承並存取

    void go() {

        cout << "汽車啟動了!" << endl;

    }

    void stop() {

        cout << "汽車熄火了!" << endl;

    }

};

// 子類別 freighter 繼承自 car（使用 public 繼承）

class freighter : public car {

    // 不需額外定義成員，繼承 car 的 go() 與 stop()

};

int main() {

    freighter ft;

    ft.stop();  // 使用繼承自 car 的 stop()

    cout << "-----------------------------" << endl;

    ft.go();    // 使用繼承自 car 的 go()

    return 0;

}

## 範例：protected 繼承的基本範例

#include <iostream>

using namespace std;

// 基礎類別 student

class student {

private:

    int age;          // 私有成員（無法直接被子類別使用）

protected:

    int lang;         // 受保護成員（可被子類別使用）

public:

    int math;         // 公有成員（可被子類別與外部物件使用）

    student() {       // 建構子：初始化三個成員變數

        age = 0;

        lang = 0;

        math = 0;

    }

    void set\_age(int a1) {

        age = a1;

    }

    void show\_age() {

        cout << "age = " << age << endl;

    }

};

// 子類別 s1，繼承 student，採用 protected 繼承方式

class s1 : protected student {

public:

    void set\_lang(int v3) {

        lang = v3; // 可直接存取 protected 成員

    }

    void set\_math(int v4) {

        math = v4; // 可直接存取 public 成員

    }

    void setage(int v5) {

        // 無法直接存取 private 成員 age

        // 需使用 public 函數 set\_age() 間接設定

        set\_age(v5);

    }

    void show\_data() {

        // 無法直接存取 private 成員 age

        show\_age(); // 使用 public 函數印出 age

        cout << "lang = " << lang << endl;

        cout << "math = " << math << endl;

    }

};

int main() {

    s1 obj1;

    // 無法 obj1.set\_age(...)，因為繼承類別 s1 是 protected 繼承，會將原本 public 成員變成 protected

    obj1.setage(19);     // 用間接方式設定 age

    obj1.set\_lang(90);   // 設定 protected 成員 lang

    obj1.set\_math(88);   // 設定 public 成員 math

    obj1.show\_data();    // 輸出資料

    return 0;

}

## 範例：private 繼承的基本範例

#include <iostream>

using namespace std;

class Student {

private:

    int age;

protected:

    int lang;

public:

    int math;

    Student(){

        age = 0;

        lang = 0;

        math = 0;

    }

    void set\_age(int a) {

        age = a;

    }

    void show\_age() {

        cout << "age = " << age << endl;

    }

};

class S2 : private Student {

public:

    void set\_lang(int v) {

        lang = v;  // 原本 protected，現在變 private，還是可以在類別內部用

    }

    void set\_math(int v) {

        math = v;  // 原本 public，現在也變成 private，但內部可以用

    }

    void setage(int v) {

        set\_age(v);

    }

    void show\_data() {

        show\_age();

        cout << "lang = " << lang << endl;

        cout << "math = " << math << endl;

    }

};

int main() {

    S2 obj;

    obj.setage(35);

    obj.set\_lang(100);

    obj.set\_math(95);

    obj.show\_data();

    return 0;

}

# 多重繼承（Multiple Inheritance）

#include <iostream>

#include <string>  // 加這個才能用 string

using namespace std;

class Math {

    int Math\_Score;

public:

    void Math\_make(int a) { Math\_Score = a; }

    int Math\_take() { return Math\_Score; }

};

class Chinese {

    int Chinese\_Score;

public:

    void Chinese\_make(int b) { Chinese\_Score = b; }

    int Chinese\_take() { return Chinese\_Score; }

};

class History {

    int History\_Score;

public:

    void History\_make(int c) { History\_Score = c; }

    int History\_take() { return History\_Score; }

};

class Student : public Math, protected Chinese, private History {

    int Student\_Number;

    string Student\_Name;  // ✅ 改成 string 型態

public:

    Student(int d, const string& N) {  // ✅ 改成傳 string

        Student\_Number = d;

        Student\_Name = N;              // ✅ 可以用等號直接指定

    }

    void Student\_C\_make(int e) { Chinese\_make(e); }

    void Student\_H\_make(int f) { History\_make(f); }

    void Student\_Show() {

        cout << "學號: " << Student\_Number << endl;

        cout << "姓名: " << Student\_Name << endl;

        cout << "數學: " << Math\_take() << endl;

        cout << "國文: " << Chinese\_take() << endl;

        cout << "總成績: " << Math\_take() + Chinese\_take() + History\_take() << endl;

    }

};

int main() {

    Student obj(31322, "Alex");   // ✅ 可以直接傳 C 字串給 string

    obj.Math\_make(65);

    obj.Student\_C\_make(78);

    obj.Student\_H\_make(34);

    obj.Student\_Show();

    return 0;

}

## 衍生類別得建構子與解構子

當建立類別之後，會呼叫建構子。直到程式結束執行後，才會自動呼叫解構子,將不再使用的記憶體空間釋放。

衍生類別因為具有新的特性,所以不能繼承基礎類別的建構子與解構子，而必須要有自己版本的建構子與解構子。

但是針對繼承而來的特性,衍生類別就會呼叫基礎類別的建構子與解構子。

那現在要討論德問題是，建立衍生類別時要如何建立建構子及解構子呢？

* 在建立衍生類別時，會先建立基礎類別的建構子，然後再呼叫衍生類別的建構子。
* 當程式結束時，會先呼叫衍生類別的解構子，然後再呼叫基礎類別的解構子

## 範例：單一繼承下的建構子與解構子

#include <iostream>

using namespace std;

class stclass {

public:

    stclass() {

        cout << "呼叫基礎類別的建構子" << endl;

    }

    ~stclass() {

        cout << "呼叫基礎類別的解構子" << endl;

    }

};

class student : public stclass {

public:

    student() {

        cout << "呼叫衍生類別的建構子" << endl;

    }

    ~student() {

        cout << "呼叫衍生類別的解構子" << endl;

    }

};

int main(){

    student st1;

    return 0;

}

## 範例：多重繼承下的建構子與解構子

#include <iostream>

using namespace std;

// 基礎類別 stclass

class stclass {

public:

    stclass() { cout << "呼叫 stclass 類別的建構子" << endl; }

    ~stclass() { cout << "呼叫 stclass 類別的解構子" << endl; }

};

// 基礎類別 score

class score {

public:

    score() { cout << "呼叫 score 類別的建構子" << endl; }

    ~score() { cout << "呼叫 score 類別的解構子" << endl; }

};

// 衍生類別 student，繼承自 stclass 與 score

class student : public stclass, public score {

public:

    student() { cout << "呼叫類別 student 的建構子" << endl; }

    ~student() { cout << "呼叫類別 student 的解構子" << endl; }

};

// 測試函數

void call() {

    student st1; // 宣告 student 物件

}

int main() {

    call(); // 呼叫函數，建立並釋放 st1

    return 0;

}

# 多型（Polymorphism）

多型（Polymorphism）：指同一個函數名稱，在不同類別中有不同的行為表現。

當我們在程式中呼叫函數時，編譯器會將函數呼叫連接到函數的實體位置，這樣的過程稱呼為綁定(binding)

若是綁定在編譯時期就已經形成，就稱為靜態綁定

## 範例：靜態綁定(就沒有做報多形)

/\*\*

 \* 問題：雖然指標指向的是 aircraft 或 car，但呼叫的還是 vehicle 裡的函數！ => 靜態綁定

 \*/

#include <iostream>

using namespace std;

class vehicle {

public:

    void start() { cout << "運輸工具啟動" << endl; }

    void stop() { cout << "運輸工具停止" << endl; }

};

class aircraft : public vehicle {

public:

    void start() { cout << "飛行器啟動" << endl; }

    void stop() { cout << "飛行器停止" << endl; }

};

class car : public vehicle {

public:

    void start() { cout << "汽車啟動" << endl; }

    void stop() { cout << "汽車停止" << endl; }

};

int main() {

    vehicle\* ve = new vehicle();

    ve->start();  // 呼叫 vehicle::start()

    ve->stop();   // 呼叫 vehicle::stop()

    delete ve;

    ve = new aircraft();  // 雖然指向 aircraft，但型別是 vehicle\*

    ve->start();  // ❗️仍呼叫 vehicle::start()

    ve->stop();   // ❗️仍呼叫 vehicle::stop()

    ve = new car();

    ve->start();  // ❗️仍呼叫 vehicle::start()

    ve->stop();   // ❗️仍呼叫 vehicle::stop()

    return 0;

}

# 虛擬函數(virtual function)

讓我們可以做到【呼叫相同函數，卻做不同運算】

只要在基本類別的函數前面加上 virtual。

## 純虛擬函數(virtual function)

如果我今天要基本類別中宣告虛擬函數(而且是沒有內容的)

class Animal **{**

public**:**

virtual void speak**()** **=** 0**;** // 純虛擬函數

**};**

## 範例：使用 virtual 完成多型

#include <iostream>

using namespace std;

class vehicle {

public:

    virtual void start() { cout << "運輸工具啟動" << endl; }

    virtual void stop() { cout << "運輸工具停止" << endl; }

};

class aircraft : public vehicle {

public:

    void start() { cout << "飛行器啟動" << endl; }

    void stop() { cout << "飛行器停止" << endl; }

};

class car : public vehicle {

public:

    void start() { cout << "汽車啟動" << endl; }

    void stop() { cout << "汽車停止" << endl; }

};

int main() {

    vehicle\* ve = new vehicle();

    ve->start();  // 呼叫 vehicle::start()

    ve->stop();   // 呼叫 vehicle::stop()

    delete ve;

    ve = new aircraft();

    ve->start();

    ve->stop();

    ve = new car();

    ve->start();

    ve->stop();

    return 0;

}

# 基礎抽象類別

因為純虛擬函數(virtual function)無法在【單一類別、衍生類別中宣告】。

當衍生類別繼承純虛擬函數(virtual function)後，需要在衍生類別中【重新定義(override)】與【實行(implement)】所繼承的虛擬函數

不能 new，只能當基底類別

#include <iostream>

using namespace std;

class vehicle {

public:

    virtual void start() = 0; // 純虛擬

    virtual void stop() = 0;  // 純虛擬

};

class aircraft : public vehicle {

public:

    void start() override { cout << "飛行器啟動\n"; }

    void stop() override { cout << "飛行器停止\n"; }

};

class car: public vehicle{

public:

    void start() override { cout << "汽車啟動\n"; }

    void stop() override { cout << "汽車停止\n"; }

};

int main(){

    vehicle\* ve ; // 宣告基礎類別vehicle指標,抽象基礎類別不可實體化

    aircraft af;

    car cr;

    ve = &af; // 將基礎類別指標指向衍生類別aircraft

    ve->start ();

    ve->stop();

    ve = &cr; // 將基礎類別指標指向衍生類別 car

    ve->start ();

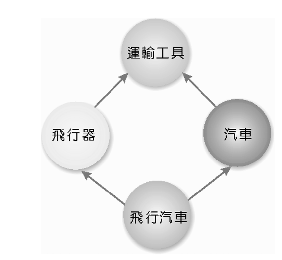
    ve->stop ();

    return 0;

}

# 虛擬基礎類別(virtula base class)

為了解決菱形問題。



解法：將 aircaft 跟 car 設定為虛擬基礎類別(virtula base class)

class 衍生類別名稱**:** virtual 繼承關鍵字 基礎類別

## 範例：虛擬基礎類別(virtula base class)

#include <iostream>

using namespace std;

// 基礎類別 vehicle

class vehicle {

public:

    void start() {

        cout << "運輸工具啟動" << endl;

    }

    void shutdown() {

        cout << "運輸工具熄火" << endl;

    }

};

// 使用虛擬繼承：virtual public vehicle

class aircraft : virtual public vehicle {

public:

    void fly() {

        cout << "飛行器飛行" << endl;

    }

    void land() {

        cout << "飛行器著陸" << endl;

    }

};

// 使用虛擬繼承：virtual public vehicle

class car : virtual public vehicle {

public:

    void go() {

        cout << "汽車啟動" << endl;

    }

    void stop() {

        cout << "汽車熄火" << endl;

    }

};

// 同時繼承 aircraft 與 car

class aircar : public aircraft, public car {

    // 不需要額外定義，直接繼承所有成員

};

int main() {

    aircar ac;

    // 呼叫 vehicle 類別的成員

    ac.start();

    // 呼叫 car 類別成員

    ac.go();

    // 呼叫 aircraft 類別成員

    ac.fly();

    ac.land();

    // 呼叫 car 類別成員

    ac.stop();

    // 呼叫 vehicle 類別的成員

    ac.shutdown();

    return 0;

}