

# Section 16 Polymorphism

## C++ 的 Polymorphism（多型）

### 定義

多型（Polymorphism）是物件導向程式設計（OOP）的三大特性之一（另兩個是封裝和繼承）。

多型的目標是：讓「同一介面」可以對「不同型別的物件」做出「不同的行為」。

---

### 類型分類

#### 1. 編譯時期多型（Compile-time Polymorphism）

又稱為靜態多型，包含：

- 函式多載（Function Overloading）
- 運算子多載（Operator Overloading）

#### 2. 執行時期多型（Runtime Polymorphism）

又稱為動態多型，透過 虛擬函式（virtual function）和 繼承（inheritance）實作。

---

### 編譯時期多型（靜態多型）

#### 函式多載範例：

```
class Printer {  
public:  
    void print(int i) { std::cout << "Printing int: " << i << '\n'; }  
    void print(double d) { std::cout << "Printing double: " << d << '\n'; }  
};
```

## ✓ 運算子多載範例：

```
class Vector {
public:
    int x, y;
    Vector(int x, int y) : x(x), y(y) {}
    Vector operator+(const Vector& other) {
        return Vector(x + other.x, y + other.y);
    }
};
```

## ★ 執行時期多型（動態多型）

### ✓ 重點條件：

1. 需要繼承（inheritance）
2. 需要虛擬函式（virtual function）
3. 需要透過指標或參考（pointer/reference）來呼叫函式

### ✓ 範例：

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

class Animal {
public:
    virtual void speak() { cout << "Animal speaks\n"; }
};

class Dog : public Animal {
public:
    void speak() override { cout << "Dog barks\n"; }
};

class Cat : public Animal {
public:
```

```
void speak() override { cout << "Cat meows\n"; }
};

void makeSound(Animal* a) {
    a->speak(); // 動態多型發生的地方
}
```

## ✓ 使用：

```
int main() {
    Dog dog;
    Cat cat;

    makeSound(&dog); // 輸出: Dog barks
    makeSound(&cat); // 輸出: Cat meows
}
```

## 示意比較

呼叫方式	是否多型
Animal a; a.speak();	✗ (靜態)
Animal* a = new Dog; a->speak();	✓ (動態)

## 小結

類型	時機	方式
靜態多型	編譯時	函式多載、運算子多載
動態多型	執行時	virtual 函式 + 指標/參考 + 覆寫(override)

# 什麼是 Virtual Function (虛擬函式)

## ✓ 定義：

在 C++ 中，`virtual` 函式是一種支援 **執行期多型 (runtime polymorphism)** 的機制。

當一個成員函式被宣告為 `virtual`，C++ 編譯器會建立一張**虛擬函式表 (vtable)**，用來在執行時動態決定要呼叫哪一個版本的函式。

## 使用時機

當**基底類別 (base class)** 的函式可能在**派生類別 (derived class)** 中被覆寫，且你想透過**指標或參考**呼叫時能呼叫到「正確版本」，就需要 `virtual`。

## 基本語法範例

```
class Base {
public:
    virtual void speak() const {
        std::cout << "I'm Base" << std::endl;
    }
};

class Derived : public Base {
public:
    void speak() const override {
        std::cout << "I'm Derived" << std::endl;
    }
};

void call(const Base& b) {
    b.speak();
}

int main() {
    Base b;
    Derived d;

    call(b); // → I'm Base
```

```
call(d); // → I'm Derived (因為 virtual)
}
```

## 為什麼需要 virtual ?

若你不使用 `virtual` :

```
class Base {
public:
    void speak() const { std::cout << "I'm Base\n"; }
};
```

即使你傳入 `Derived` 物件，只要它是 `Base&` 或 `Base*`，都會呼叫到 `Base::speak()`。

這稱為 **靜態繫結 (static binding)**，決定權在「編譯期」。


## 使用 virtual 則產生 動態繫結 (dynamic binding)


```
Base* ptr = new Derived();
ptr->speak(); // 如果 speak 是 virtual，則會呼叫 Derived::speak
```

在這種情況，程式會執行期依照 `ptr` 實際指向的物件 (`Derived`)，而不是靜態型別 (`Base`) 來決定呼叫哪個函式。

## 底層原理：VTable (虛擬函式表)

當一個類別含有虛擬函式時，編譯器會幫這個類別建立一張表格，稱為：

 **VTable (虛擬表)** 是編譯器自動產生的一張表格，記錄每個虛擬函式對應的實作。當呼叫虛擬函式時，會查表呼叫對應子類的版本。

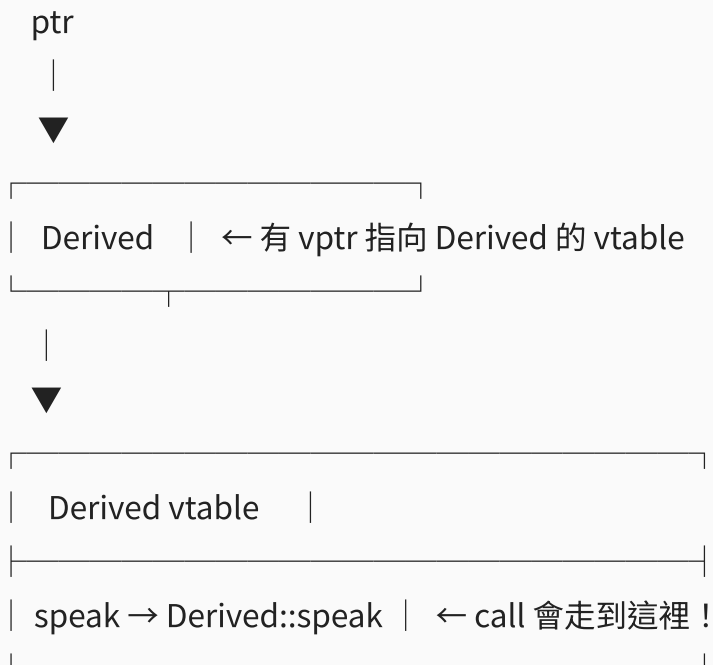
 每個物件內部都會有一個指向該表格的指標，稱為：**vptr (virtual pointer)**



## 虛擬函式原理 (VTable 概念)

```
Base* ptr = new Derived();
ptr->Speak();
```

🔍 底層模擬：



## ⚠️ 注意事項

1. 若未使用指標或參考來呼叫虛擬函式，將不會產生動態多型。
2. 若基底類沒有宣告函式為 `virtual`，即使子類覆寫，呼叫的仍是基底版本。



## 加上 `override` 關鍵字 (C++11)

```
class Derived : public Base {
public:
    void Speak() const override { ... }
};
```

## 優點：

- 如果你寫錯函式簽名，編譯器會報錯，避免你以為你覆寫了其實沒有。

## ！注意事項

狀況	結果
沒有 virtual	無法多型，呼叫的是基底版本
用指標或參考	才會發生多型（非物件呼叫）
物件呼叫（非指標/參考）	即使有 virtual 也不會多型！
建構子內呼叫 virtual 函式	只會呼叫到當前類別版本， <b>不是動態多型</b>

## ? FAQ 常見問題

### ◆ 為什麼我有 virtual 還是呼叫到 base ？

你可能是直接呼叫物件而非指標或參考：

```
Base b;
Derived d;
b = d;
b.speak(); // ❌ 不會多型 (b 是 Base 物件)
```

### ◆ virtual function 有效能問題嗎？

是的，但非常小。因為它需要：

- 多一層間接查表（vtable lookup）
- 所以比非 virtual 的呼叫慢一點點（但幾乎可以忽略）

## 延伸：純虛擬函式與抽象類別

```
class Shape {
public:
    virtual void draw() const = 0; // 純虛擬函式
};
```

這代表 Shape 是一個 **抽象類別**，不能被實例化，必須由子類實作 draw()。

## 總結表

特性	說明
virtual	開啟執行時期多型機制
override	明確標註覆寫，提高安全性
= 0	宣告純虛擬函式，讓類別成為抽象類別
必須使用指標/參考	才會產生多型效果

### #觀念釐清

## ? Derived class 中的「同名函式」是不是 virtual function ?

### 結論先講：

不是所有 derived class 中同名的函式都是 virtual function !

只有當 base class 中的函式是 virtual，derived class 中的函式才會成為 virtual function 的覆寫 (override) 版本。

## 範例比較



## ✗ 沒有 virtual，雖然名稱相同，但不是 virtual function：

```
class Base {  
public:  
    void speak() const {  
        std::cout << "Base::speak()\n";  
    }  
};  
  
class Derived : public Base {  
public:  
    void speak() const {  
        std::cout << "Derived::speak()\n";  
    }  
};
```

即使 Derived 中 `speak()` 跟 Base 同名，它也不是 virtual function，只是名稱遮蔽 (name hiding)。

## 呼叫測試：

```
Base* ptr = new Derived();  
ptr->speak(); // → Base::speak()
```

因為沒有 virtual → 靜態繫結。

---

## ✓ 有 virtual，Derived 的同名函式會成為 virtual 的 override：

```
class Base {  
public:  
    virtual void speak() const {  
        std::cout << "Base::speak()\n";  
    }  
};
```

```
class Derived : public Base {
public:
    void speak() const override { // ✓ 明確覆寫 virtual
        std::cout << "Derived::speak()\n";
    }
};
```

## 呼叫測試：

```
Base* ptr = new Derived();
ptr->speak(); // → Derived::speak()
```

因為有 virtual → 動態繫結 → 這就是執行時期多型

## 重點整理

條件	結果
base 中有 virtual	derived 中同名函式會變成 virtual 的 override (多型成立)
base 中沒有 virtual	derived 中只是重新定義一個新函式，名稱遮蔽，不是多型
derived 中函式加上 override	編譯器會檢查 base 有無 virtual function，保證正確覆寫
使用指標或參考呼叫 base 類函式	只有 virtual 函式會產生多型效果

## 使用 override 檢查很重要！

## 錯誤範例：

```
class Base {
public:
```

```
void speak() const {} // ❌ 不是 virtual
};

class Derived : public Base {
public:
    void speak() const override {} // ❌ 編譯錯誤！沒有 base virtual 函式
};
```

## 額外補充：函式簽名不同 ≠ 覆寫

```
class Base {
public:
    virtual void show(int x) const;
};

class Derived : public Base {
public:
    void show(double x) const; // ❌ 不算 override，是 name hiding
};
```

只有函式名稱 + 參數型別 + const 修飾都一樣，才算 override。

## 結語

- ▶ Derived class 的「同名函式」不一定是 virtual function。
- ▶ 只有 base class 有 virtual，才會產生多型。

#觀念釐清

## 重寫同名 method 的情況總整理

### 情況 1：Base class 沒有 virtual

```

class Base {
public:
    void speak() const {
        std::cout << "Base speaking\n";
    }
};

class Derived : public Base {
public:
    void speak() const {
        std::cout << "Derived speaking\n";
    }
};

```

## 呼叫行為：

```

Base b;
Derived d;

b.speak(); // → Base speaking
d.speak(); // → Derived speaking

Base* ptr = new Derived();
ptr->speak(); // → ! Base speaking (靜態繫結)

```

## 結論：

- Derived 的 `speak()` 不會覆寫 **base** 的函式
- 它只是「名稱遮蔽 (name hiding)」：derived 類別定義了自己的函式
- 這種情況下，不管物件實際型別，只會呼叫 base 的版本

## 情況 2：Base class 有 `virtual`

```

class Base {
public:

```

```
virtual void speak() const {
    std::cout << "Base speaking\n";
}

};

class Derived : public Base {
public:
    void speak() const override { // ✓ 這是正確的 override
        std::cout << "Derived speaking\n";
    }
};
```

## 呼叫行為：

```
Base* ptr = new Derived();
ptr->speak(); // → ✓ Derived speaking (動態繫結)
```

## 結論：

- 這才是「多型」(polymorphism)
- Base class 的 virtual 開啟了 vtable 機制
- Derived 中重寫相同函式名稱 (並配合 `override`) 是完全正確也推薦的做法

## ⚠ 注意：「簽名不同」就不算 `override` ！

```
class Base {
public:
    virtual void speak(int n) const {}
};

class Derived : public Base {
public:
    void speak() const override {} // ✗ 錯誤！簽名不同不能 override
};
```

這樣會造成編譯錯誤。

## ✅ 總結：重寫同名 method 的合法性與建議

類型	合法性	結果	建議
重寫 base 沒有 virtual 的 method	✅ 合法	名稱遮蔽，不會動態繫結	⚠️ 小心混淆，不建議
重寫 base 有 virtual 的 method	✅ 合法	多型生效，動態繫結	✅ 建議加上 override
重寫但簽名不同	✅ 合法（如果沒寫 override）	name hiding，新函式	⚠️ 小心造成誤解
加了 override 但簽名不同	❌ 編譯錯誤	--	✅ 幫你抓錯！

## 📌 建議風格

```
// base class
class Shape {
public:
    virtual void draw() const = 0; // 純虛函式
};

// derived class
class Circle : public Shape {
public:
    void draw() const override { // ✓ 最標準寫法
        std::cout << "Drawing a circle\n";
    }
};
```

## 🎯 final 是什麼？

在 C++ 中，final 可以用在：

用法位置	作用
類別後面	阻止該類別被繼承
虛擬函式後	阻止該虛擬函式被覆寫（override）

## ◆ 用法一：防止類別被繼承

```
class Animal final {
    // ...
};
```

- 表示：任何類別 **都不能繼承** `Animal`
- 嘗試繼承會導致編譯錯誤：

```
class Dog : public Animal {}; // ❌ 錯誤：Animal is final
```

## ✅ 什麼時候要這樣做？

- 當這個類別的邏輯設計上不允許擴展（例如禁止外部擴充）
- 或作為 **安全關閉繼承點**（像 `std::string` 就是 `final`）

## ◆ 用法二：防止虛擬函式被覆寫

```
class Base {
public:
    virtual void speak() final { std::cout << "Base speaking\n"; }
};

class Derived : public Base {
public:
    void speak() override {} // ❌ 錯誤：speak() is final
};
```

## ✓ 什麼時候要這樣做？

- 當你希望子類別不能再改寫某個 virtual 函式的行為
- 可用於封裝、保護關鍵邏輯，防止錯誤改寫

## 🧠 final + override 可以一起用嗎？

是的！可以這樣寫：

```
class Dog : public Animal {  
public:  
    void bark() override final;  
};
```

- 意思是：「這是對 base class 的正確覆寫，但不能再被任何 subclass 改寫。」

## 🧪 實用範例：防止意外 override 的安全封鎖

```
class Logger {  
public:  
    virtual void log(const std::string& msg) final {  
        std::cout << "[LOG] " << msg << "\n";  
    }  
};  
  
class SecureLogger : public Logger {  
public:  
    void log(const std::string& msg) override {  
        // ❌ 編譯錯誤！防止 override  
    }  
};
```

這樣你可以保證：不管誰繼承 Logger，都不能改寫 log 的行為。



## 總結：final 的用途

使用情境	final 可達成的效果
封鎖類別繼承	<code>class Foo final {}</code>
封鎖函式覆寫	<code>virtual void bar() final;</code>
增強安全與穩定性	防止未預期的擴充與 <code>override</code>
提升效能（微幅）	某些編譯器可優化 <code>final</code> 函式的呼叫

## Pure Virtual Function是什麼？

### 一句話先說明：

一般 **virtual function**：在 base class 中有「預設實作」，可以被 `override`。

**pure virtual function**：在 base class 中 **沒有實作**，強迫所有 derived class 必須 `override`，否則不能實例化。

### 語法差異

#### 一般 **virtual function**：

```
class Animal {
public:
    virtual void speak() const {
        std::cout << "Animal speaks\n";
    }
};
```

- ✓ 可以有實作
- ✓ 派生類（如 Dog）可以 `override`，也可以不 `override`
- ✓ `Animal` 可以被建立實例

## ✓ pure virtual function (純虛擬函式) :

```
class Animal {
public:
    virtual void speak() const = 0; // 🔥 純虛擬函式
};
```

- ✗ 不能有實作 (但可以在 class 外部提供)
- ! 派生類 必須 **override**，否則也變成 abstract class
- ✗ Animal 不能實體化 (抽象類別)

## 📖 實例比較

```
class Animal {
public:
    virtual void speak() const = 0; // 純虛擬函式
};

class Dog : public Animal {
public:
    void speak() const override {
        std::cout << "Woof!\n";
    }
};

int main() {
    Animal a; // ✗ 編譯錯誤：Animal is abstract
    Dog d;    // ✓ OK
    d.speak(); // → "Woof!"
}
```

## 💡 兩者的意圖差異

比較面向	virtual	pure virtual (=0)
有沒有預設實作？	✅ 有	❌ 沒有（可選 class 外實…
是否要求 override？	❌ 不強制	✅ 必須 override
base class 可否建立實例？	✅ 可以	❌ 不行（抽象類別）
設計意圖	預設行為，可被覆蓋	抽象接口，強迫子類實作

## 🔑 補充技巧：純虛擬也可以「偷偷」有實作！

```
class Animal {
public:
    virtual void speak() const = 0;
};

void Animal::speak() const {
    std::cout << "[default animal sound]\n";
}
```

雖然 `= 0` 宣告為純虛擬函式，但仍可在 class 外實作，供 `derived class` 使用 `Animal::speak()` 明確呼叫。

## 🏠 小結表格

項目	virtual	pure virtual (= 0)
Base class 可以實體化？	✅ 可以	❌ 不行
Derived class 一定要 override 嗎？	❌ 不一定	✅ 一定要
Base class 有沒有預設行為？	✅ 有	❌ 沒有（但可手動寫）
多型是否可用？	✅ 可以	✅ 可以

## 🧠 抽象基底類別是什麼？

- ◆ 一個無法被實體化（無法創建物件）的類別
- ◆ 至少包含一個純虛擬函式（pure virtual function）
- ◆ 用來作為其他類別的父類（基底類別）

```
class Shape {
public:
    virtual void draw() const = 0; // 純虛擬函式
};
```

這個 Shape 類別就是一個抽象基底類別。

## 📌 判斷標準：什麼情況下類別是 abstract base class？

條件	是否為抽象類別？
有一個以上純虛擬函式	✅ 是
沒有純虛擬函式	❌ 否
只有 virtual 函式但有實作	❌ 否

## ❌ 抽象基底類別不能做的事

Shape s; // ❌ 編譯錯誤：Shape 是抽象類別，不能創建實例

但它可以被繼承：

```
class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing Circle\n";
    }
};
```

```

}
};

```

## ✓ 使用抽象基底類別的目的

目的	說明
✓ 定義「介面」	規定子類必須實作哪些功能
✓ 觸發多型	搭配 Shape* 或 Shape& 呼叫 virtual 函式
✓ 強制一致性	子類別不實作抽象函式就不能實體化
✓ 分離設計與實作	使用者操作的是抽象介面，內部實作可以隨時替換

## 🎯 範例：Shape 抽象類別

```


class Shape {
public:
    virtual void draw() const = 0; // 純虛擬
    virtual ~Shape() = default;
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a Circle\n";
    }
};

class Square : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a Square\n";
    }
};

```

```

void render(const Shape& shape) {
    shape.draw(); //  多型發生
}

int main() {
    Circle c;
    Square s;
    render(c); // Drawing a Circle
    render(s); // Drawing a Square
}

```

## 你可以把 abstract base class 想成什麼？

就像是：

✨ 「我定義了一組規則（function 接口），所有子類別都要自己實作，否則就不能用。」

## 小結

抽象基底類別（Abstract Base Class）	說明
含有至少一個純虛擬函式	virtual func() = 0;
不能創建實例	Base b; 編譯錯
可作為指標或參考使用	Base* ptr = new Derived();
子類別必須實作所有純虛擬函式	否則該子類也會變成抽象類別

## 以base class作為interface?

### 一句話解釋：

以 base class 作為 interface，就是把 base class 設計成「只定義要實作的功能（method），不包含任何實際邏輯」，並讓 derived class 來實作這些功能。

## 換句話說：

你定義一個**抽象基底類別**（abstract base class, ABC），它的唯一作用是：

- 規範 derived class 必須提供哪些功能
- 提供一個 統一的使用介面

這樣的 base class 就是「interface」的角色。

## C++ 如何實作 interface？

不像 Java 有 interface 關鍵字，

C++ 是透過「只包含純虛擬函式的 class」來實作 interface：

```
class Printable {  
public:  
    virtual void print() const = 0; // 純虛擬函式 = 規定介面  
    virtual ~Printable() = default; //  一定要加虛擬解構子  
};
```

這樣的 Printable 就是個「interface」。

## 實作端（Derived Class）要怎麼用？

```
class Document : public Printable {  
public:  
    void print() const override {  
        std::cout << "Printing document..." << std::endl;  
    }  
};  
  
class Image : public Printable {  
public:  
    void print() const override {
```

```
std::cout << "Printing image..." << std::endl;
}
};
```

## ✅ 使用端（以 base class 作為介面）怎麼寫？

```
void printAnything(const Printable& p) {
    p.print(); // 多型發生在這裡！
}

int main() {
    Document d;
    Image img;

    printAnything(d); // → Printing document...
    printAnything(img); // → Printing image...
}
```

## 🧠 小結：為什麼要「以 base class 作為介面」？

目的	說明
🌟 支援多型	透過 base pointer/reference 使用 derived class
✂️ 解耦實作	呼叫端不用知道是哪個具體類別，只要會 print() 就好
📦 實現設計模式	很多設計模式（Strategy、Visitor、Factory）都依賴這種介面
✅ 強制行為一致	所有繼承的類別都「被強制」要提供一樣的函式簽名

## ✅ Interface 的 C++ 常見寫法範本



```
class InterfaceName {
public:
    virtual ReturnType functionName(...) = 0; // 純虛擬函式
    virtual ~InterfaceName() = default;    // 一定要有虛擬解構子
};
```

## ← END 最後簡表：Base Class 作為 Interface 的特徵

特徵	是否存在
成員變數	✗ 通常沒有
非純虛擬函式	✗ 沒有實作邏輯
至少一個純虛擬函式	✓ 有
虛擬解構子	✓ 必加
可以被實體化？	✗ 不行（抽象類別）
目的	建立統一介面

## 🔍 現象名稱：Name Hiding（名稱遮蔽）

### #觀念釐清

在 C++ 中，如果 derived class 定義了一個與 base class 同名的函式，即使參數不一樣，base class 中所有同名函式都會被「遮蔽（hidden）」，除非你特別指出要使用 base 的版本。

## ✓ 正確理解

derived 的確繼承了 base class 的所有 public 和 protected method，但只要你在 derived 中定義了「同名」函式，哪怕參數不同，base 的所有同名函式就會被遮蔽。

此時：

- `derivedObj.name()` → 預設呼叫 `derived class` 的版本（就算參數不一樣也會遮掉 `base` 的所有同名函式）
- 若你要用 `base` 的版本，需要 `Base::name()` 明確呼叫

## 例子：Name Hiding vs 正確繼承呼叫

```
class Base {
public:
    void hello() {
        std::cout << "Hello from Base\n";
    }

    void hello(int n) {
        std::cout << "Hello from Base with int\n";
    }
};

class Derived : public Base {
public:
    void hello() {
        std::cout << "Hello from Derived\n";
    }
};
```

## 呼叫：

```
Derived d;
d.hello();    // → Hello from Derived
d.hello(10);  // ✗ 編譯錯誤！Base::hello(int) 被遮蔽了
```

## 解法：用 `using` 引入 `base` 的同名方法

```
class Derived : public Base {
public:
```

```
using Base::hello; // → 解除名稱遮蔽

void hello() {
    std::cout << "Hello from Derived\n";
}


};
```

## 再呼叫：

```
Derived d;
d.hello(); // → Hello from Derived
d.hello(42); // → Hello from Base with int ✓
```

## 小結

現象	說明
Derived 中定義了與 Base 同名的函式	Base 所有同名函式（不管參數是否相同）都會被隱藏
這叫做？	Name hiding（名稱遮蔽）
要怎麼讓 base 的同名函式也能呼叫？	在 derived 中加入 <code>using Base::methodName;</code>

 **重點：**Derived 定義了 **完全相同簽名** 的函式，會「**override**」或「**遮蔽**」base 的函式（取決於是否 **virtual**）

### #觀念釐清

C++ 的 `using Base::methodName;` 只是把 base class 的名字帶進來，不會改變虛擬 dispatch 或遮蔽的行為本質。

## 範例 1：參數相同 → 呼叫 derived 版本

```
class Base {
public:
    void greet() {
        std::cout << "Hello from Base\n";
    }
};

class Derived : public Base {
public:
    using Base::greet;

    void greet() { // same name & same signature
        std::cout << "Hello from Derived\n";
    }
};
```

### 呼叫：

```
Derived d;
d.greet(); // → "Hello from Derived" ✓
```

➔ 因為 `Derived::greet()` 和 `Base::greet()` 完全相同，所以 derived 的會「遮蔽」base 的版本。

## 範例 2：參數不同 → using 有效解除遮蔽

```
class Base {
public:
    void greet(int n) {
        std::cout << "Hello from Base with int = " << n << "\n";
    }
};

class Derived : public Base {
```

```
public:
    using Base::greet;

    void greet() {
        std::cout << "Hello from Derived\n";
    }
};
```

## 呼叫：

```
Derived d;
d.greet(); // → Hello from Derived
d.greet(42); // → Hello from Base with int = 42 ✓
```

✓ 這就是 `using` 發揮作用的場景：解除「參數不同」情況下的 `name hiding`。

## 小結

情況	行為
參數相同	Derived 的方法會完全遮蔽 base，無法呼叫 base 版本（即使有 <code>using</code> ）
參數不同	預設會 <code>name hiding</code> ，但可用 <code>using Base::xxx;</code> 解決
虛擬函式 + 相同簽名	若 base 是 <code>virtual</code> ，則 derived 是 <code>override</code> ，透過 <code>vtable</code> 執行
非虛擬函式 + 相同簽名	那只是名稱遮蔽（靜態繫結），不會發生多型行為

太好了！這是你要的 C++ 名稱遮蔽（Name Hiding）vs 函式覆寫（Override）vs 多型（Polymorphism）的對照筆記，一次幫你釐清：

## C++ 函式重定義行為總整理筆記

## #觀念釐清

特性	說明	是否呼叫 Derived 版本？	可否呼叫 Base 版本？	備註
名稱遮蔽 (Name Hiding)	Derived 定義同名函式 (不管參數是否相同)，會遮蔽 Base 的所有同名函式	✅ (預設呼叫 derived)	❌ (參數不同也遮)	用 <code>using Base::name</code> 解決
簽名完全相同	Derived 定義和 Base 名稱與參數完全相同的函式	✅	❌ (即使有 using)	此情況等同於 override
參數不同	Derived 定義同名不同參數的函式	✅	⚠️ <code>Base::func(x)</code> 編譯錯，除非加 using	<code>using Base::func;</code> 解決 name hiding
函式非 virtual	即使名稱與參數相同，也不構成多型	✅	❌	靜態繫結，不會根據實際物件類型變化
virtual + 相同簽名	Base 是 virtual，Derived 同簽名函式則為 override	✅ (透過 pointer/reference 呼叫)	✅ (可用 <code>Base::func()</code> 指定)	多型成立 (vtable dispatch)
override 指定子句	C++11 的語法，確認函式真的 override base	✅ (必須與 base 完全一致)	編譯器幫你檢查	最安全的寫法！

## 對照範例一覽

### ① 名稱遮蔽 (不會多型)

```
class Base {
public:
    void speak(int x) { std::cout << "Base\n"; }
};

class Derived : public Base {
public:
    void speak() { std::cout << "Derived\n"; }
};

Derived d;
d.speak(); // Derived
d.speak(5); // ❌ 編譯錯誤：Base::speak 被遮蔽
```


✅ 解法：

```
class Derived : public Base {
public:
    using Base::speak; // ✅ 解決 name hiding
    void speak() { std::cout << "Derived\n"; }
};
```

## ② 多型 + override 成功

```
class Base {
public:
    virtual void draw() const { std::cout << "Base\n"; }
};

class Derived : public Base {
public:
    void draw() const override { std::cout << "Derived\n"; }
};
```

```
Base* ptr = new Derived;
ptr->draw(); //  Derived (動態繫結)
```


### ③ override 錯誤 (參數不同)

```
class Base {
public:
    virtual void run(int x) const {}
};

class Derived : public Base {
public:
    void run() const override {} //  編譯錯誤：簽名不一致
};
```

### 建議寫法風格

```
class Base {
public:
    virtual void foo() const;
};

class Derived : public Base {
public:
    void foo() const override; //  最佳習慣
};
```

- Base → virtual 是關鍵
- Derived → override 可強制編譯器檢查



## 總結心智圖



Base 定義？



有 virtual 嗎？

是      否



Derived 同名同簽？      僅為遮蔽，靜態繫結

↓ 是

這是 override，多型成立

使用 pointer/reference 呼叫

### #觀念釐清

**✓ 想要使用多型，不一定要在 main 裡用指標或 reference**

### #觀念釐清

**✓ 只要設計的函式參數是 base class 的 reference 或指標，就可以觸發多型**

**🎯 說得更具體一點：**

設計階段	多型條件
📦 函式設計	接收 Base& 或 Base* (不是 by value)
✂ 使用階段	傳入 Derived 類別的物件

設計階段	多型條件
⚡ 呼叫 virtual 函式	就會觸發動態繫結（多型）

## ✅ 範例一：透過 reference

```
void do_action(Account& acc) {
    acc.withdraw(100); // ✅ 多型可能在這裡發生
}
```

在 main() 中：

```
Trust trust_account;
do_action(trust_account); // ✅ 傳入 derived class，會呼叫 Trust::withdraw()
```

## ✅ 範例二：透過 pointer

```
void do_action(Account* acc) {
    acc->withdraw(100); // ✅ 這裡一樣可以觸發多型
}
```

在 main() 中：

```
Trust trust_account;
do_action(&trust_account); // ✅ 一樣是 base pointer 指向 derived instance
```

## ❌ 反例：傳 by value 無法觸發多型

```
void do_action(Account acc) {
    acc.withdraw(100); // ❌ 即使是 virtual，也只能呼叫 Account::withdraw()
}
```

為什麼不行？

- 因為 `acc` 是值複製 (pass by value)
- 傳入 derived 物件時會「切割 (object slicing)」，只保留 base class 的部分
- 即使有 virtual，也無法動態跳轉

## 小結

傳遞方式	可否觸發多型	原因
<code>Base obj</code> (by value)	✗ 否	物件切割，無法保留 derived 行為
<code>Base&amp;</code> (by reference)	✓ 可	保留物件型別資訊
<code>Base*</code> (by pointer)	✓ 可	同上，保留型別資訊
<code>std::unique_ptr&lt;Base&gt;</code> / <code>shared_ptr&lt;Base&gt;</code>	✓ 可	只要是「以 base 為介面」的指標都可

如果你未來打算設計一個「操作各種帳戶」的系統，可以只寫：

```
void process(Account& acc) {
    acc.withdraw(100);
}
```

然後對任何子類別都適用，這就是多型在設計上最強大的地方。