

Section 15 Inheritance

C++ 繼承 (Inheritance)

定義 | Definition

在 C++ 中，**繼承**是一種讓新類別（稱為「子類別」或「衍生類別」）**重用**現有類別（稱為「父類別」或「基底類別」）的資料成員與成員函式的機制。

Inheritance allows a class (derived class) to acquire the properties and behaviors (members) of another class (base class), promoting code reuse and extensibility.

基本語法 | Basic Syntax

```
class Base {
public:
    void sayHello() {
        std::cout << "Hello from Base!" << std::endl;
    }
};

class Derived : public Base {
public:
    void sayHi() {
        std::cout << "Hi from Derived!" << std::endl;
    }
};

int main() {
    Derived d;
    d.sayHello(); // 繼承自 Base
    d.sayHi();    // 自己的函式
}
```

存取權限 (Access Specifiers)

```
class Derived : public Base // 公開繼承
class Derived : protected Base // 保護繼承
class Derived : private Base // 私有繼承
```

Base 成員權限	public 繼承後	protected 繼承後	private 繼承後
public	public	protected	private
protected	protected	protected	private
private	不可繼承	不可繼承	不可繼承

類型關係圖 | Class Hierarchy Diagram







```
Base
↑
Derived
↑
MoreDerived
```

繼承可以形成階層式結構 (hierarchy)，利於邏輯分類。

注意事項 | Important Notes

1. C++ 不支援多重基底類別中的同名函式自動解決，要用 scope resolution。
2. 建構子與解構子不會自動繼承（但父類的建構子會被呼叫）。
3. 要記得把解構子寫成 virtual，避免資源釋放不完全。
4. 支援多重繼承 (multiple inheritance)，但會有菱形繼承問題（可用 virtual inheritance 解決）。

public inheritance vs composition

項目	public inheritance (公開繼承)	composition (組合)
 關係語意	是一種「is-a」關係	是一種「has-a」關係
 繼承方式	使用 <code>class Derived : public Base</code>	在類別中包含另一個類別作為成員
 可替換性	子類別可替代父類別位置	不可替代，因為不是同一型別
 彈性	繼承結構較緊密、不易更改	組合結構鬆散、較易替換與擴充
 多重重用	易受限制（C++ 有菱形繼承問題）	組合可以同時使用多個物件
 用途	模型化行為繼承	模型化功能委派、功能組裝

◆ public inheritance 範例：「是一個」關係


```

class Animal {
public:
    void eat() { std::cout << "Animal eats\n"; }
};

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


int main() {
    Dog d;
    d.eat(); // Dog 是 Animal，所以可以吃
    d.bark();
}

```

 語意：Dog 是一種 Animal (is-a)，可以自然地吃（繼承 eat()），也是多型用途。

◆ composition 範例：「擁有一個」關係

```
class Engine {  
public:  
    void start() { std::cout << "Engine starts\n"; }  
};  
  
class Car {  
private:  
    Engine engine; // 組合關係  
public:  
    void drive() {  
        engine.start(); // 委託 engine 執行任務  
        std::cout << "Car drives\n";  
    }  
};
```

 語意：Car 並不是 Engine，但 Car 擁有一個 Engine (has-a)。這讓你可以更靈活地替換或擴充 Engine 類別。

何時用 public inheritance ?

當你要表達：

- 「A 是 B 的一種」
- 希望使用 多型 (virtual functions)
- 子類應完全遵守父類的語意與規則 (Liskov Substitution Principle)

 適合用 public inheritance。

何時用 composition ?

當你要：

- 重用別的類別的功能
- 將物件作為組件插入 (Plug-and-Play)
- 易於日後替換、測試、擴充

✅ 適合用 composition。

實際開發建議（重要）

✅ 優先使用組合（composition over inheritance）是現代 C++ 的主流原則，因為它更靈活、更好測試、更少耦合。

題目背景：圖形繪製系統

設計一個繪圖系統，支援各種圖形：Circle、Rectangle 等。我們要實作一個 draw() 函式來畫出這些圖形。

方式一：使用 public inheritance（繼承）

```
#include <iostream>
#include <vector>
#include <memory>

class Shape {
public:
    virtual void draw() const {
        std::cout << "Drawing shape...\n";
    }

    virtual ~Shape() = default;
};

class Circle : public Shape {
public:
    void draw() const override {
```

```

        std::cout << "Drawing circle\n";
    }
};

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

int main() {
    std::vector<std::shared_ptr<Shape>> shapes;
    shapes.push_back(std::make_shared<Circle>());
    shapes.push_back(std::make_shared<Rectangle>());

    for (const auto& shape : shapes)
        shape->draw();
}

```

✓ 特點：

- Circle 、 Rectangle 是 Shape 的一種 (is-a)
- 可用多型 (polymorphism) 處理所有 Shape 類別
- 如果有新圖形，只需繼承並 override draw()

■ 方式二：使用 composition (組合)

```

#include <iostream>
#include <vector>
#include <memory>

// 繪圖策略介面
class DrawingStrategy {
public:
    virtual void draw() const = 0;
};

```

```

    virtual ~DrawingStrategy() = default;
};

// 各種繪圖策略
class CircleDrawing : public DrawingStrategy {
public:
    void draw() const override {
        std::cout << "Drawing circle\n";
    }
};

class RectangleDrawing : public DrawingStrategy {
public:
    void draw() const override {
        std::cout << "Drawing rectangle\n";
    }
};

// Shape 組合繪圖策略
class Shape {
private:
    std::shared_ptr<DrawingStrategy> strategy;

public:
    Shape(std::shared_ptr<DrawingStrategy> s) : strategy(s) {}

    void draw() const {
        strategy->draw(); // 委託給策略物件
    }
};

int main() {
    std::vector<Shape> shapes;
    shapes.emplace_back(std::make_shared<CircleDrawing>());
    shapes.emplace_back(std::make_shared<RectangleDrawing>());

    for (const auto& shape : shapes)

```

```
shape.draw();
}
```

✓ 特點：

- Shape 擁有一個繪圖策略 (has-a)
- 若未來要改變畫法，只需更換策略物件
- 不用依賴繼承結構，更適合模組化擴充

📄 結論比較

項目	public inheritance	composition
結構語意	Circle 是 Shape	Shape 擁有一個繪圖策略
新增圖形方式	繼承 Shape 並覆寫	實作新策略，注入給 Shape
擴充性	較差（耦合緊）	較佳（策略可替換）
重用性	較差	可將策略獨立重用
維護性	若類別變複雜會難以維護	組合較容易測試與維護

🛡️ protected 關鍵字介紹

🔍 基本意義

protected 是 C++ 的一種存取權限修飾詞 (access specifier)，其權限介於 public 與 private 之間。

權限類型	本類別中可見	子類別中可見	類別外可見
public	✓	✓	✓
protected	✓	✓	✗
private	✓	✗	✗

🔑 protected 表示「只能由自己和子類別存取，但類別外不能存取」。



protected 在繼承中的角色

在繼承中，`protected` 讓子類別可以看到並使用父類別的資料或函式，但不會暴露給外部使用者。



範例：protected 欄位

```
#include <iostream>

class Base {
protected:
    int value = 42;
};

class Derived : public Base {
public:
    void showValue() {
        std::cout << "Value: " << value << std::endl; // ✓ 合法：子類可以存取 protected
    }
};

int main() {
    Derived d;
    d.showValue();

    // std::cout << d.value; ✗ 編譯錯誤：main() 是外部函式，不能存取 protected 成員
}
```



使用場景與原則

使用情境	是否建議使用 <code>protected</code>
讓子類別能存取但不外露的內部實作	✓ 可行
避免使用 <code>private</code> 使子類別重複定義資料	✓ 合理
替代 <code>public</code> 設定為外部可用的成員	✗ 不建議

使用情境	是否建議使用 <code>protected</code>
過度依賴使子類耦合基類實作	✗ 不建議（違反 encapsulation）

`protected` vs. `private` 在 inheritance 中的影響

```
class Base {
protected:
    int a = 1;
private:
    int b = 2;
};

class Derived : public Base {
public:
    void test() {
        std::cout << a << std::endl; // ✓ OK
        // std::cout << b << std::endl; ✗ 編譯錯誤：private 無法存取
    }
};
```

- `protected` 可讓 `Derived` 使用 `a`
- `private` 則完全封鎖 `b`，只能由 `Base` 操控


在不同繼承類型下的影響

```
class Base {
protected:
    int x;
};

class PublicDerived : public Base {};
```

```
class ProtectedDerived : protected Base {};  
class PrivateDerived : private Base {};
```

成員	在 PublicDerived 可見	在 ProtectedDerived 可見	在 PrivateDerived 可見
x	protected	protected	private

 **結論：** `protected` 成員的可見性在繼承中「保留」，但會根據繼承類型調整權限（越來越封閉）。

建議使用時機

使用 `protected`

- 當你希望子類可以直接存取某些欄位或函式（例如抽象基礎類別內的共通資料）

避免過度使用

- 過度依賴 `protected` 可能導致子類與父類緊密耦合，違反封裝原則（encapsulation）

最佳實踐（Modern C++ 建議）：

- 多數情況優先使用 `private + public getter/setter`
- 若真需要子類使用，才使用 `protected`

? 為什麼 `protected` 有時會違反 **encapsulation**（封裝性）？

先回顧封裝（Encapsulation）是什麼？

封裝的目標是：

「隱藏實作細節，讓使用者只透過明確定義好的介面來操作物件。」

也就是說，一個物件的內部狀態（欄位、邏輯）不應暴露給使用者或子類直接操作，這樣可以：

- 隨時替換實作細節（不影響外部使用者）

- 控制資料一致性與合法性
 - 降低耦合性
-

💣 **protected** 違反封裝的原因是什麼？

✅ **private** 做得好：

```
class Base {  
    private:  
        int data;  
    public:  
        void setData(int d) {  
            if (d >= 0) data = d;  
        }  
};
```

- 外部無法直接改動 data
 - 改變內部邏輯不會影響使用者
-

❌ **protected** 把實作細節暴露給了子類：

```
class Base {  
    protected:  
        int data; // 🚨 子類直接看到與操作  
};  
  
class Derived : public Base {  
    public:  
        void breakEncapsulation() {  
            data = -999; // 沒有經過檢查，可能破壞狀態  
        }  
};
```

➡ 問題：

- 子類現在「知道」並依賴 Base 的內部資料結構

- 將來如果 `Base` 想改變 `data` 的設計（例如換成 `std::optional` 或加入驗證），就會導致所有子類壞掉

這會造成：

❗ 子類 **tightly coupled**（緊密耦合）到父類的實作細節，也就是封裝失敗

🔄 對比：正確封裝方式（用 `private` + 公開介面）

```
class Base {
private:
    int data;
protected:
    void setData(int d) { data = d; }
    int getData() const { return data; }
};

class Derived : public Base {
public:
    void safeChange() {
        setData(42); // ✅ 間接操作，維持封裝性
    }
};
```

🔒 封裝性仍然保留，子類只是透過合法管道使用功能，不依賴具體實作。

✅ 總結：為什麼過度使用 `protected` 是危險的？

問題	說明
❌ 子類能直接操作內部資料	沒有任何檢查或防錯機制
❌ 子類依賴實作細節	難以修改父類，不穩定
❌ 形成高度耦合	破壞模組化與彈性設計
✅ 正確設計	使用 <code>private</code> 成員 + <code>protected</code> 函式作為介面



Constructor 在繼承中的行為



基本原則

- 建構時，會先呼叫基底類別（Base Class）的 constructor，再呼叫派生類別（Derived Class）的 constructor。
- 如果你沒有指定，會自動呼叫 Base 的 預設建構子（default constructor）。

◆ 範例：基本建構順序

```
#include <iostream>

class Base {
public:
    Base() { std::cout << "Base constructor\n"; }
};

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

int main() {
    Derived d;
}
```

 輸出：

```
Base constructor
Derived constructor
```

◆ 若 Base 沒有預設建構子怎麼辦？

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

```
class Derived : public Base {
public:
    Derived() : Base(10) {
        std::cout << "Derived constructor\n";
    }
};
```

 你必須在 Derived 的 constructor initializer list 中手動呼叫 Base 的建構子。

2. Destructor 在繼承中的行為

基本原則

- 解構順序相反：先解構 Derived → 再解構 Base。
- Base 的 destructor 必須是 `virtual` 才能確保用 base 指標刪除 derived 物件時，會完整呼叫 derived 的 destructor。

◆ 不用 virtual 的危險情況：

```
class Base {
public:
    ~Base() { std::cout << "Base destructor\n"; }
};

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

int main() {
    Base* p = new Derived();
    delete p; //  只會呼叫 Base::~~Base()，導致資源沒正確釋放
}
```

 輸出：

Base destructor

✓ 正確寫法：使用 virtual destructor

```
class Base {
public:
    virtual ~Base() { std::cout << "Base destructor\n"; }
};

class Derived : public Base {
public:
    ~Derived() { std::cout << "Derived destructor\n"; }
};
```

📌 輸出（正確）：

```
Derived destructor
Base destructor
```

🔧 延伸：Constructor Chaining

你可以讓 Derived class 同時初始化自己的欄位與 Base class 的欄位：

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

class Derived : public Base {
    int d;
public:
    Derived(int a, int b) : Base(a), d(b) {
        std::cout << "Derived(" << b << ")\n";
    }
};
```






```

}
};

```

🧠 小結：Constructor / Destructor 規則整理

項目	規則與建議
 初始化順序	Base constructor → Derived constructor
 解構順序	Derived destructor → Base destructor
⚠️ virtual destructor	若用 base pointer 操作 derived，Base destructor 必須為 virtual
 constructor initializer list	用來手動指定 base class constructor
🚫 constructor 不會被繼承	Derived 不會自動擁有 Base 的 constructor，必須自定