

Section 13-4 Copy Constructor

什麼是 Copy Constructor（複製建構子）？

Copy Constructor 是用來用一個已經存在的物件來初始化另一個同類型的新物件的特殊建構子。

◆ 語法格式

```
ClassName(const ClassName& other);
```

- 參數是對同類型物件的**常數參考**（const &），避免不必要的複製並保護原物件不被修改。
- 主要目的是：用已有物件複製一個新物件。

◆ 什麼時候會自動呼叫？

1. 用已存在的物件建立新物件時：

```
Student s1("Tom", 20);  
Student s2 = s1; // 呼叫 copy constructor
```

2. 當物件以值傳遞（pass-by-value）做函式參數時：

```
void foo(Student s); // 這裡會呼叫 copy constructor 複製參數  
  
foo(s1);
```

3. 函式回傳值時（回傳類別物件）也可能觸發複製建構：

```
Student createStudent() {  
    Student s("Jerry", 18);  
    return s; // 回傳時會呼叫 copy constructor（視編譯器優化而定）  
}
```

◆ C++ 預設提供的複製建構子

若你沒有自己寫複製建構子，編譯器會自動產生一個淺拷貝（shallow copy）的複製建構子：

- 逐成員複製：每個成員使用其 own copy constructor（或直接複製數值）複製。
- 如果成員是指標，則只會複製指標的地址，不會複製指標所指向的物件 → 可能造成多個物件共用同一塊記憶體。

◆ 重點總結

事項	說明
Copy Constructor 是什麼	用一個物件複製並初始化另一個物件
預設複製建構子	編譯器會產生淺拷貝版本
淺拷貝問題	指標成員只複製指標位址，可能造成重複刪除或資源競爭
自己寫複製建構子時機	類別有動態配置資源（指標等）時，需深拷貝
複製建構子參數	一定是 <code>const ClassName&</code>

 **複製建構子（Copy Constructor）只能有一個，因為它的定義是固定的**

```
ClassName(const ClassName& other);
```

原因：

- 複製建構子的目的就是用「同一個類別的物件」來複製一個新物件，參數型態就是該類別的 `const` 引用。
- 如果有多個複製建構子，編譯器就無法判斷你到底要呼叫哪一個，造成模糊不清（ambiguous）。

小提醒：

- 你可以重載建構子（Constructor overloading），讓不同參數組合呼叫不同建構子，但複製建構子本身就是「用同一類別的 `const` 引用參數」來定義的，不

會有多個版本。

- 不過你可以搭配**移動建構子 (Move Constructor)**，兩者合起來是類別的「特殊建構子」：
 - 複製建構子：`ClassName(const ClassName& other);`
 - 移動建構子：`ClassName(ClassName&& other);`

這兩個有不同參數型態，算是不同的建構子，但都是「特殊建構子」，其中複製建構子只能有一個。

總結

事項	狀況說明
複製建構子數量	只能有一個
為什麼？	參數型態固定且唯一（ <code>const ClassName&</code> ）
其他相似建構子	可搭配移動建構子，有不同參數型態，非複製建構子

淺拷貝vs深拷貝

什麼是淺層複製 (shallow copy) ?

```
#include <iostream>

using namespace std;

class Shallow {
private:
    int *data;
public:
    void set_data_value(int d) { *data = d; }
    int get_data_value() { return *data; }
    // Constructor
    Shallow(int d);
    // Copy Constructor
    Shallow(const Shallow &source);
    // Destructor
```

```

    ~Shallow();
};

Shallow::Shallow(int d) {
    data = new int;
    *data = d;
}

Shallow::Shallow(const Shallow &source)
: data(source.data) {
    cout << "Copy constructor - shallow copy" << endl;
}

Shallow::~Shallow() {
    delete data;
    cout << "Destructor freeing data" << endl;
}

void display_shallow(Shallow s) {
    cout << s.get_data_value() << endl;
}

int main() {

    Shallow obj1 {100};
    display_shallow(obj1);

    Shallow obj2 {obj1};
    obj2.set_data_value(1000);

    return 0;
}

```

當你的類別成員是「指標」時，預設的複製建構子只會複製這個指標的值（也就是記憶體位置的位址），不會去複製「那塊指標所指向的資料」。

結果：

- 你會有兩個物件指向同一塊記憶體。

- 其中一個物件刪除或改變內容，另一個也會受到影響。
- 一不小心會造成 **double free** 或 **dangling pointer** 問題。

✅ 程式碼概觀與核心重點

```
Shallow obj1 {100};           // 建立 obj1，data 指向 heap 上的 int(100)
display_shallow(obj1);        // 傳值呼叫，會呼叫 copy constructor -> shallow copy
Shallow obj2 {obj1};          // 用 obj1 初始化 obj2，copy constructor 被呼叫 -> shallow copy
obj2.set_data_value(1000);    // 改變 obj2 中指標指向的值
```

🧠 重點是：obj1 和 obj2 的 data 指向相同的記憶體位置！

💀 Shallow Copy 的問題：重複釋放記憶體

```
Shallow::Shallow(const Shallow &source)
: data(source.data) { //assign source.data的位址給data
    cout << "Copy constructor - shallow copy" << endl;
}
```

這個 copy constructor 只是把指標直接複製，所以兩個物件（如 obj1 和 obj2）共用同一塊堆積記憶體。

當 main 結束時：

- obj2 被銷毀，呼叫 destructor → delete data，釋放 heap 空間。
- obj1 被銷毀，也呼叫 destructor → delete data，但這時記憶體已經被釋放，導致 **double free** 錯誤（未定義行為 UB）。

✂ 解法：使用 Deep Copy（深層複製）

讓我們改寫 copy constructor，使其建立新記憶體，並複製值：

```
Shallow::Shallow(const Shallow &source) {
    data = new int;
    *data = *source.data; // 複製值而不是指標
```

```
cout << "Copy constructor - deep copy" << endl;
}
```

✅ 深層複製後，obj1 和 obj2 就擁有獨立的 data：

- obj2.set_data_value(1000) 不會改變 obj1 的值。
- obj1 和 obj2 各自的 destructor 各自 delete 自己的 data，✅ 安全無誤！

💡 額外補充：display_shallow 的傳值也會觸發複製！

```
void display_shallow(Shallow s) {
    cout << s.get_data_value() << endl;
}
```

這裡 s 是 pass by value，也就是會觸發 複製建構子，然後結束函數時會呼叫 destructor。

也就是說：

```
display_shallow(obj1);
```

會輸出：

```
Copy constructor - shallow copy
100
Destructor freeing data
```

這代表複製出來的那個暫時物件（s）先把 heap delete 掉，之後 obj1 delete 時就 GG 了。

🌴 物件生命週期

💡 Shallow Copy

```
int main() {
    Shallow obj1 {100};    // (1)
```

```

display_shallow(obj1); // (2) 傳值 -> 複製 obj1 -> 建立臨時物件 s
Shallow obj2 {obj1};   // (3) 複製 obj1 -> 建立 obj2
obj2.set_data_value(1000); // (4)
return 0;              // (5) 所有物件被銷毀
}

```

Shallow Copy：物件生命週期與記憶體視覺化

物件與記憶體分佈圖：

```

+-----+
|  Stack Memory  |
+-----+
| obj1           | ---> [Heap] --> (int: 1000) ←★只存在一份 heap 資源
| obj2           | ----- ↑
| s (in display_shallow) | ----- ↑
+-----+

```

生命週期流程（含 Shallow Copy）：

- (1) obj1 被建立：
 - 呼叫 constructor：在 heap 配置 `int(100)`
 - `obj1.data --> heap int(100)`
- (2) 呼叫 `display_shallow(obj1)`：
 - 傳值 → 呼叫 copy constructor (shallow)
 - 建立臨時物件 `s`，`s.data = obj1.data`
 - 印出值後，`s` 結束生命週期 → destructor 被呼叫 → `delete obj1.data` 🔥❌
→ ⚠️ `obj1` 和 `s` 都指向同一個 heap → 已被 `delete`
- (3) 建立 `obj2(obj1)`：
 - 呼叫 copy constructor → `obj2.data = obj1.data` (shallow copy)
 - 這時 heap 已經被 `s` 的 destructor `delete` 掉，`obj2` 和 `obj1` 指向的位址是懸掛指標 🧠
- (4) `obj2.set_data_value(1000)`：💣 Undefined Behavior !
- (5) 離開 `main()` → `obj2`, `obj1` 被銷毀：

- destructor 會 **delete** data (同一塊懸掛記憶體)
- 造成 **double delete** → 程式崩潰、未定義行為 ⚠⚠⚠

✅ Deep Copy：正確作法與生命週期視覺化

```
Shallow::Shallow(const Shallow &source) {
    data = new int;
    *data = *source.data;
    cout << "Copy constructor - deep copy" << endl;
}
```

📦 Deep Copy 的記憶體圖：

```
+-----+
|  Stack Memory  |
+-----+
| obj1           | ---> [Heap] --> (int: 100)
| obj2           | ---> [Heap] --> (int: 1000)
| s (in display_shallow) | ---> [Heap] --> (int: 100)
+-----+
```

🔄 生命週期流程 (含 Deep Copy)：

- (1) obj1 被建立：
 - obj1.data 指向 heap **int(100)**
- (2) **display_shallow**(obj1)：
 - 呼叫 copy constructor (deep copy)
 - 建立 s，s.data 是新的 heap，值為 **100**
 - s 被銷毀，釋放自己的 heap → ✓ 安全
- (3) **obj2**(obj1)：
 - 呼叫 copy constructor (deep copy)
 - obj2.data 是新 heap，複製 **100**

(4) `obj2.set_data_value(1000)` :

- 改變 `obj2.data` 指向的值為 1000，不影響 `obj1`

(5) 離開 `main()`:

- `obj2` 被銷毀，釋放自己的 heap
 - `obj1` 被銷毀，釋放自己的 heap
- ✓ 所有資源正確釋放，沒有重複 `delete`

總結表格

項目	Shallow Copy	Deep Copy
指標複製	直接複製指標	複製值並開新記憶體
物件共享資料	是（指向同一記憶體）	否（各自獨立）
安全性	✗ double delete、UB	✓ 安全、各自 delete
用途	POD 或無資源物件	有資源（heap）物件必須用

✓ 建議：若類別有指標資源，請實作 Rule of Three

```
class Shallow {
private:
    int *data;
public:
    void set_data_value(int d) { *data = d; }
    int get_data_value() { return *data; }

    Shallow(int d);           // Constructor
    Shallow(const Shallow &source); // 1.Copy Constructor
    Shallow &operator=(const Shallow &rhs); // 2.Copy Assignment
    ~Shallow();               // 3.Destructor
};
```

如果你要更安全管理資源，可以改用 `std::unique_ptr<int>`，就能自動避免 shallow copy 的陷阱（但也無法複製，因為 unique）。

✓ 小結

項目	Shallow Copy	Deep Copy
指標複製方式	複製指標本身（兩個指向同一塊記憶體）	複製指標所指的值，開新記憶體
安全性	會導致 double delete、UB	安全，各自管理自己的資源
使用場景	無資源時可用（如整數）	有資源管理責任時必要

一句話精華版：

深層複製就是：「我重新開一塊自己的記憶體，把 source 指向的內容『值』複製過來，而不是共用指標本身。」

深層複製的標準步驟：

以 name 為指標來說，假設你有這樣的 class：

```
class Person {
private:
    char* name;
public:
    Person(const char* n);        // 一般建構子
    Person(const Person& source);  // 複製建構子（重點）
    ~Person();                    // 解構子
};
```

複製建構子（Deep Copy）內容如下：



```
Person::Person(const Person& source) {
    // 1. 配置自己的記憶體空間
    name = new char[strlen(source.name) + 1];

    // 2. 把來源內容 copy 過來
    strcpy(name, source.name);
}
```

圖解記憶體配置（淺層 vs 深層）

情境： `Person p1("Alice"); Person p2 = p1;`

淺層複製（錯誤）：

p1.name →  "Alice"
p2.name →  （共用同一塊）

深層複製（正確）：

p1.name →  "Alice" p2.name →  "Alice" （不同區塊，內容一樣）

為什麼深層複製比較安全？

- 每個物件都有自己的記憶體空間。
- 彼此修改不會影響對方。
- 解構時 `delete[] name;` 不會出現 double free 的錯誤。

對照程式片段：

```
Person p1("Alice");
Person p2 = p1; // 呼叫深層複製建構子
p1.print();    // Alice
p2.print();    // Alice
strcpy(p2.name, "Bob"); // 改 p2.name 的內容
p1.print();    // Alice ✓ 沒被影響
p2.print();    // Bob   ✓ 有自己的記憶體
```

範例：Player 類別，含複製建構子

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

class Player {
private:
```

```

string name;
int health;
int xp;

public:
    // 普通建構子
    Player(string name_val, int health_val, int xp_val)
        : name{name_val}, health{health_val}, xp{xp_val} {
        cout << "Three-arg constructor called for " << name << endl;
    }

    // 複製建構子
    Player(const Player& source)
        : name{source.name}, health{source.health}, xp{source.xp} {
        cout << "Copy constructor - made copy of: " << source.name << endl;
    }

    // 顯示資訊
    void display() const {
        cout << "Player: " << name << ", Health: " << health << ", XP: " << xp << endl;
    }
};

int main() {
    Player p1{"Link", 100, 50}; // 呼叫普通建構子
    Player p2{p1};              // 呼叫複製建構子
    Player p3 = p1;             // 同樣呼叫複製建構子

    p1.display();
    p2.display();
    p3.display();

    return 0;
}

```

執行結果分析（呼叫順序）：

Three-arg constructor called for Link
 Copy constructor - made copy of: Link
 Copy constructor - made copy of: Link
 Player: Link, Health: 100, XP: 50
 Player: Link, Health: 100, XP: 50
 Player: Link, Health: 100, XP: 50

- 第一句：p1 用普通建構子建立
- 第二句：p2 使用 p1 複製 → 呼叫複製建構子
- 第三句：p3 使用 = 也會呼叫複製建構子（不是賦值運算子）

重點回顧

建構方式	呼叫哪個建構子
Player p1{"Link", 100, 50}	普通建構子
Player p2{p1}	✓ 複製建構子
Player p3 = p1	✓ 複製建構子（不是賦值）

問題：為什麼複製建構子的參數不能用傳值（by value）？

✓ 正確的寫法（避免遞迴）

```
class MyClass {
public:
    MyClass(const MyClass& other); // ✓ 傳參考，沒有複製發生
};
```

✗ 錯誤的寫法（會無限遞迴）

```
class MyClass {
public:
```

```
MyClass(MyClass other); // ❌ 傳值（會觸發複製建構子）
};
```

為什麼錯？

我們來分析這句話：

```
MyClass obj2 = obj1;
```

你以為你只會呼叫一次 `MyClass(MyClass other)`，但其實...

呼叫順序拆解（無限遞迴）

1. 你寫了這行：

```
MyClass obj2 = obj1;
```

2. 這會呼叫複製建構子：

```
MyClass(MyClass other) // 注意：參數是傳值
```

3. 為了傳值給 `other`，你必須「複製 `obj1`」👉 所以又會呼叫複製建構子來建立參數 `other`

4. 但複製建構子又是用傳值！又要複製！

5. 💣 這個「為了傳值 → 又呼叫複製建構子 → 又要傳值 → 又呼叫...」會無限遞迴

示意流程圖（遞迴發生）

你呼叫 `MyClass obj2 = obj1;`

↓ 呼叫複製建構子

```
MyClass(MyClass other)
```

↓ 要複製 `obj1` 傳進來

```
又呼叫 MyClass(MyClass other)
```



↓ 要複製 `obj1` 傳進來

```
又呼叫 MyClass(MyClass other)
```

... 無限循環 💣

那為什麼「用參考」就沒事？

```
MyClass(const MyClass& other);
```

- 傳進來的是「記憶體位置」（不是複製一份新的物件）
- 所以就不需要再呼叫複製建構子來產生這個參數
-  遞迴中斷， 程式能正常編譯與執行

範例程式（會錯的版本）

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

class MyClass {
public:
    MyClass(MyClass other) { // ❌ 傳值 → 觸發遞迴
        cout << "複製建構子\n";
    }
};

int main() {
    MyClass a; // 錯：你沒寫預設建構子會先報錯
    MyClass b = a; // 🚫 如果前面修好，這行會導致無限遞迴
    return 0;
}
```

你會看到編譯器報錯，可能像這樣（GCC）：

```
error: invalid initialization of reference of type 'MyClass&' from expression of type
'MyClass' note: passing 'MyClass' as 'this' argument discards qualifiers
```

正確版本

```
class MyClass {
public:
    MyClass() {} // 預設建構子
    MyClass(const MyClass& other) {
        cout << "複製建構子\n";
    }
};
```

```
}  
};
```