Section 14-2 Copy&Move assignment constructor

📝基本概念

Copy Assignment Operator (複製賦值運算子)

將一個已存在物件的值，從另一個同類型物件「複製」過來。
簽名通常是：

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

Move Assignment Operator (移動賦值運算子)

將資源「移動」而非複製，主要用於可被安全「轉移」所有權的資源（如動態記憶體、檔案描述符等）。
簽名通常是：

```
ClassName& operator=(ClassName&& other) noexcept;
```

❓為什麼要同時定義？

- **Copy assignment** 複製資料，適合左值物件（有名字的物件）。
- **Move assignment** 移動資源，優化性能，適合右值物件（臨時物件或使用 `std::move` 的物件）。
- 定義了 copy constructor 但不定義 copy assignment，兩者在語義和用法上是不同的，編譯器不會自動用 copy constructor 替代 copy assignment。

Copy Assignment Operator 範例

```
class MyClass {
private:
  int* data;

public:
  // Copy assignment operator
```

```
  MyClass& operator=(const MyClass& other) {
    if (this == &other)  // 自我賦值檢查
      return *this;

    delete[] data;  // 釋放自己原本的資源
    data = new int[1];  // 配置新資源
    data[0] = other.data[0];  // 複製資料

    return *this;  // 支援連鎖賦值
  }
};
```

細節

- 必須先檢查自我賦值（`this == &other`）避免刪除自己後還要讀取。
- 先清理原資源，避免記憶體洩漏。
- 深拷貝（資料一份一份複製），防止多個物件共用同一資源造成雙重刪除。

Move Assignment Operator 範例

```
class MyClass {
private:
  int* data;

public:
  // Move assignment operator
  MyClass& operator=(MyClass&& other) noexcept {
    if (this == &other)
      return *this;

    delete[] data;    // 釋放自己原本的資源
    data = other.data;  // 直接「偷取」對方資源指標
    other.data = nullptr;  // 將對方指標設為 nullptr，避免被刪除兩次

    return *this;
  }
};
```

細節

- 移動後，原物件 `other` 不再擁有該資源，因此必須將其指標清空。
- 使用 `noexcept` 是為了符合標準庫要求，確保在移動過程中不會拋出例外。
- 移動可以大幅降低效能成本，尤其是大型資源管理時。

其他注意事項

- **Rule of Three/Five**：
  如果自訂了 destructor、copy constructor 或 copy assignment operator，通常也要自訂 move constructor 和 move assignment operator，否則可能會出現未定義或效能差的行為。
- **編譯器自動產生的行為**：
  - 若未自訂，編譯器會產生淺拷貝的 copy assignment（成員逐一賦值），不適用於動態資源。
  - Move assignment 只有在 C++11 之後才會被自動產生，前提是沒有自訂 copy assignment 或 destructor。

總結示意圖（資源轉移）

```
Copy assignment:
 A.data ——copy——> B.data
(兩份獨立的資源)

Move assignment:
 A.data <—— steal — B.data
 B.data = nullptr
(單份資源權限轉移)
```

📌 跟copy/move constructor有什麼差?

> ✅ **Copy constructor 負責「用一個已存在的物件來建立新物件」**
> ✅ **Copy assignment operator 負責「把一個已存在的物件指派給另一個已存在的物件」**

這兩者語意完全不同！

🔍 差異說明

1 Copy Constructor（拷貝建構子）

觸發時機：用一個物件初始化另一個「剛出生」的物件

```
MyClass a(100);
MyClass b = a;  // ← 呼叫 Copy Constructor
```

- b 是「剛剛建立」的物件。
- 所以用 `copy constructor` 來「出生即複製」。

2 Copy Assignment Operator（拷貝指派運算子）

觸發時機：一個「已存在」的物件被賦值成另一個物件

```
MyClass a(100);
MyClass b(200);

b = a;  // ← 呼叫 Copy Assignment Operator
```

- `b` 已經存在了（記憶體與資源都分配好了）。
- 所以不能呼叫 constructor，必須用 `operator=` 來重新「指派內容」。

⚠ 如果你只寫了 copy constructor，卻沒寫 copy assignment？

那 `b = a;` 會呼叫 **編譯器自動產生的 copy assignment operator**。
這會是 **shallow copy**，導致問題（如 double delete、記憶體洩漏）。

🧪 更具體的例子

```
class MyClass {
private:
    int* data;
public:
    MyClass(int val) { data = new int(val); }
    ~MyClass() { delete data; }

    MyClass(const MyClass& other) {        // Copy Constructor
        data = new int(*other.data);
```

```
    std::cout << "Copy Constructor\n";
  }

  // MyClass& operator=(const MyClass& other); ← 假設你沒寫它
};

int main() {
  MyClass a(10);
  MyClass b(20);

  b = a; // ❗你沒寫 copy assignment → 編譯器用 shallow copy → double delete ❌
}
```

- `b = a` 會使用 **預設的 operator=**，只複製指標（shallow copy）。
- 兩個物件會指向同一個 heap。
- 結果就是：`delete` 兩次 → 崩潰。

🧭 總結表格

動作	範例	使用函式
初始化時複製（建構）	`MyClass b = a;`	Copy Constructor
指派既有物件的值	`b = a;`	Copy Assignment Operator

✅ 建議

若你的類別有資源（如 `new` 出來的指標）：

- **一定要實作 Rule of Three**（Destructor + Copy Constructor + Copy Assignment）
- 或直接用 `std::unique_ptr` / `std::shared_ptr`，避免這些問題（Rule of Zero）

🟦 C++ Copy / Move Assignment 選擇與 Fallback

```
#include <iostream>
#include <cstring>
#include "Mystring.h"
```

```
// No-args constructor
Mystring::Mystring()
  : str{nullptr} {
  str = new char[1];
  *str = '\0';
}

// Overloaded constructor
Mystring::Mystring(const char *s)
  : str {nullptr} {
    if (s==nullptr) {
      str = new char[1];
      *str = '\0';
    } else {
      str = new char[strlen(s)+1];
      strcpy(str, s);
    }
}

// Copy constructor
Mystring::Mystring(const Mystring &source)
  : str{nullptr} {
    str = new char[strlen(source.str)+ 1];
    strcpy(str, source.str);
    std::cout << "Copy constructor used" << std::endl;

}

// Move constructor
Mystring::Mystring( Mystring &&source)
  :str(source.str) {
    source.str = nullptr;
    std::cout << "Move constructor used" << std::endl;
}

// Destructor
Mystring::~Mystring() {
```

```
    if (str == nullptr) {
        std::cout << "Calling destructor for Mystring : nullptr" << std::endl;
    } else {
         std::cout << "Calling destructor for Mystring : " << str << std::endl;
    }
    delete [] str;
}

 // Copy assignment
Mystring &Mystring::operator=(const Mystring &rhs) {
    std::cout << "Using copy assignment" << std::endl;

    if (this == &rhs)
        return *this;
    delete [] str;
    str = new char[strlen(rhs.str) + 1];
    strcpy(str, rhs.str);
    return *this;
}

// Move assignment
Mystring &Mystring::operator=(Mystring &&rhs) {
    std::cout << "Using move assignment" << std::endl;
    if (this == &rhs)
        return *this;
    delete [] str;
    str = rhs.str;
    rhs.str = nullptr;
    return *this;

}
// Display method
void Mystring::display() const {
    std::cout << str << " : " << get_length() << std::endl;
}
 // getters
 int Mystring::get_length() const { return strlen(str); }
 const char *Mystring::get_str() const { return str; }
```

```
int main() {
    Mystring a{"Hello"};          // Overloaded constructor
    a = Mystring{"Hola"};         // Overloaded constructor then move assignment
    a = "Bonjour";                // Overloaded constructor then move assignment

}
```

📌 範例語句分析

這段程式中，`Mystring` 類別提供了 **copy assignment operator** 和 **move assignment operator**，兩者的選擇是由 **右值還是左值** 來決定的。

🔁 Copy Assignment 與 🚀 Move Assignment 的選擇時機：

```
Mystring &operator=(const Mystring &rhs);  // Copy assignment
Mystring &operator=(Mystring &&rhs);       // Move assignment
```

- `const Mystring &rhs`：這是給 **左值（lvalue）** 使用的。
- `Mystring &&rhs`：這是給 **右值（rvalue）** 使用的，也就是可以被搬移的臨時物件。

🔍 逐行分析 `main()` 的呼叫情況：

```
Mystring a{"Hello"};
```

- `Mystring{"Hello"}` 建立一個臨時物件，呼叫 **Overloaded constructor**
- 然後把這個臨時物件 move 到 `a`：呼叫 **Move constructor**

（這裡實際上是初始化，不是 assignment，所以不是用 `operator=`）

```
a = Mystring{"Hola"};
```

- `Mystring{"Hola"}`：建立一個臨時物件，呼叫 **Overloaded constructor**
- 因為是 **右值**，會呼叫 `operator=(Mystring&&)` → **Move assignment**

印出：

```
Using move assignment
```

```
a = "Bonjour";
```

這一行比較有趣，它發生了兩件事：

1. `"Bonjour"` 是 `const char*`，C++ 會呼叫 `Mystring(const char* s)`，建立一個臨時的 `Mystring` 物件。
2. 然後這個臨時物件是 **右值**，所以會呼叫 `operator=(Mystring&&)` → **Move assignment**

印出：

```
Using move assignment
```

✅ 總結選擇規則：

型別	呼叫哪個 assignment operator？
`a = b;`（`b` 是左值）	Copy assignment（`operator=(const Mystring&)`）
`a = Mystring{"text"}`	Move assignment（`operator=(Mystring&&)`）
`a = "text"`	建立臨時物件後 → Move assignment

如果你想觀察 copy assignment 被使用，可以加上這行測試：

```
Mystring b{"Ciao"};
a = b; // b 是左值 → 呼叫 Copy assignment
```

輸出會是：

```
Using copy assignment
```

🔁 Assignment Operator 決策流程

```
IF 有 move assignment operator：
  使用 move assignment（Mystring&&）
```

```
ELSE IF 有 copy assignment operator：
  使用 copy assignment（const Mystring&）
ELSE
  使用編譯器自動產生的 default assignment operator（前提是允許）
```

🧪 各種情況實例

✅ 情況 A：自定義 copy + move assignment（你的原始程式）

```
Mystring &operator=(const Mystring &rhs);
Mystring &operator=(Mystring &&rhs);
```

```
a = "Bonjour";
```

輸出：

```
Using move assignment
```

✅ 情況 B：只有 copy assignment（移除 move）

```
Mystring &operator=(const Mystring &rhs);
// operator=(Mystring&&) 被註解掉
```

```
a = "Bonjour";
```

輸出：

```
Copy constructor used
Using copy assignment
```

即使是右值，fallback 仍使用 copy assignment！

✅ 情況 C：都沒寫 assignment operator（使用編譯器產生的）

```
// 兩者都沒寫
// 編譯器會自動產生
```

條件：成員變數必須可以 default assign，如沒用 raw pointer。

```
a = "Bonjour"; // ✓ 成功，使用 default assignment operator
```

❌ 情況 D：明確刪除兩者

```
Mystring &operator=(const Mystring&) = delete;
Mystring &operator=(Mystring&&) = delete;
```

```
a = "Bonjour"; // ❌ 編譯失敗
```

錯誤訊息範例：

```
error: use of deleted function ‘Mystring& Mystring::operator=(const Mystring&)’
```

🧠 Rule of Five 小提醒

當你定義以下其中一個，建議全都要定義：

- Destructor
- Copy constructor
- Copy assignment
- Move constructor
- Move assignment

否則會出現：

> 編譯器「不自動生成 move」或「不自動生成 copy」，造成 fallback 行為與預期不同。

☒ Assignment 選擇流程圖

```
a = rvalue
  ↓
```

Is move assignment available?

↓ yes ↓ no

use move assignment Is copy assignment available?

↓ yes ↓ no

use copy assignment use compiler default