

## Section 13-5 Move Constructor

### Move Constructor（移動建構子）介紹

#### 定義

Move Constructor 是 C++11 引入的一種建構子，主要目的是將資源的所有權從一個物件「搬移」到另一個物件，而非複製它們，以提高效能。

語法形式如下：

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

參數是右值參考（rvalue reference），也就是用 `&&` 表示的參考型態。

#### 和 Copy Constructor 有何不同？

特性	Copy Constructor 複製建構子	Move Constructor 移動建構子
接收參數型態	<code>const ClassName&amp;</code>	<code>ClassName&amp;&amp;</code>
資源處理方式	複製（耗費資源）	搬移（不複製內容）
用途	對 lvalue 物件進行複製	對 rvalue（臨時物件）進行搬移

#### 什麼是「右值」與「右值參考」？

- 右值（rvalue）：臨時物件，例如 `std::string("Hello")`
- 右值參考（T&&）：可以綁定右值的參考，允許你「偷走」其資源

#### 範例程式碼

```
#include <iostream>
#include <cstring>
```

```

class MyString {
private:
    char* data;

public:
    // Constructor
    MyString(const char* str) {
        data = new char[strlen(str) + 1];
        strcpy(data, str);
        std::cout << "Constructed: " << data << "\n";
    }

    // Copy Constructor
    MyString(const MyString& other) {
        data = new char[strlen(other.data) + 1];
        strcpy(data, other.data);
        std::cout << "Copied: " << data << "\n";
    }

    // Move Constructor
    MyString(MyString&& other) noexcept {
        data = other.data;    // 搬移指標
        other.data = nullptr; // 避免 double delete
        std::cout << "Moved\n";
    }

    // Destructor
    ~MyString() {
        if (data) {
            std::cout << "Destroyed: " << data << "\n";
            delete[] data;
        }
    }
};

int main() {
    MyString s1("Hello");
    MyString s2 = std::move(s1); // 會呼叫 move constructor

```

```
return 0;
}
```

## 執行輸出：

```
Constructed: Hello
Moved
Destroyed: Hello
```

## 為什麼需要 Move Constructor ？

1. 效率提升：避免不必要的資源複製（例如 string、vector 的資料）
2. 優化 STL 操作：例如 `std::vector::push_back(std::move(obj))` 使用搬移而非複製
3. 支援右值傳遞與返回值最佳化（RVO）

## 注意事項

- 若定義了 Move Constructor，建議同時定義 Move Assignment Operator
- 若有資源管理，Move Constructor 必須轉移資源所有權，並清空來源物件的資源
- 若類別中含有原始指標或資源，建議實作 Rule of Five（五大函數）

## l-value reference 和 r-value reference 詳解

### 名詞定義

名稱	符號	可綁定對象	範例
l-value reference	T&	左值（具名、有位置）	<code>int&amp; a = x;</code>
r-value reference	T&&	右值（臨時、無名）	<code>int&amp;&amp; b = 5;</code>

### l-value（左值）

- 定義：能出現在賦值運算子左邊的值，有名稱、記憶體地址
- 特徵：
  - 可多次使用

- 可取地址（用 `&` 取址符）

```
int x = 10;
int& ref = x; // OK, x 是 l-value
```

## ✓ 常見例子：

```
int a = 5; // a 是 l-value
int b = a; // a 是 l-value, 5 是 r-value
int* p = &a; // &a 是 l-value 的地址
```

## r-value（右值）

- 定義：無法取地址、無名稱的「臨時值」
- 特徵：
  - 僅存在於一行語句中
  - 無法被修改（預設）

```
int&& ref = 5; // OK, 5 是 r-value
```

## ✓ 常見例子：

```
int x = 10;
int y = x + 3; // (x + 3) 是 r-value
int&& r = x + 3; // r 綁定 r-value
```

## 實驗範例：看哪些能編譯

```
int a = 10;
int& lref = a; // ✓ OK, a 是 l-value
int& lref2 = 5; // ✗ 錯誤, 5 是 r-value, 不能綁定到 l-value reference

int&& rref = 5; // ✓ OK, 5 是 r-value
int&& rref2 = a; // ✗ 錯誤, a 是 l-value, 不能綁定到 r-value reference

int&& rref3 = std::move(a); // ✓ OK, std::move 將 a 轉成 r-value
```

## 用 ASCII 圖形幫助理解：

// 假設變數 a 是一個左值，有一個地址和內容

```
+-----+ +-----+
| a | ---> | value | <--- 有名稱、有記憶體位置
+-----+ +-----+
```

// 右值像是 5 或 x+3，是臨時的，不可取址

5 或 (x+3) <--- 無名稱、不佔有記憶體位址，可直接搬移

## 實務用途

### l-value reference ( T& )：

- 傳遞已有物件給函數（避免複製）

```
void print(const std::string& str); // 接收 l-value
```

### r-value reference ( T&& )：

- 搬移臨時物件資源，提高效率

```
void setName(std::string&& name) { this->name = std::move(name); // 搬移資源 }
```

## const correctness 是什麼？

簡單來說，就是盡可能地告訴編譯器：哪些東西是不能被修改的。  
在 C++ 類別中，它可以出現在三個主要地方：

- 成員函式後面的 `const`
- 函式參數的 `const` 修飾
- 資料成員前的 `const`（或 `mutable`）

## 一、const 成員函式

### 語法：

ReturnType **functionName**(...) **const**;

## ✓ 用途：

表示此函式**不會修改任何資料成員**（非 `mutable`），也不能呼叫任何非 `const` 的成員函式。

## 📌 範例：

```
class MyClass {
private:
    int value;

public:
    int getValue() const { // const 成員函式
        return value;
    }

    void setValue(int v) {
        value = v;
    }
};
```

```
const MyClass obj; // obj 是 const 的
obj.getValue();    // ✓ 可以呼叫 const 函式
obj.setValue(5);   // ✗ 錯誤，不能呼叫非 const 函式
```

## 📌 二、參數加上 `const`（避免不必要的複製）

## ✓ 語法：

```
void func(const std::string& str);
```

## ✓ 用途：

- 使用參考（&）避免複製
- 使用 `const` 確保參數在函式中**不會被修改**

## 範例：

```
void printMessage(const std::string& msg) {
    std::cout << msg << std::endl;
}
```

```
std::string s = "Hello";
printMessage(s); //  可接受 l-value
printMessage("Hi"); //  可接受 r-value
```

## 三、const 資料成員 & mutable

### const 資料成員


```
class Example {
private:
    const int id; // 一旦建構後就不能變

public:
    Example(int i) : id(i) {}
};
```

- 只能透過建構式初始化
- 無法在其他函式內被修改

### mutable 修飾（例外允許 const 成員函式修改的變數）

```
class Logger {
private:
    mutable int accessCount;

public:
    void log() const {
        ++accessCount; //  合法，因為 mutable
    }
};
```


## 總結表格

用法	說明
const 成員函式	保證不會修改任何資料成員
const 參數（傳值、參考）	避免意外改動參數，或避免複製
const 資料成員	一旦建構完成後無法更改
mutable 修飾資料成員	允許 const 成員函式中修改此成員

## 範例總覽

```
class Book {
private:
    std::string title;
    mutable int accessCount = 0;

public:
    Book(const std::string& t) : title(t) {}

    std::string getTitle() const {
        ++accessCount; //  因為 accessCount 是 mutable
        return title;
    }

    void setTitle(const std::string& t) {
        title = t;
    }
};
```

## const & 非-const 傳參方式選擇原則總整理

我們可以依據「參數的型態」與「是否需要修改」來選擇最適合的傳遞方式：

## 四種常見傳參方式



傳遞方式	會不會複製？	可以修改參數？	備註建議使用時機
T (傳值)	✅ 是	✅ 可以	小型內建型別如 <code>int</code> , <code>char</code> , <code>bool</code>
T& (reference)	❌ 否	✅ 可以	必須修改實參值
const T& (const 參考)	❌ 否	❌ 不行	推薦用於大型物件只讀用途
T&& (右值參考)	❌ 否	✅ 可以	用於支援 move semantics (見下節)

## 🧠 什麼情況用 `const T&` ？

### ✅ 範例：接收大型物件，但不想複製

```
void printName(const std::string& name) {
    std::cout << name << std::endl;
}
```

- 避免複製 `string` (可能很大)
- 又能保證 `name` 不會在函式中被修改

## 🔧 小型型別 (例如 `int`、`double`) 就用傳值 T

```
int square(int x) {
    return x * x;
}
```

- 傳值簡單、效能沒差
- 用 `const int&` 反而多此一舉

## 🔧 什麼情況用 T& ？

當你需要 直接修改呼叫者的資料：

```
void addOne(int& x) {
    x += 1;
}
```

呼叫端：

```
int a = 5;
addOne(a); // a 變成 6
```

## ⚠ const T 傳值 vs const T& 傳參？

- `const T` 是個副本，但你保證不會改它
- `const T&` 是避免複製的優化手段

```
void funcA(const std::string s); // 會複製
void funcB(const std::string& s); // 不會複製 ✅ 推薦
```

## 小結：參數傳遞選擇表

類型	需要改值？	傳遞方式	範例
小型資料 (如 int)	否	傳值	<code>void print(int x)</code>
小型資料	是	<code>int&amp;</code>	<code>void update(int&amp; x)</code>
大型物件 (如 string、vector)	否	<code>const T&amp;</code>	<code>void print(const std::string&amp;)</code>
大型物件	是	<code>T&amp;</code>	<code>void modify(std::vector&lt;int&gt;&amp;)</code>
支援移動語意	是	<code>T&amp;&amp;</code>	<code>void setName(std::string&amp;&amp;)</code>