

Section 5 C++程式的結構

定義：什麼是 Identifier ？

在 C++ 裡，`identifier`（識別字）是用來表示變數、函式、類別、物件、參數、常數、陣列、命名空間等的名稱。

```
int age = 21;
double hourly_rate = 350.5;
std::string name = "Jorson";
```

命名規則（C++ 語法規定）

一個合法的識別字要遵守這些規則：

1. 只能用英文字母（A-Z、a-z）、數字（0-9）、底線 _
2. 第一個字不能是數字
3. 不能跟關鍵字（例如 `int`，`for`，`return`）一樣

合法的 identifier：

```
myVariable _age rate1 MAX_VALUE
```

不合法的 identifier：

```
1age //  開頭不能是數字
double //  是關鍵字
my-variable //  有非法字元 (-)
```

運算子(Operator)

什麼是「運算子重載」？

重載的意思就是「讓同一個東西在不同情況下做不同的事情」。

在 C++ 裡面，你可以把某些運算子重新定義，讓它們能夠套用在你自訂的資料型別

(例如你自己寫的類別) 上。

舉個例子：<< 本來是什麼意思？

在 C 語言中，<< 是「左位移 (bit shift left)」：

```
int x = 5; // 二進位：00000101
int y = x << 1; // 結果是 10 (00001010)
```

這是它原本的「位元操作功能」。

那為什麼可以寫成：

```
std::cout << "Hello";
```

這是因為在 C++ 裡，std::ostream (例如 std::cout) 的類別把 << 運算子重載成「輸出字串」的功能。

<< 輸出是怎麼寫出來的？

在 ostream 裡有定義這樣的東西 (簡化版)：

```
std::ostream& operator << (std::ostream& out, const char* str) {
    // 實作方式是把 str 的內容輸出到 out (也就是 cout)
    out.write(str, strlen(str));
    return out; }
```

所以當你寫：

```
std::cout << "Hello";
```

這會被 C++ 編譯器解讀成：

```
operator<<(std::cout, "Hello");
```

什麼是Preprocessor Directives?

預處理器指令是我們程式碼中包含的行，它們不是程式語句，而是預處理器的指令。這些行前面總是有一個井號 (#)。

這些指令的前面都會有一個 # 符號，並且在程式編譯前就會被執行，用來：

- 包含檔案（例如 .h 標頭檔）
- 定義常數或巨集
- 控制條件編譯（Conditional Compilation）

常見的 Preprocessor Directives：

指令	說明
#include	將標頭檔案內容插入進來
#define	定義巨集（macro），可用來替換常數或簡化程式碼片段
#undef	取消先前定義的巨集
#ifdef	如果有定義某個巨集，則編譯某段程式
#ifndef	如果沒有定義某個巨集，則編譯某段程式
#if, #elif, #else, #endif	條件式編譯，用於更靈活的控制程式是否要被編譯
#pragma	編譯器特定指令（非標準）

#ifdef 範例 (if defined)

說明：

#ifdef MACRO 的意思是：如果這個 MACRO 有被定義過，就編譯下面的程式碼。

範例程式碼：

```
#include <stdio.h>

#define DEBUG_MODE // 註解掉這行看看會怎樣

int main() {
    #ifdef DEBUG_MODE
        printf("Debugging mode is ON\n");
    #endif
}
```

```
printf("Program is running...\n");
return 0;
}
```

💡 結果：

如果有 `#define DEBUG_MODE`，就會輸出：

```
Debugging mode is ON
Program is running...
```

如果你把 `#define DEBUG_MODE` 註解掉，則只會看到：

```
Program is running...
```

✂️ #ifndef 範例 (if not defined)

📌 說明：

`#ifndef MACRO` 的意思是：如果這個 `MACRO` 沒有被定義過，就編譯下面的程式碼。

這個通常用來避免重複包含標頭檔 (Include Guard)。

✅ 範例程式碼（常見於標頭檔）：

```
// file: myheader.h
#ifndef MYHEADER_H //沒有define MYHEADER_H (myheader.h第一次被引用)
#define MYHEADER_H //則define這個巨集

void hello();

#endif
```

```
// file: main.c
#include <stdio.h>
#include "myheader.h"
```

```
void hello() {
    printf("Hello!\n");
}

int main() {
    hello();
    return 0;
}
```

為什麼要這樣寫？

因為一個 `.h` 檔案可能會被重複包含（尤其是多個檔案都包含了同一個標頭），使用 `#ifndef` 可以確保這個標頭內容只會被編譯一次，避免重複定義錯誤。

Include Guard

在大型專案中，一個標頭檔（`.h`）可能會被多個 `.c` 或 `.h` 檔案包含：

```
// a.c
#include "myheader.h"
#include "b.h" // 假設 b.h 裡面也 include "myheader.h"
```

如果你不小心重複定義了函式或變數，像這樣：

```
// myheader.h
void hello();
```

C 語言會報錯說：「函式重複定義」，因為這個檔案的內容被插入了兩次以上。

解法：使用 `#ifndef` 包起來

我們用 `#ifndef` 加上一個唯一的標記（通常使用大寫加底線，如 `MYHEADER_H`）：

```
#ifndef MYHEADER_H // 如果尚未定義 MYHEADER_H
#define MYHEADER_H // 現在定義它

// 標頭檔內容
void hello();
```

```
#endif // 結束條件編譯
```

✅ 編譯流程解析：

假設有兩個檔案都 `#include "myheader.h"`，第一次編譯時：

1. 發現 `MYHEADER_H` 沒有被定義 → 通過 `#ifndef`
2. 定義 `MYHEADER_H`
3. 編譯 `void hello();`

第二次再碰到 `#include "myheader.h"` 時：

1. 發現 `MYHEADER_H` 已經被定義 → 跳過 `#ifndef` 包起來的所有內容
2. 所以不會再次宣告 `void hello();`

🎯 `#ifndef MYHEADER_H` 的作用是？

它的意思是：

如果 `MYHEADER_H` 還沒被定義過，就執行下面的程式碼。

也就是說：

這行會判斷 "這個巨集 `MYHEADER_H` 有沒有被 `#define` 過？"

📌 重點：這裡的 `MYHEADER_H` 不是函式也不是變數，而是一個巨集名稱（macro）。

🌟 巨集是什麼？

巨集是你用 `#define` 定義的東西，例如：

```
#define MYHEADER_H
```

這行的意思是「定義一個名字叫做 `MYHEADER_H` 的巨集」，它不用指定數值，只是個旗子（flag），用來代表某件事情已經發生。

💡 預處理器的運作流程圖

```
main.c
|
```

```

|—— #include "myheader.h" ← 第一次包含
|
|   |
|   |—— 檢查 MYHEADER_H 有沒有被定義？
|   |
|   |   |
|   |   |—— ✗ 沒有 → 通過 #ifndef
|   |   |—— 定義 MYHEADER_H (#define MYHEADER_H)
|   |   |—— 加入 hello() 函式宣告
|
|—— #include "myheader.h" ← 第二次包含
|
|   |
|   |—— 檢查 MYHEADER_H 有沒有被定義？
|   |
|   |   |
|   |   |—— ✓ 已經有了！ → 跳過整個區塊，不做事
|
|—— 編譯成功，不會出現重複定義

```

當 #ifndef 沒通過時（=巨集已經被定義過）

```

#ifndef MYHEADER_H ← ✗ 不成立（因為 MYHEADER_H 已經定義了）
#define MYHEADER_H ← ⛔ 這行會被「跳過」
void hello();      ← ⛔ 這行也「跳過」
#endif            ← ✓ 直接跳到這裡

```

 所以：

- 整段區塊（從 #ifndef 到 #endif）都會被忽略不處理
- 中間的 函式宣告、巨集定義、結構定義等等，都不會出現在編譯器眼中

函式的宣告/定義

```

// 📁 myheader.h
#ifndef MYHEADER_H
#define MYHEADER_H

void hello(); // ← 這是「宣告」

```

#endif

```
// 📁 main.c
#include <stdio.h>
#include "myheader.h"

void hello() {           // ← 這是「定義」
    printf("Hello from header!\n");
}

int main() {
    hello();
    return 0;
}
```

問：

myheader.h 裡有 `void hello();`，main.c 又寫了 `void hello() {...}`
為什麼不會造成重複定義？😞

✨ 重點觀念：

➤ 「函式宣告 (Declaration)」≠ 「函式定義 (Definition)」

行為	例子	意思
宣告	<code>void hello();</code>	告訴編譯器：「這個函式之後會出現」
定義	<code>void hello() { ... }</code>	真的寫出完整內容

🔄 你可以宣告很多次，但只能定義一次！

✅ 所以這段程式碼沒問題的原因是：

- myheader.h 裡面只是「宣告」：我會有個叫 `hello()` 的函式
- main.c 裡面「定義」了這個函式
- 一個檔案裡「一個定義、多個宣告」是完全合法的！

🧠 額外提醒一個好習慣：

通常 `.h` 檔只會放「宣告」，真正的「定義」會寫在 `.c` 檔，例如：

```
// 📁 myheader.h  
void hello(); // 宣告
```

```
// 📁 myheader.c  
void hello() {  
    printf("Hello!\n");  
}
```

然後在 `main.c`

```
#include "myheader.h"  
  
int main() {  
    hello();  
}
```

Namespace

在 C++ 中，`namespace` 是一種用來將變數、函數、類別等標識符分組的機制，以避免名稱衝突。特別是在大型程式或使用第三方函式庫時，`namespace` 可以幫助你區分不同區域或模組中的名稱，確保不同模組或庫中的相同名稱不會互相干擾。

主要功能：

1. **避免名稱衝突**：不同模組或函式庫中可能會使用相同的變數或函數名稱。使用 `namespace` 可以將這些名稱分開，避免衝突。
2. **組織代碼**：可以將相關的函數、變數、類別等組織在一起，讓代碼結構更加清晰。

範例:

```
#include <iostream>  
  
namespace Math {  
    int add(int a, int b) {
```

```

    return a + b;
}
}

namespace String {
    void print(const std::string& str) {
        std::cout << str << std::endl;
    }
}

int main() {
    int result = Math::add(3, 4);
    String::print("Result: " + std::to_string(result));
    return 0;
}

```

在這個範例中，`Math` 和 `String` 是兩個命名空間（`namespace`），其中分別定義了不同的函數。這樣即便有其他命名空間也有 `add` 或 `print` 函數，也不會造成衝突。

namespace 的使用方法：

- **基本定義：** `namespace` 通常是使用關鍵字 `namespace` 來定義。
- **全域使用：** 如果你希望在不使用 `namespace` 前綴的情況下使用其中的元素，可以使用 `using` 關鍵字。

`namespace` 並不一定只包含一個 `library`。事實上，一個 `namespace` 可以包含多個函數、變數、類別、結構、類型定義等，並且它們可以來自不同的庫或模組。

進一步說明：

- **多個元素：** 你可以將多個相關的函數、類別或變數放入同一個 `namespace` 中，即使這些元素來自不同的庫。這樣有助於將相關功能組織在一起，讓代碼結構更清晰。
- **多個 namespace 和庫的關係：** 不同的庫可以使用相同名稱的 `namespace`，並將不同的功能放在同一個 `namespace` 下。這樣不會導致名稱衝突，因為它們是在不同的 `namespace` 中。

範例：

假設有兩個不同的庫，分別提供數學和字串處理的功能，但它們都使用相同的 namespace 名稱 MyLib 。

```
// 第一個庫：數學相關功能
namespace MyLib {
    int add(int a, int b) {
        return a + b;
    }
}

// 第二個庫：字串相關功能
namespace MyLib {
    void print(const std::string& str) {
        std::cout << str << std::endl;
    }
}

int main() {
    int result = MyLib::add(3, 4); // 使用 MyLib 中的 add 函數
    MyLib::print("Result: " + std::to_string(result)); // 使用 MyLib 中的 print 函數
    return 0;
}
```

在這個範例中，我們看到 MyLib 被用來包含數學和字串處理的功能。儘管這些功能來自不同的庫，但它們都被放在同一個 namespace 下，這樣可以更清楚地表達它們是來自相同的邏輯組織。

namespace 可以跨多個檔案

在 C++ 中，namespace 可以跨越多個檔案。你可以在不同的檔案中定義相同的 namespace，並在這些檔案中將不同的元素添加進去：

```
// file1.cpp
namespace MyLib {
    int add(int a, int b) {
        return a + b;
    }
}
```

```
// file2.cpp
namespace MyLib {
    void print(const std::string& str) {
        std::cout << str << std::endl;
    }
}
```

這樣，即使在不同的檔案中定義 namespace ，它們依然會被視為同一個 namespace ，並且可以在其他地方共同使用。