

Section 10 Function

Function

一、什麼是Function（函式）

定義：

C++ 的函式是一段可以重複使用的程式碼區塊，用來執行某些特定任務。它通常包含：

- 函式名稱
- 回傳型別
- 參數列表
- 主體（也就是程式碼區塊）

語法範例：

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

// 定義一個回傳 int、名稱為 add、帶有兩個 int 參數的函式
int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(3, 5); // 呼叫函式
    cout << "結果是: " << result << endl;
    return 0;
}
```

二、Function Prototype（函式原型）

定義：

函式原型就是函式的「宣告」，告訴編譯器該函式的名稱、參數類型與回傳型別，但不包含實作細節（主體）。

作用：

- 允許你在 main 函式或其他函式前呼叫函式（即使實作在後面）
- 幫助編譯器做「型別檢查」

語法範例：

```
#include <iostream>
using namespace std; // 函式原型（只有宣告，沒有主體）
int add(int a, int b);
int main() {
    int result = add(10, 20); // 即使函式定義在後面也能使用
    cout << "加總結果: " << result << endl;
    return 0;
} // 函式定義
int add(int a, int b) {
    return a + b;
}
```

小整理：

名稱	位置與功能
函式定義（Definition）	包含完整實作（主體）
函式原型（Prototype）	告訴編譯器這個函式的名稱、參數與回傳型別
函式呼叫（Call）	執行該函式的動作，並傳入所需的參數

Default Argument Value

一、什麼是 Default Argument Value（預設參數值）？

預設參數值 指的是：當呼叫函式時 沒有提供某些參數時，系統會使用預先設定的預設值來代替。

✓ 語法格式：

```
return_type function_name(type1 param1 = default1, type2 param2 = default2, ...);
```

◆ 預設值只能從 右邊往左設定，不能設定中間的某個參數而不設定右邊的參數。

✖ 範例 1：簡單的函式有預設參數

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

// 函式原型含預設參數
void greet(string name = "使用者") {
    cout << "哈囉，" << name << "！" << endl;
}

int main() {
    greet();          // 沒給參數，使用預設值
    greet("小明");    // 有給參數，使用提供的值
    return 0;
}
```

◆ 輸出：

```
哈囉，使用者！
哈囉，小明！
```

✖ 範例 2：多個參數中使用預設值

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

// 只有第二個參數有預設值
int power(int base, int exponent = 2) {
    int result = 1;
```

```

for(int i = 0; i < exponent; ++i) {
    result *= base;
}
return result;
}

int main() {
    cout << power(3) << endl; // 使用預設 exponent = 2，結果是 3^2 = 9
    cout << power(2, 5) << endl; // 使用指定 exponent = 5，結果是 2^5 = 32
    return 0;
}

```

⚠ 注意事項：

1. 預設值只能定義一次：
預設值只能出現在函式原型或定義其中之一，不能兩邊都寫。

```

// 正確：只在原型定義預設值
int add(int a, int b = 10);

int add(int a, int b) {
    return a + b;
}

```

2. 從右至左設定：
你不能只為左邊的參數設定預設值而跳過右邊的參數。

```

// 錯誤 ❌
void foo(int x = 1, int y); // 錯在 y 沒有預設值但 x 有

// 正確 ✅
void foo(int x, int y = 2);

```

🧠 小整理：

特性	說明
預設值定義位置	可以在函式宣告（原型）或定義裡，但只能擇一

特性	說明
從右往左給預設值	否則會導致編譯錯誤
可提升函式彈性	呼叫時不必提供所有參數，讓使用者使用更方便

Function Overload

一、什麼是 Function Overloading（函式多載）？

函式多載 是指：多個同名函式，但它們的參數數量或型別不同，編譯器會根據你呼叫時的方式，自動選擇對應的版本。

好處：

- 增加程式可讀性
- 讓相似功能的函式使用相同名稱，減少命名負擔
- 使用者不需記太多函式名稱，只需改變參數就能完成不同工作

二、基本語法與範例

```
#include <iostream>
using namespace std; // 函式多載：同名函式，但參數不同
int add(int a, int b) {
    return a + b;
}
double add(double a, double b) {
    return a + b;
}
int add(int a, int b, int c) {
    return a + b + c;
}
int main() {
    cout << add(3, 4) << endl;    // 呼叫第一個：int + int
    cout << add(2.5, 3.7) << endl; // 呼叫第二個：double + double
    cout << add(1, 2, 3) << endl;  // 呼叫第三個：三個 int
    return 0;
}
```

◆ 輸出：

7 6.2 6

三、哪些差異可以「多載」？

差異類型	可多載？	說明
函式參數數量	✓	可使用不同數量的參數
函式參數型別	✓	即使參數數量相同，只要型別不同也可
函式回傳型別	✗	單靠回傳型別不同無法多載（編譯器會報錯）

錯誤示範（僅改變回傳型別）：

```
// ✗ 不合法，因為參數一樣但只改了回傳型別
int getValue(int x) {
    return x;
}
double getValue(int x) {
    return (double)x;
} // 錯誤：無法多載
```

小整理

特性	說明
函式名稱相同	是多載的必要條件
參數數量或型別不同	才能成功多載
回傳型別不同不算多載	編譯器無法僅根據回傳型別來區分函式
與預設參數可以搭配使用	但需注意避免造成模糊不清的呼叫（ambiguous）

Pass By Value, Pass By Reference, Pass By Address



一、三種傳遞方式總覽

名稱	中文名稱	傳遞內容	原始變數會改變嗎？
Pass by Value	傳值呼叫	傳遞「變數的值副本」	✗ 不會
Pass by Reference	傳參考呼叫	傳遞「變數的別名（參考）」	✓ 會
Pass by Address	傳地址呼叫	傳遞「變數的位址」	✓ 會

✖ 二、Pass by Value（傳值呼叫）

📌 特點：

- 將變數的值複製一份
- 在函式內修改的是「副本」，不會影響原始變數

🧠 圖解（變數獨立）：

main: [a = 5] —→ 傳值 → [x = 5] (函式內的變數)
 ↑ 修改 x 不會影響 a

🔍 範例：

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

void modify(int x) {
    x = 100;
}

int main() {
    int a = 5;
    modify(a);
}
```

```
cout << "a = " << a << endl; // 輸出 a = 5
}
```

三、Pass by Reference（傳參考呼叫）

特點：

- 將變數的參考（別名）傳進函式
- 函式內外操作的是同一塊記憶體

圖解（變數共用）：

main: [a = 5] ←—— 參考 → [x = a]
 ↑ 修改 x 就是修改 a

範例：

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

void modify(int& x) { // 注意 & 是參考
    x = 100;
}

int main() {
    int a = 5;
    modify(a);
    cout << "a = " << a << endl; // 輸出 a = 100
}
```

四、Pass by Address（傳地址呼叫）

特點：

- 傳遞的是變數的位址（記憶體位置）

- 使用指標 (pointer) 來間接修改值
- 與 C 語言方式一致

圖解（操作位址）：

main: [a = 5] —→ 傳地址 → [ptr 指向 a]
 ↑ *ptr 修改的是 a

範例：

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

void modify(int* ptr) {
    *ptr = 100; // 解參考後修改值
}

int main() {
    int a = 5;
    modify(&a); // 傳入 a 的地址
    cout << "a = " << a << endl; // 輸出 a = 100
}
```

五、比較整理表格

傳遞方式	使用方式	可否改變原始變數	常見用途
Pass by Value	void f(int x)	✗ 不會	傳遞數值但不希望原變數被改動
Pass by Reference	void f(int& x)	✓ 會	需修改外部變數（高階 C++ 建議）
Pass by Address	void f(int* x)	✓ 會	與 C 語言互通、或處理陣列

六、小提醒與實務應用

1. 若不想讓變數被改到 → 傳值
2. 希望函式內能修改變數 → 傳參考或傳地址
3. 使用傳地址時要注意指標合法性與 nullptr 問題
4. 傳參考是 C++ 特有功能，較現代、安全，建議優先使用

指標 (Pointer) vs 參考 (Reference) 的差異

特性	參考 (T&)	指標 (T*)
是否可以為 null	❌ 不行 (參考一出生就得指向某物)	✅ 可以為 nullptr
是否可以重新指向	❌ 不行，一旦設定不能再變	✅ 可以改變指向的對象
存取語法	像變數一樣用 x	需用 *x 解參考、x->member 存取
需要額外符號	✅ 不需 * 或 & 存取	❗ 需解參考 * 或成員存取箭頭 ->
常用場景	語法簡潔、安全性高	更靈活 (動態記憶體/可為 null 等)

範例比較：使用 string& vs string*

✅ 使用參考 (Reference)

```
void modify(std::string& ref) {
    ref += " world";
}
```

呼叫時像這樣：

```
std::string s = "hello";
modify(s); // 直接改變 s 本身
```

- ✅ 語法簡潔，不用解參考，也不能是 nullptr

使用指標 (Pointer)

```
void modify(std::string* ptr) {
    if (ptr) {
        *ptr += " world";
    }
}
```

呼叫時：

```
std::string s = "hello";
modify(&s); // 傳址，需要取址符號 &
```

✅ 可以處理空指標

⚠ 語法較複雜，需要用 `*ptr` 和 `->` 存取

📌 那為什麼 `getData()` 傳 `string&` 而不是 `string*` ？

因為：

- 語法簡潔 → 呼叫者只要寫 `demo.getData()` 就可以拿到 `string`
- 不需要判斷 `null` → 類別內的成員一定存在
- 參考更「安全」：物件一旦回傳出去，不用 worry 指標失效或錯誤操作

👁 形象比喻：參考 vs 指標

類型	比喻
指標	有地址的小紙條（你可以丟掉、改寫、空指）
參考	幫物品取了個暱稱（你無法改掉這個暱稱指向）

✅ 小結：你該怎麼選？

- 👉 **Class 成員函式常用 `&`**，因為內部成員一定存在，不需 `null` 檢查
- 👉 若有「不一定存在」的可能，或需動態管理，**才用指標**