C++筆記

4B4G0100 傅正榮

第一週：C++ 的起源與語言核心哲學

了解 C++ 如何從 C 語言進化，探討其核心設計原則（如 Zero-overhead），以及從經典標準到現代 C++ 的發展里程碑 。

1-1. C++ 的歷史背景

- 創始人：Bjarne Stroustrup 。
- 誕生時間：1979 年開始開發，初名 "C with Classes" 。
- 更名寓意：1983 年正式更名，++ 是 C 的遞增運算子，暗示其為進化版

1-2. 從 C 到 C++ 主要的演變

- 繼承與相容：C++ 是 C 語言的超集 (Superset)，大部分 C 程式碼可在其編譯器執行 。
- 引入物件導向 **(OOP)**：增加了類別 (Class)、物件 (Object)、繼承與多型
- 泛型程式設計：引入範本 (Template)，為標準函式庫 (STL) 的基礎 。

1-3. C++ 的標準化進程

- **C++98 / C++03**：第一個正式標準，奠定基礎 。
- **C++11 (**關鍵轉折**)**：現代 C++ 的起點，引入 auto、Lambda、智能指標
- **C++20**：引入 Concepts (概念)、Modules (模組) 與 Coroutines (協程) 。

1-4. C++ 的核心設計

- **"Zero-overhead"** 原則：沒用到的功能，不需付出效能代價 。
- 硬體控制：保留強大的低階記憶體控制能力與直接存取硬體的能力 。
- 強類型檢查：編譯時期發現更多錯誤，比 C 更嚴格 。

第二週：資料型態、初始化與常數安全

深入探討變數在記憶體中的儲存大小，學習現代 C++ 推薦的初始化手段，並區分不同類型的常數以優化效能。

2-1. 基本資料型態深度解析

- **int (整數)**：通常佔 4 Bytes。
- **double (雙精準度浮點數)**：佔 8 Bytes，提供更高的小數精度。
- **char (字元)**：佔 1 Byte，儲存 ASCII 碼。
- **bool (布林值)**：佔 1 Byte，僅有 true 或 false。
- **auto (型別推導)**：編譯器根據初始值自動推導型別，宣告時必須初始化

2-2. 變數宣告與初始化方式

- 拷貝初始化：int a = 1;。
- 直接初始化：double b(2.0);。
- 列表初始化：int c{5};，具備一致性且能防止「窄化轉換」。

2-3. 核心運算子與修飾字

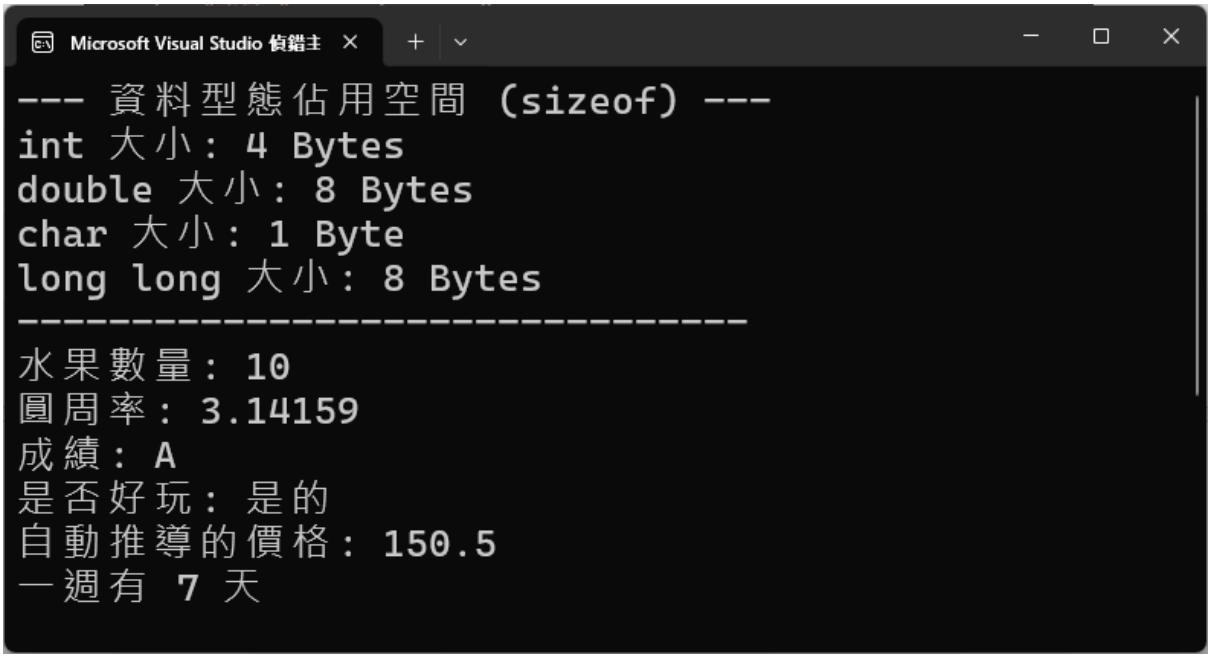
- **sizeof()** 運算子：查詢型態或變數在記憶體中所佔的位元組 (Byte) 數。
- 型別修飾字：unsigned 處理正數；short / long 調整記憶體佔用長度。

2-4. 常數與唯讀變數

- **const**：唯讀變數，賦值後不可修改，常用於函式參數。
- **constexpr**：編譯時期常數，值在編譯時算好，不佔執行時間，效能最優

程式演練

```cpp
#include <iostream>
#include <iomanip> // 用於控制輸出格式
using namespace std;

int main() {
    // --- 2-1 & 2-2. 資料型態與初始化 ---
    int appleCount = 10;              // 拷貝初始化
    double pi(3.14159265);            // 直接初始化
    char grade{ 'A' };                // 列表初始化 (推薦！安全且一致)
    bool isCPlusPlusFun = true;       // 布林值
    auto price = 150.5;               // 自動推導為 double

    // --- 2-3. 核心運算子與修飾字 ---
    unsigned int positiveOnly = 500;  // 僅能儲存正數
    long long bigNumber = 9999999999;  // 較大的整數空間

    cout << "--- 資料型態佔用空間 (sizeof) ---" << endl;
    cout << "int 大小: " << sizeof(int) << " Bytes" << endl;
    cout << "double 大小: " << sizeof(double) << " Bytes" << endl;
    cout << "char 大小: " << sizeof(char) << " Byte" << endl;
    cout << "long long 大小: " << sizeof(bigNumber) << " Bytes" << endl;
    cout << "------------------------------" << endl;

    // --- 2-4. 常數與唯讀變數 ---
    const int MAX_SCORE = 100;        // 執行期常數
    constexpr int DAYS_IN_WEEK = 7;   // 編譯期常數 (效能最優)
    // 嘗試修改常數會報錯，例如：MAX_SCORE = 110; (這行會導致編譯失敗)

    // --- 輸出結果展示 ---
    cout << "水果數量: " << appleCount << endl;
    cout << "圓周率: " << pi << endl;
    cout << "成績: " << grade << endl;
    cout << "是否好玩: " << (isCPlusPlusFun ? "是的" : "不是") << endl;
    cout << "自動推導的價格: " << price << endl;
    cout << "一週有 " << DAYS_IN_WEEK << " 天" << endl;
    return 0;
}
```


Microsoft Visual Studio 偵錯主
--- 資料型態佔用空間 (sizeof) ---
int 大小: 4 Bytes
double 大小: 8 Bytes
char 大小: 1 Byte
long long 大小: 8 Bytes

水果數量: 10
圓周率: 3.14159
成績: A
是否好玩: 是的
自動推導的價格: 150.5
一週有 7 天

第三週：字串、編譯原理與記憶體配置

涵蓋強大的 std::string 應用，並解析程式從原始碼到執行檔的「建置過程」，以及執行時期的記憶體邏輯分區。

3-1. 字串物件 (std::string) 深度解析

- 初始化技巧：支援重複字元構造 string s(10, '*') 與 Raw String R"(...)"（不需跳脫字元）。
- 基礎存取：.length() 取得長度；.at(i) 存取字元具備邊界檢查，較 [] 安全。
- 型別轉換：to_string() 數字轉字串；stoi() 字串轉整數。

3-2. 編譯核心：宣告與定義

- 宣告 **(Declaration)**：介紹名稱與型別給編譯器，不分配記憶體（如 extern）。
- 定義 **(Definition)**：實際分配記憶體空間或撰寫實作邏輯。
- **ODR** 原則：宣告可多次，但定義在程式中只能有一次。
- **DRY** 原則：系統中的每一部分，都必須有一個單一的、明確的、權威的代表

原則	對象	目的	違反後果
DRY	程式設計師	減少重複邏輯，好維護。	程式碼變髒、難改
ODR	編譯器/連結器	確保符號唯一，不衝突。	編譯失敗或連結失敗。

3-3. 作用域與記憶體分區

- 作用域 **(Scope)**：分為區域 (Local)、全域 (Global) 與遮蔽 (Shadowing)。
- 記憶體佈局 **(Runtime Layout)**：
  - 程式碼區：儲存二進位機器指令，唯讀以防止修改邏輯。
  - 靜態/全域區：儲存全域變數與 static 變數，程式結束才釋放。
  - **Stack (堆疊)**：存區域變數，自動管理，速度極快但空間有限。
  - **Heap (堆積)**：儲存動態分配數據（如 new），空間大但須手動管理。

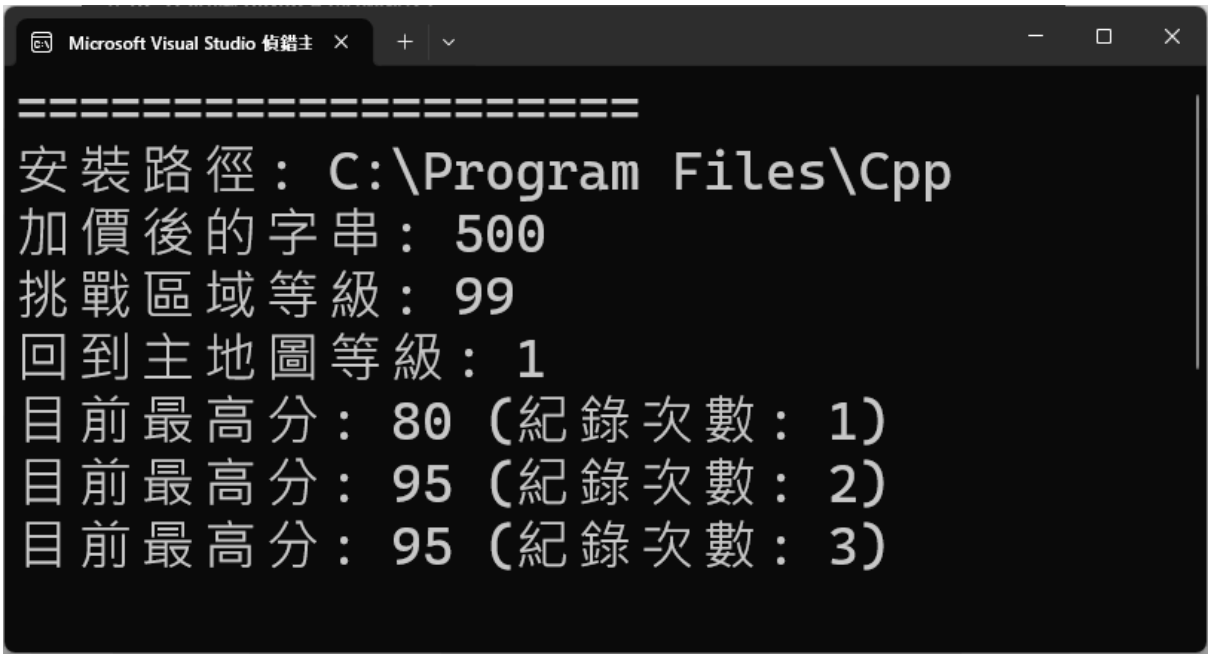
3-4. 關鍵字 static 之妙用

- 區域靜態變數：存於全域區，會「記住」上次的值，不隨函式結束消失 。
- 全域靜態：將可見性限制在目前的 .cpp 檔案內 。
- 類別靜態成員：屬於「類別」而非個別「物件」，所有實體共用同一份資料 。

程式演練

```cpp
#include <iostream>
#include <string>   // 處理字串必備
using namespace std;

// [3-2] 宣告與定義
extern int totalCalls; // 宣告 (告訴編譯器有這個人)
int totalCalls = 0;    // 定義 (分配記憶體空間)

// [3-4] static 區域變數：會「記住」上次的值
void recordScore(int score) {
    static int highestScore = 0; // 只會在第一次呼叫時初始化
    if (score > highestScore) {
        highestScore = score;
    }
    totalCalls++;
    cout << "目前最高分: " << highestScore << " (紀錄次數: " << totalCalls << ")" << endl;
}

int main() {
    // [3-1] std::string 進階技巧
    string divider(20, '=');          // 重複字元構造
    string rawPath = R"(C:\Program Files\Cpp)"; // Raw String (原始字串)

    cout << divider << endl;
    cout << "安裝路徑: " << rawPath << endl;

    // [3-1] 型別轉換
    string priceStr = "450";
    int price = stoi(priceStr) + 50;   // 字串轉整數並加 50
    cout << "加價後的字串: " << to_string(price) << endl;

    // [3-3] 作用域 (Scope)
    int level = 1;
    {
        int level = 99; // 遮蔽 (Shadowing)：外面的 level 被藏起來了
        cout << "挑戰區域等級: " << level << endl;
    }
```

```
    cout << "回到主地圖等級: " << level << endl;

    // [3-4] 測試 static 的記憶能力
    recordScore(80);
    recordScore(95);
    recordScore(70); // 雖然這次比較低，但 static 變數會記住 95

    return 0;
}
```



第四週：迴圈結構與進階控制邏輯

涵蓋 C++迴圈應用，解析「控制結構」如何改變程式執行流

4-1. 基礎迴圈控制 (Loop Control)

- **for 迴圈 (計數型)**：最常用於已知執行次數的場景。支援傳統 (init; cond; inc) 結構與現代 **Range-based for**（用於掃描陣列/容器）。
- **while 迴圈 (條件型)**：先檢查門票（條件）再執行。適合處理未知執行次數，僅依賴某種狀態改變的情況。
- **do-while 迴圈 (保證執行)**：先執行再檢查。保證程式碼至少執行一次，最常用於「功能選單」與「輸入驗證」。

4-2. 迴圈流程控制：跳轉關鍵字

- **break**：強制終止並跳出目前所在的整個迴圈。
- **continue**：跳過本次循環剩餘內容，直接進入下一次迭代。
- 無窮迴圈 **(Infinite Loop)**：使用 `while(true)` 或 `for(;;)`，常用於伺服器監聽或遊戲主迴圈。

4-3. 進階循環：STL 演算法與遞迴

- **for_each 演算法**：
  - 隸屬 `#include <algorithm>`，屬於高階控制結構。
  - 搭配 `std::begin()` 與 `std::end()`，讓傳統陣列也能交給「管家」自動處理。

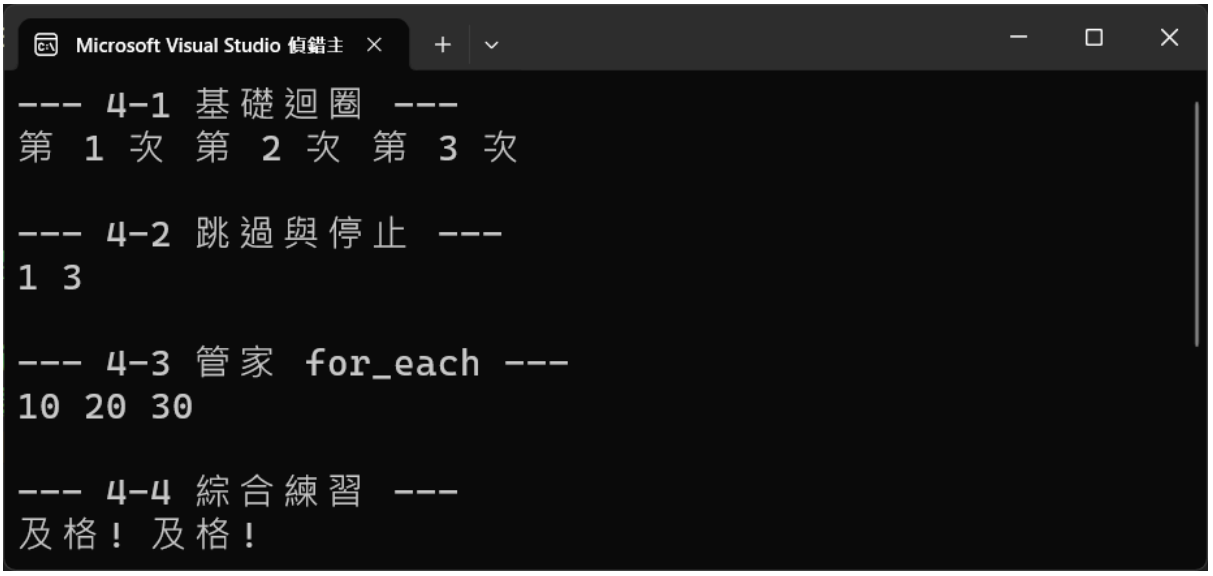
4-4. 程式三大控制結構解析

- 循序結構 **(Sequential)**：程式由上而下逐行執行，無分支跳轉。
- 選擇結構 **(Selection)**：根據條件轉彎，關鍵字為 `if-else` 與 `switch`
- 控制結構 **(Repetitive)**：即迴圈結構。透過條件判斷，控制 CPU 反覆執行特定區塊。

程式演練

```
#include <iostream>
#include <algorithm> // 4-3 的 for_each
using namespace std;

void printValue(int n) { cout << n << " "; } // 輔助工具

int main()
{
    // --- 4-1. 基礎迴圈：重複 3 次 ---
    cout << "--- 4-1 基礎迴圈 ---" << endl;
    for (int i = 1; i <= 3; i++)
    {
        cout << "第 " << i << " 次 ";
    }
    cout << "\n\n";

    // --- 4-2. 流程控制：遇到 2 就跳過，遇到 4 就停止 ---
    cout << "--- 4-2 跳過與停止 ---" << endl;
    for (int j = 1; j <= 5; j++)
    {
        if (j == 2)
            continue; // 跳過 2
        if (j == 4)
            break; // 看到 4 就停
        cout << j << " ";
    }
    cout << "\n\n";

    // --- 4-3. 進階工具處理陣列 ---
    cout << "--- 4-3 管家 for_each ---" << endl;
    int nums[] = {10, 20, 30};
    for_each(begin(nums), end(nums), printValue);
    cout << "\n\n";

    // --- 4-4. 三大結構：循序 -> 選擇 -> 控制 ---
    cout << "--- 4-4 綜合練習 ---" << endl;
    int score = 80; // 1. 循序 (給分數)
```

```cpp
    if (score >= 60)
    { // 2. 選擇 (判斷及格)
        for (int k = 0; k < 2; k++)
        { // 3. 控制 (重複印兩次)
            cout << "及格! ";
        }
    }

    return 0;
}
```



第五週：函式的定義與宣告

函式是一段只在被呼叫才會執行的程式碼，可以將資料(引數)回遞給函式，它們對於程式碼的複用很重要，一次定義，多次使用

5-1. 函式的基礎架構

- 一個標準的 C++ 函式長這樣：

```
傳回值型態 函式名稱(參數列表) {
    // 這裡是要執行的程式碼
    return 傳回值;
}
```

- 組成要素解析：

傳回值型態 **(Return Type)**	函式執行完後要交還給你的資料類型 (`int`,`double`, `string`)。 如果不需要傳回任何東西，就寫 `void`。
函式名稱 **(Function Name)**	為函式設定名稱，通常為函式本身用途之名稱 (如 `calculateScore`)
參數列表 **(Parameters)**	外部傳進去給函式用的資料。
Return 關鍵字	用來結束函式並回傳結果

5-2. 函式的引數和參數

1. 參數 **(Parameter)**：定義函式時，寫在括號裡的「變數名稱」。它是虛擬的佔位符。
2. 引數 **(Argument)**：呼叫函式時，實際傳進去的「數值」。它是真實的資料。

❖ C++ 中傳遞引數的三種方式

1. 傳值(Pass by value)

把資料影印一份送進去。函式裡怎麼改，都不會影響原本的變數。

- 優點：安全，不會弄壞原本的資料
- 缺點：如果資料很大，影印很慢且浪費記憶體

2. 傳址/指標 (Pass by Pointer)

把資料的地址送進去。函式透過地址直接操作原本的資料。

- 語法：void func(int* ptr)
- 缺點：要處理 NULL 指標，容易出錯

3. 傳參考 (Pass by Reference)

給原本的變數取個綽號。函式直接用這個綽號操作原本的資料，但不需影印

- 語法：`void func(int& ref)`
- 強大組合：`void func(const string& str)`

❖ 引數的特殊功能

- 預設引數

可以在定義函式時給參數一個預設值。如果呼叫時沒給引數，就用預設

```
void greet(string name = "客人") {
    cout << "你好，" << name << endl;
}

greet();          // 輸出：你好，客人
greet("小明");  // 輸出：你好，小明
```

5-3. 函式簽章與函式過載

在 C++ 中，一個函式的簽章由以下 兩大要素 組成：

- 函式過載可以讓多個函式「共用同一個名字」，只要它們的「引數拿法」不一樣。
- 透過函式過載，多個函式可以擁有相同的名稱，但具有不同的引數。
- 編譯器是透過「函式簽章」來區分的。只要滿足以下任一條件，就可以過載：
    1. 參數的 數量 不同。
    2. 參數的 型別 不同。
    3. 參數的 順序 不同。

- 函式簽章包含
    1. 函式的名稱 (Function Name)
    2. 參數列表 (Parameter List)：
        - ❖ 參數的型別（`int` 還是 `double`？）
        - ❖ 參數的數量（1 個還是 2 個？）
        - ❖ 參數的順序（是 `(int,double)` 還是 `(double,int`？）

- 函式簽章不包含
    1. 傳回值型態 **(Return Type)**：例如 `int` 或 `void`。
    2. 參數的名稱：例如你叫 `a` 還是叫 `b`。

Name Mangling（名稱修飾）：

C++ 編譯器為了支援「函式多載 」，偷偷幫你的函式重新取名的過程。

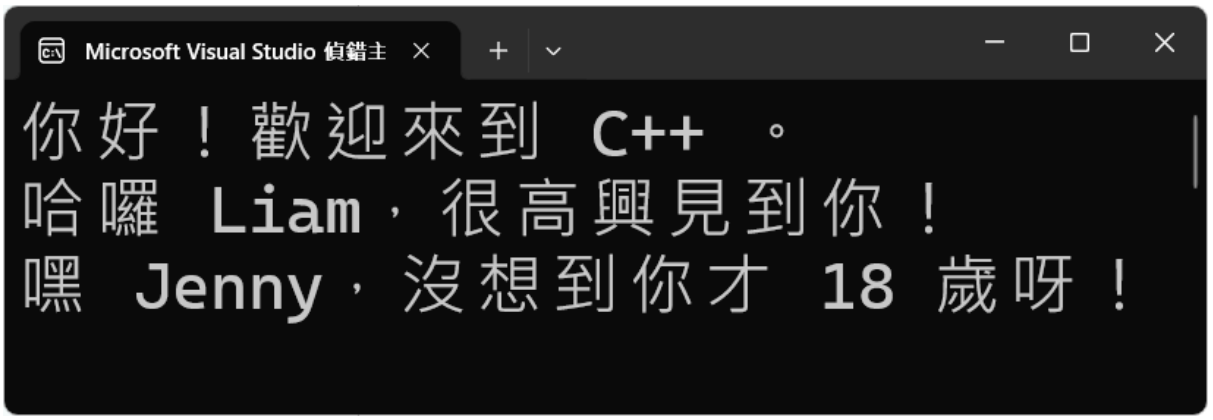
❖ 為什麼需要 Name Mangling？

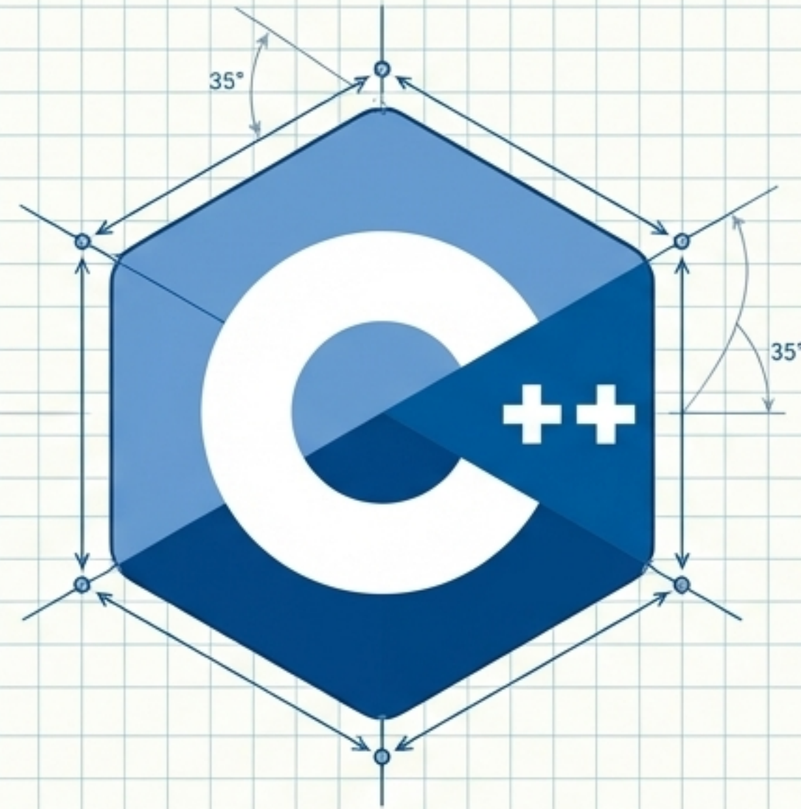
因為在 C 語言中，不支援函式多載，所以一個程式裡不能有兩個同名的函式。但在 C++ 中，我們可以寫出多個同名但參數不同的函式

特性	C 語言 (No Mangling)	C++ 語言 (With Mangling)
同名函式	不允許（會編譯錯誤）	允許（只要簽章不同）
符號名稱	原封不動（`print`）	加上參數資訊（例如:`_Z5printi`）
主要目的	簡單、直接	實現多型與多載

程式演練

```cpp
#include <iostream>
#include <string>

using namespace std;

// 1. 基礎架構：最簡單的打招呼 (無傳回值 void)
void sayHello() {
    cout << "你好！歡迎來到 C++ 。" << endl;
}
// 2. 引數與參數：傳入名字 (帶有一個參數)
void sayHello(string name) {
    cout << "哈囉 " << name << "，很高興見到你！" << endl;
}
// 3. 函式過載：傳入名字與年齡 (參數數量不同，簽章也不同)
void sayHello(string name, int age) {
    cout << "嘿 " << name << "，沒想到你才 " << age << " 歲呀！" << endl;
}

int main() {
    // 呼叫不同的函式版本，編譯器會根據「引數」自動對接正確的「簽章」

    sayHello();                 // 呼叫無參數版本
    sayHello("Liam");           // 傳入一個字串引數
    sayHello("Jenny", 18);      // 傳入字串與整數引數
    return 0;
}
```



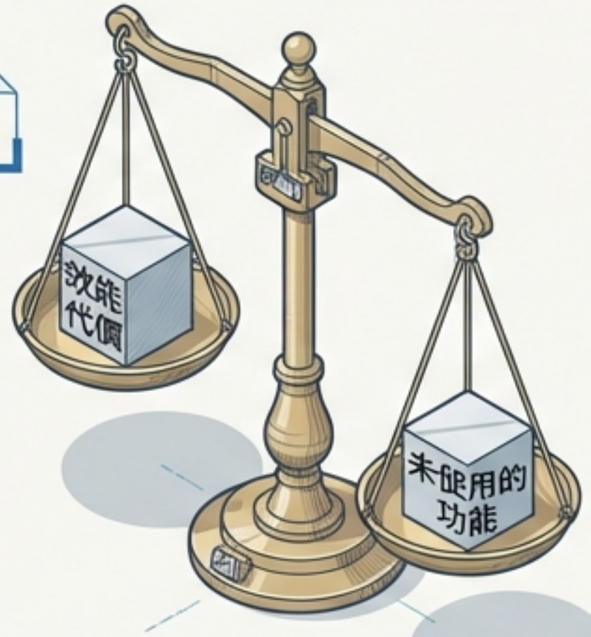


C++ 的建築學

從哲學思維到記憶體深處的工程實踐

架構解析：傅正榮

核心哲學：Zero-overhead



沒有用到的功能，絕對不需付出任何效能代價。
保留低階硬體控制力，同時施加嚴格的強型別檢查。

1979
C with Classes
(Bjarne Stroustrup 初創)

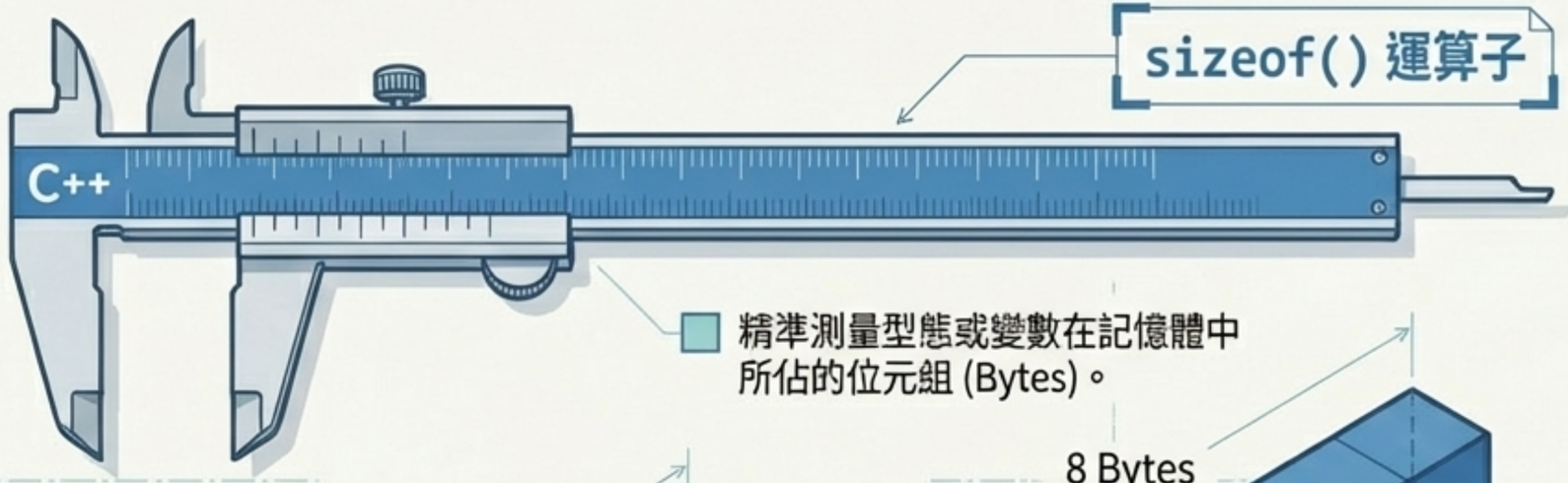
1983
正式更名 C++
(C 語言的遞增與進化)

C++98/03
奠定 OOP
與泛型基礎

C++11
現代化分水嶺
(auto，智慧指標)

C++20
架構革新
(Concepts、Modules)

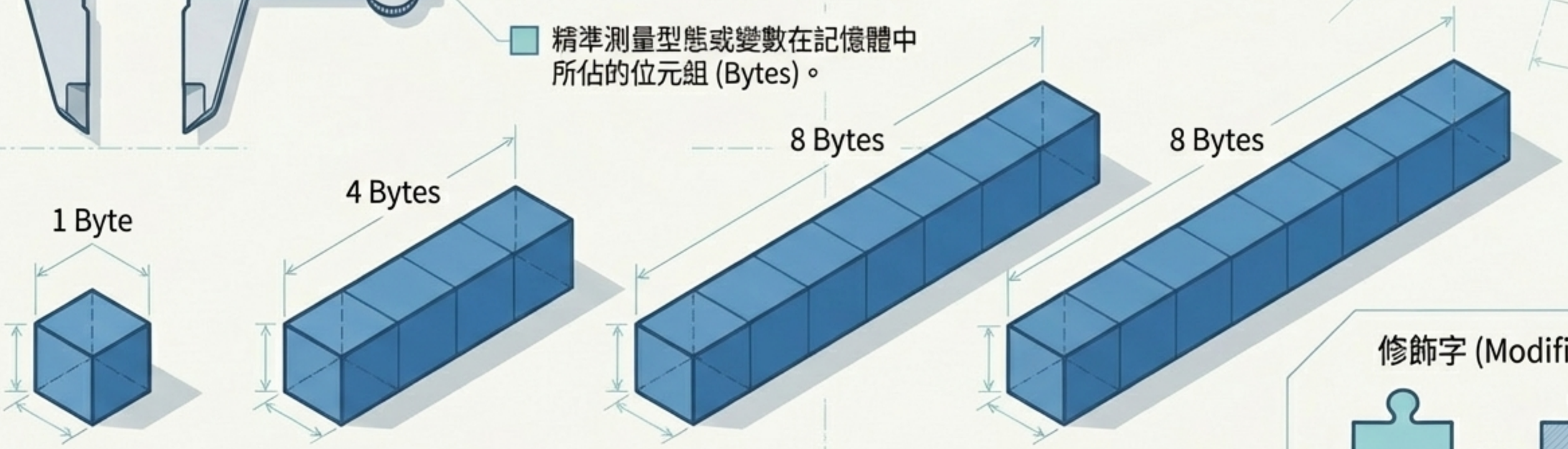
建材與規格：資料型態的物理空間



sizeof() 運算子

精準測量型態或變數在記憶體中所佔的位元組 (Bytes)。

精準測量型態或變數在記憶體中所佔的位元組 (Bytes)。



char / bool　int　double　long long

修飾字 (Modifiers)

unsigned (僅正數)　short / long (調整長度)

現代化建構法：變數初始化與推導

拷貝初始化 (傳統)

```
int appleCount = 10;
```

⚠ 略顯多餘

直接初始化

```
double pi(3.14159265);
```

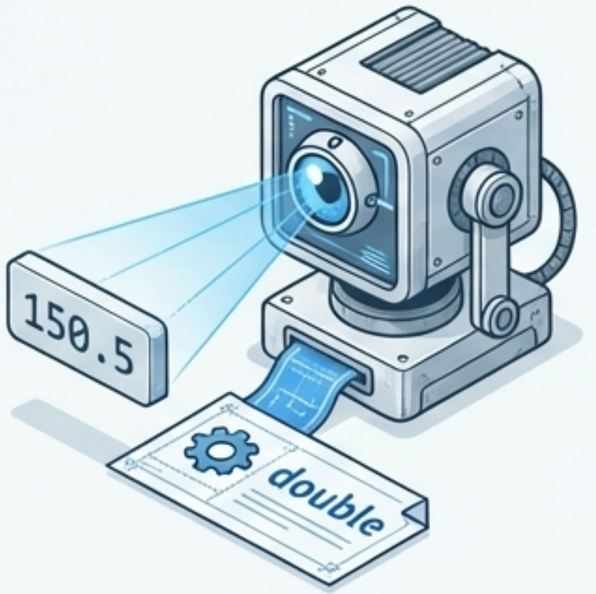
ℹ 適合物件構造

列表初始化 (現代 C++ 推薦)

```
char grade{ 'A' };
```

具備一致性，防止危險的「窄化轉換 (Narrowing Conversion)」

The Power of auto




```
auto price = 150.5;
```

⚠ 注意：宣告時必須給予初始值，編譯器才能自動推導。

常數安全矩陣 (The Constant Matrix)

	const	constexpr
本質	執行期常數 (唯讀變數)	編譯期常數 (純粹的常數)
計算時機	程式執行時 (Runtime)	程式編譯時 (Compile-time)
效能開銷	極低	絕對零開銷 (效能最優)
程式碼範例	`const int MAX_SCORE = 100;` 常用於函式參數保護	`constexpr int DAYS_IN_WEEK = 7;` 值在編譯時已算好

強型別防禦機制：嘗試修改常數 (例如 `MAX_SCORE = 110;`) 將被系統直接阻擋，導致編譯失敗。

編譯核心：宣告與定義的分野

宣告 Declaration


```
extern int totalCalls;
```

介紹名稱與型別給編譯器，告訴系統「有這個人」，但不分配實體記憶體空間。

定義 Definition


```
int totalCalls = 0;
```

實際分配記憶體空間，或撰寫完整的實作邏輯。

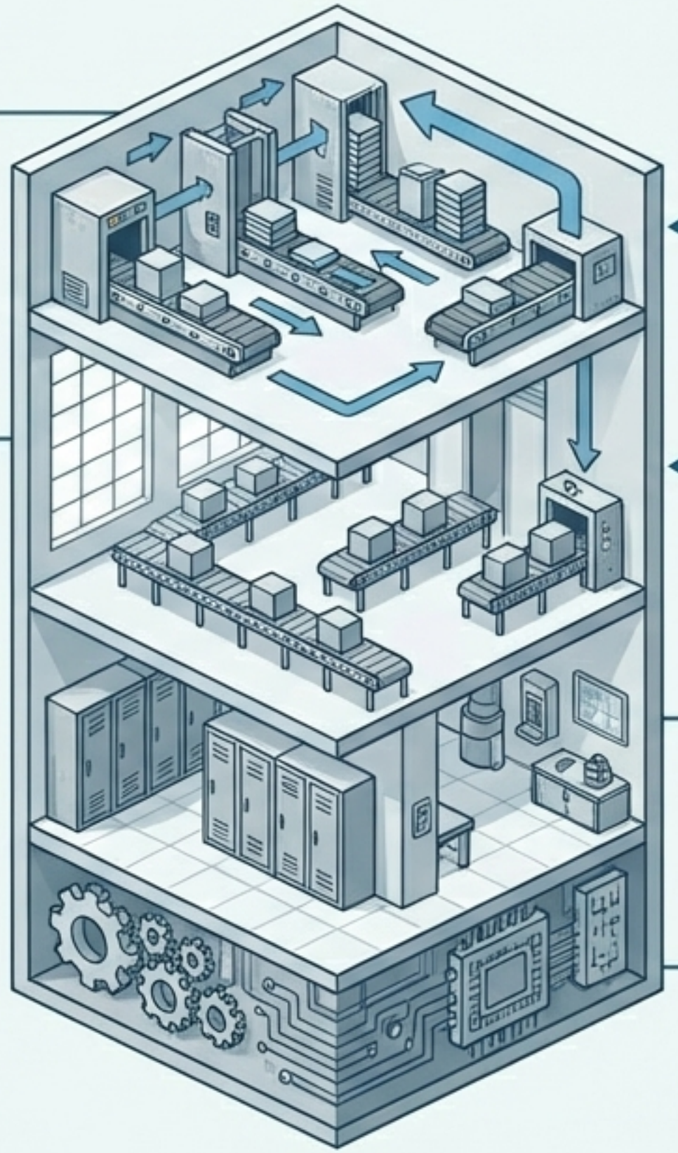
ODR 原則 (One Definition Rule)

宣告可以有很多次（到處發名片），但定義在整個程式中只能有一次（只能有一個置物櫃）。

記憶體大廈：執行時期的空間佈局

4F Stack (堆疊)
緊湊高速的頂層工作室。
特徵：儲存區域變數，自動進出管理，速度極快但空間有限。

3F Heap (堆積)
廣大自由的動態空間。
特徵：儲存動態分配數據 (如 new)，須手動管理。



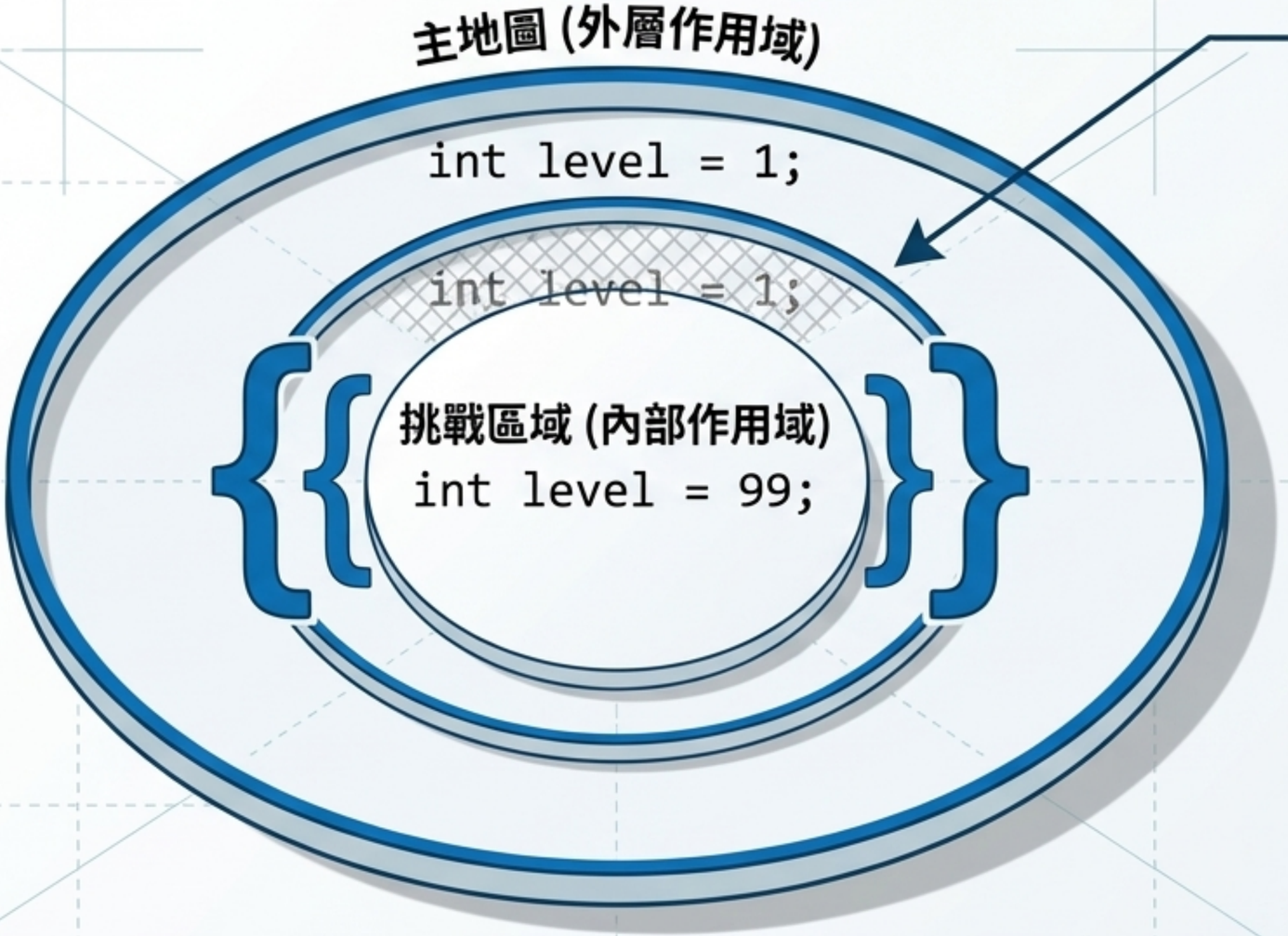
std::string 的內部機制
例如 R"(C:\Path)"。

內部機制通常將指標與長度儲存於 Stack，而將實際龐大的字串數據放於 Heap，以兼顧安全與彈性。

2F 靜態 / 全域區 (Static / Global)
儲存全域變數與 static 變數。
特徵：與應用程式同壽命，程式結束才釋放。

1F 程式碼區 (Code / Text)
儲存二進位機器指令。
特徵：唯讀 (Read-only)，防止執行中修改邏輯。

空間與動線：作用域 (Scope) 與遮蔽效應



遮蔽效應 (Shadowing)

- 當進入內部括號，外面的同名變數會被「暫時藏起來」。
- 一旦離開括號（離開 Stack 作用域），內部的 level 99 自動消滅。
- 外層的 level 1 隨即恢復，再次掌控全局。

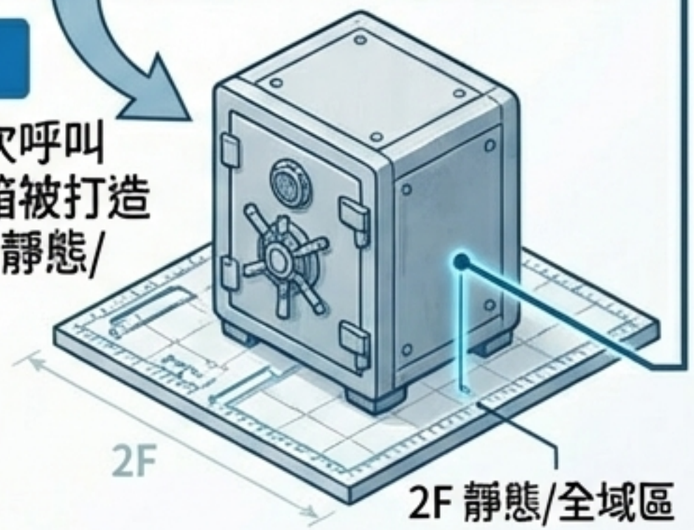
挑戰區域等級：99
回到主地圖等級：1

記憶魔法：static 關鍵字的跨越

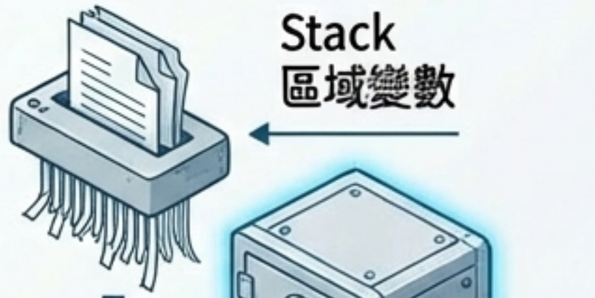
```
void recordScore(int score) {
  static int highestScore = 0;
}
```

初始化

函式第一次呼叫時，保險箱被打造並鎖在 2F 靜態/全域區。

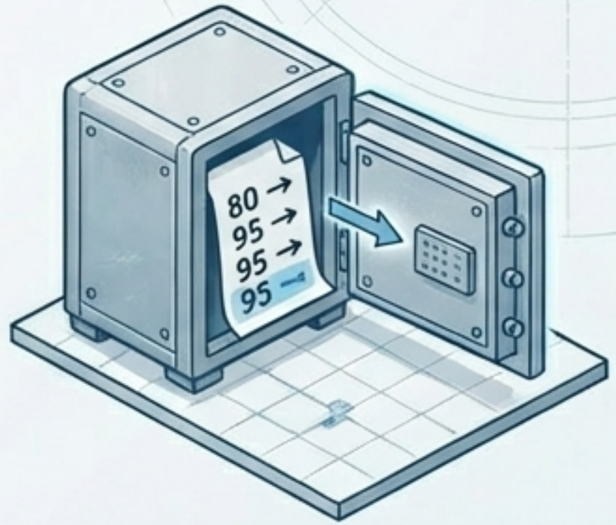


函式結束



離開 Stack 時，一般區域變數被摧毀，但 static 保險箱安然無恙。

再次呼叫



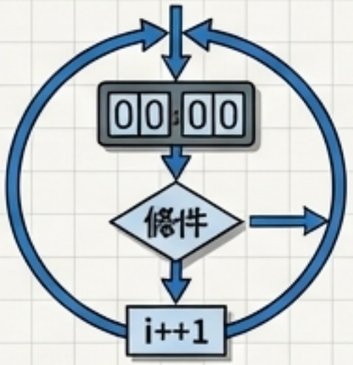
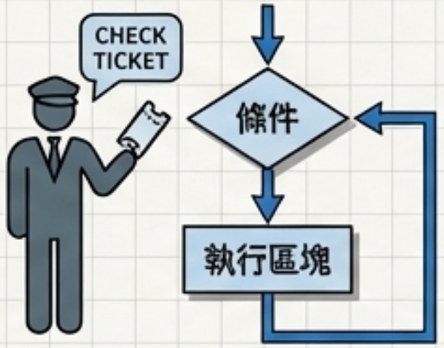
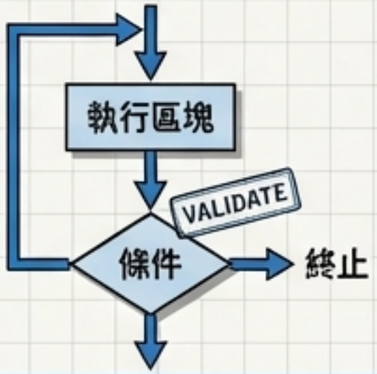
直接打開保險箱，它依然「記住」上次的數值 (例如連續記錄 80 -> 95 -> 95)。

進階延伸 (Extension)

全域靜態：
限制可見性在單一 .cpp 檔案內。

類別靜態：屬於「類別藍圖」而非個別物件實體，所有實體共用同一份資料。

迴圈決策矩陣 (The Loop Selection Matrix)

for 迴圈 (計數型)	while 迴圈 (條件型)	do-while 迴圈 (保證執行)
		
決策時機：已知明確的執行次數。 **現代應用：**Range-based for 或 for_each (搭配 std::begin) 自動掃描陣列或容器。	**決策時機：**未知執行次數，僅依賴狀態改變。 **特徵：**先檢查門票，再決定是否放行執行。(可用 while(true) 做無窮伺服器監聽)。	**決策時機：**功能選單或輸入驗證。 **特徵：**先無條件執行一次，再檢查是否繼續。

break – 強制終止並跳出整個迴圈。

continue – 跳過本次循環剩餘內容，直接進入下一次迭代。

模組化構建：函式的解剖學

1 **傳回值型態 (Return Type)**
執行完後交還的資料類型 (不回傳則用 void)。

2 **函式名稱 (Function Name)**
該模組用途的唯一識別標籤。

3 **參數列表 (Parameters)**
外部傳進來的虛擬佔位符 (引數 Argument 是實際傳入的數值)。

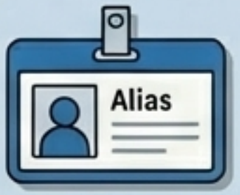
```
int calculateScore(int base, double bonus) {
    ...
    return total;
}
```

4 **Return 關鍵字**
結束函式執行並將結果回傳。

核心定義：函式簽章 (Function Signature)

僅包含「函式名稱」與「參數列表的數量、型別、順序」。
絕對不包含傳回值型態！這是多載 (Overloading) 的唯一判定基準。

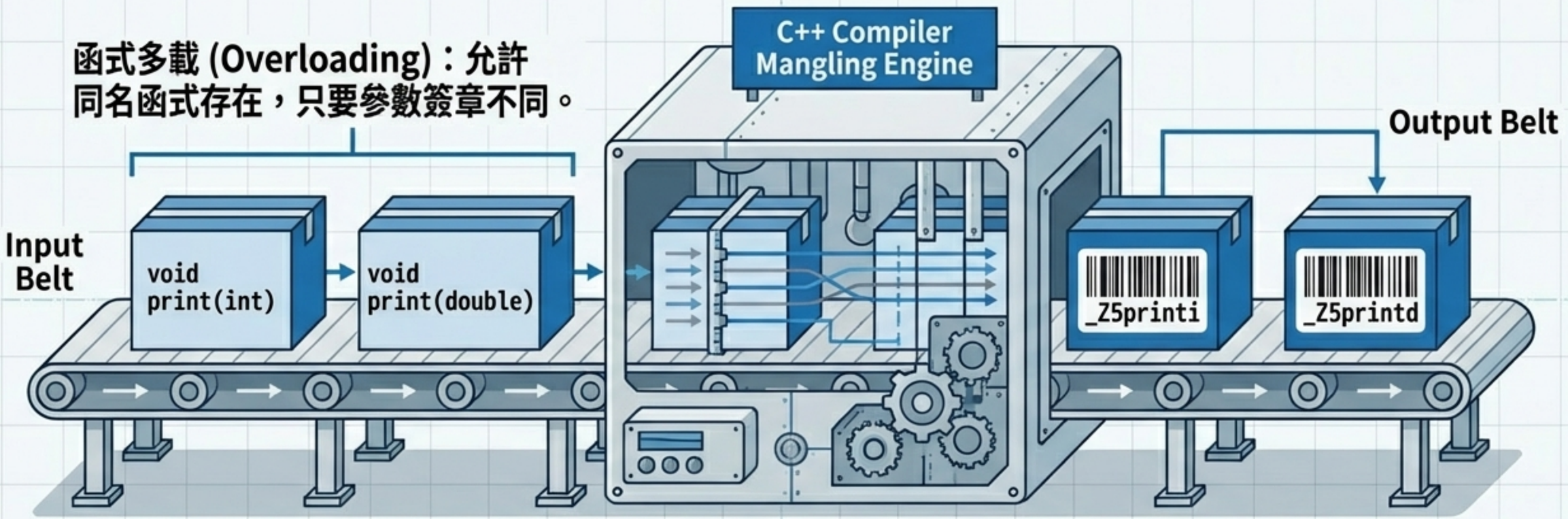
管線與樞紐：引數傳遞架構表

	**1. 傳值** **(Pass by Value)**	**語法**：`void func(int val)` **記憶體開銷**：高（需要影印一份全新資料） **安全性與特徵**：極安全。函式內修改絕對不影響原變數。
	**2. 傳址/指標** **(Pass by Pointer)**	**語法**：`void func(int* ptr)` **記憶體開銷**：低（僅傳遞記憶體位址） **安全性與特徵**：透過位址直接修改原資料。需手動處理 NULL 潛在風險。
	**3. 傳參考** **(Pass by Reference)**	**語法**：`void func(int& ref)` **記憶體開銷**：極低（零拷貝） **安全性與特徵**：給原變數取「綽號」。直接操作原資料，語法比指標更安全乾淨。

最強悍的組合：`void func(const string& str)` —
唯讀且免除影印開銷，現代 C++ 標準配備。

編譯器底層：Name Mangling (名稱修飾)

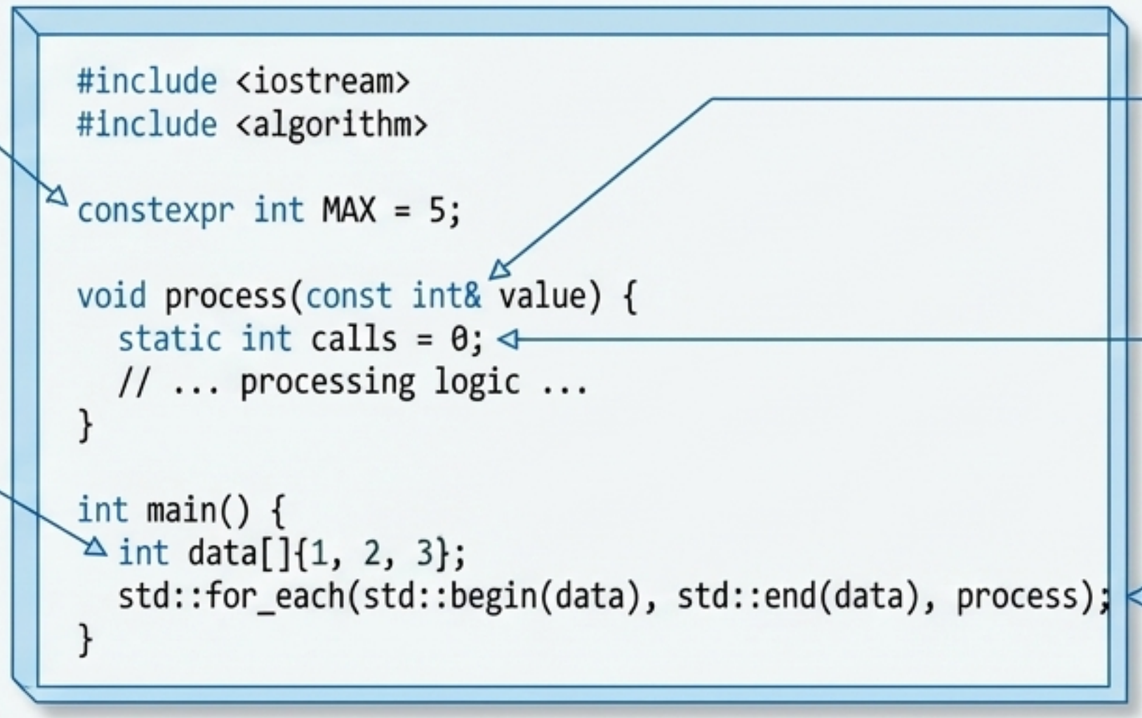
函式多載 (Overloading)：允許同名函式存在，只要參數簽章不同。



編譯器行為對照

C 語言	不允許同名函式。編譯器原封不動保留符號 (print)。
C++ 語言	完美支援多型。編譯器自動將參數資訊編碼進名稱中，解決命名衝突。

融會貫通：C++ 程式解剖學



1 **Zero-overhead:** 編譯期常數，執行期零負擔。

2 **Type Safety:** 列表初始化防窄化，配置於 Stack 記憶體。

3 **Optimal Routing:** 傳參考免影印開銷，const 保證資料絕對安全。

4 **Memory Architecture:** 駐留於全域/靜態區，超越 Stack 壽命的保險箱。

5 **Modern Control Flow:** 現代化 STL 高階迭代控制。

不只是寫出語法，而是掌控每一 Byte 的空間與每一微秒的效能。