C++程式設計：函數

**目錄**

[C++程式設計：函數 1](#_Toc204198592)

[1. 基本觀念 2](#_Toc204198593)

[1.1 範例：呼叫函數 2](#_Toc204198594)

[2. Function Declaration/Definition/Calls 3](#_Toc204198595)

[2.1 Function Declaration 函數聲明 3](#_Toc204198596)

[2.2 Function Definition 4](#_Toc204198597)

[2.3 Function Call 4](#_Toc204198598)

[2.4 範例：使用Function Declaration、Function Definition跟Function Call。 5](#_Toc204198599)

[2.5 向前申明(forward declaration) 5](#_Toc204198600)

[2.6 有回傳值的函數(單個參數) 6](#_Toc204198601)

[2.7 無回傳值的函數(單個參數) 6](#_Toc204198602)

[2.8 有回傳值的函數(多個參數) 7](#_Toc204198603)

[3. 參數傳遞 7](#_Toc204198604)

[3.1 預設參數值(Default Parameter Value) 8](#_Toc204198605)

[3.2 範例：單一預設參數值 8](#_Toc204198606)

[3.3 範例：多個預設參數值 9](#_Toc204198607)

[4. 傳值/傳位址/傳參考 9](#_Toc204198608)

[4.1 範例：主程式與函數內的地址不同 9](#_Toc204198609)

[4.2 傳值呼叫(Pass by Value) 10](#_Toc204198610)

[4.3 傳位址呼叫(Pass by Address) 10](#_Toc204198611)

[4.4 傳參考呼叫(Pass by Reference) 11](#_Toc204198612)

[5. 陣列參數傳遞與修改 11](#_Toc204198613)

[5.1 範例：傳遞陣列並印出內容 11](#_Toc204198614)

[5.2 範例：透過函式修改陣列內容(+1) 12](#_Toc204198615)

[6. Inline Function(內聯函式) 13](#_Toc204198616)

[6.1 Inline 函式語法 13](#_Toc204198617)

[6.2 範例：使用 Inline 計算平方 14](#_Toc204198618)

[6.3 範例：使用 Inline找最大值與絕對值 14](#_Toc204198619)

[7. 補充：可變參數函式(Variadic Functions) 15](#_Toc204198620)

[7.1 基本格式 15](#_Toc204198621)

[7.2 範例：可以加任意個數字的add() 函數 15](#_Toc204198622)

[7.3 範例：自訂參數數量的 print() 函式 16](#_Toc204198623)

[7.4 範例：交錯輸出整數與浮點數 17](#_Toc204198624)

[8. Lifetime(生命週期) 17](#_Toc204198625)

[8.1 範例：變數生命週期 18](#_Toc204198626)

[9. Scope 18](#_Toc204198627)

[10. static 19](#_Toc204198628)

[10.1 範例：記住函式中的變數值(區域 static) 19](#_Toc204198629)

[10.2 範例：變數封裝(檔案內可見) 19](#_Toc204198630)

[10.3 範例：函數封裝 20](#_Toc204198631)

[10.4 範例：class中的 static 成員變數 21](#_Toc204198632)

[11. static inline 21](#_Toc204198633)

[11.1 範例 ：單一檔案中宣告與使用 static inline 函式 22](#_Toc204198634)

[11.2 錯誤寫法：謹寫static沒寫inline 22](#_Toc204198635)

[12. extern 22](#_Toc204198636)

[13. 其他練習題目 23](#_Toc204198637)

[13.1 陣列搜尋(Array Search) 23](#_Toc204198638)

[13.2 判斷質數 23](#_Toc204198639)

[13.3 區間內所有質數 24](#_Toc204198640)

# 基本觀念

函式是組成程式的基本元件。每個程式中至少包含一個主程式(main function)，也就是主程序(main program)。

函數可以分為以下兩種：

|  |  |
| --- | --- |
| 系統定義函式 (System-defined Function) | * 由系統或標準函式庫(如 C++ STL)預先定義，使用者直接使用，無須自己撰寫。 * e.g., sqrt() |
| 使用者定義函式 (User-defined Function) | 使用者自行定義與撰寫的函式，依需求設計功能。像是使用者自己撰寫 main() 函式，或自定義一個加法函式。 |

函數整體可以分成五個部分：

* 標頭檔(Header File)
* 函式名稱(Function Name)
* 函式參數(Function Parameter)
* 函式回傳值(Return Value)
* 程式庫(Library)

## 範例：呼叫函數

#include <iostream> *// 標頭檔，用來輸出結果*

#include <cmath> *// 標頭檔，提供 sqrt 函式*

using namespace std;

int main() {

double number = 25.0;

double result;

result = sqrt(number);

cout << "平方根是：" << result << endl; *// 平方根是：5*

return 0;

}

# Function Declaration/Definition/Calls

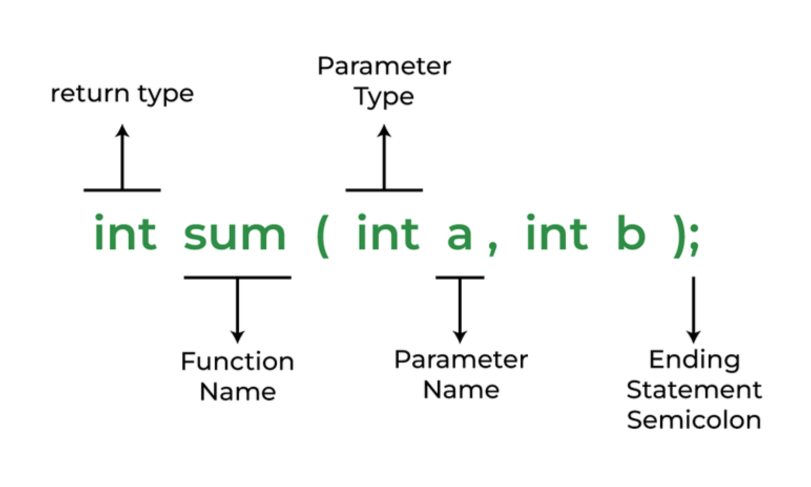
在程式碼段，可以分三個方面：

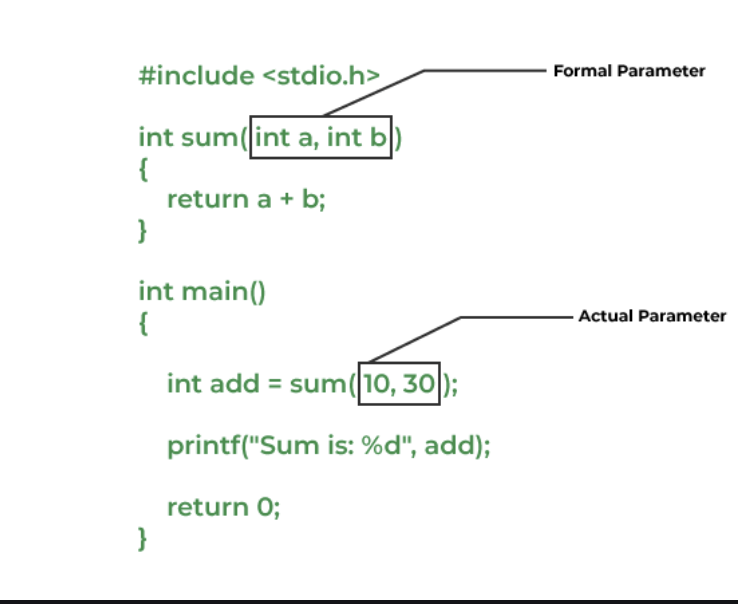
* Function Declaration 函數聲明
* Function Definition 功能定義
* Function Calls 函數調用

## Function Declaration 函數聲明

Function Declaration是告訴編譯器，程式中某處有一個具有該名稱的函式被定義。在Function Declaration中，我們必須提供函式名稱、其回傳型別，以及參數的數量與型別。

Figure 1：Function Declaration





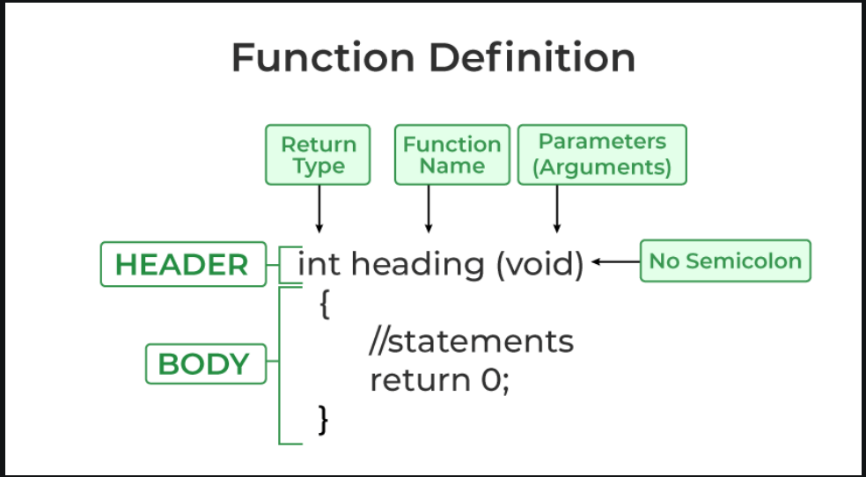
Q：函式的「引數(arguments)」與「參數(parameters)」有什麼不同

* 參數(parameters)是定義在函式宣告或定義中的變數。
* 引數(arguments)是在呼叫函式時實際傳入的值。

## Function Definition

Function Definition包含實際的敘述(statements)，這些敘述會在函式被呼叫時執行(也就是當程式的控制流程進入該函式時)。

Figure 2：Function Definition



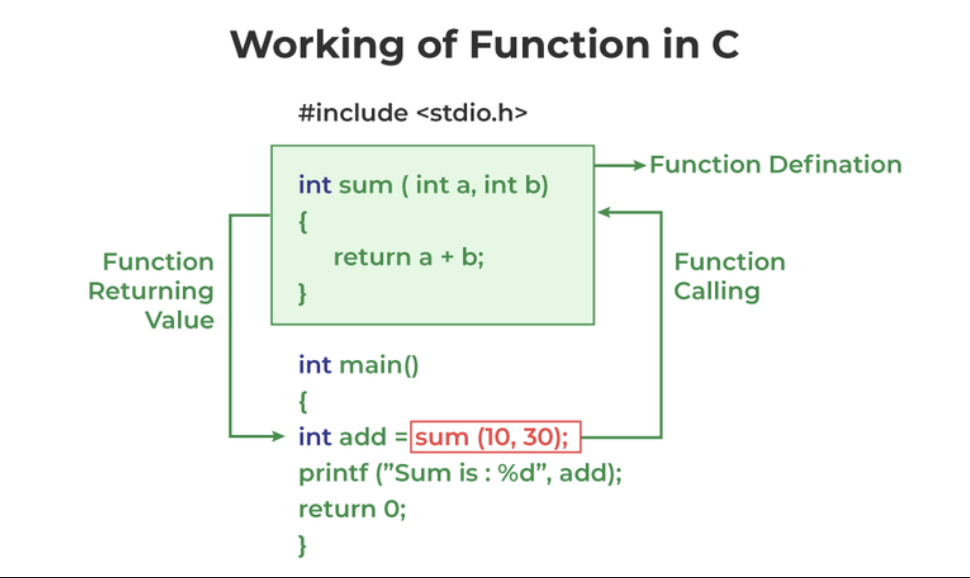
## Function Call

函式呼叫是一種敘述，用來指示編譯器去執行該函式。我們在呼叫時會使用函式名稱與對應的參數。

Q：函式宣告(declaration)與函式定義(definition)有什麼不同？

* 函式宣告包含函式名稱、回傳型別與參數等資訊。
* 函式定義則包含該函式的實際程式區塊(函式本體)

Figure 3：Function Call



## 範例：使用Function Declaration、Function Definition跟Function Call。

#include <iostream>

using namespace std;

*// Function Declaration(函式宣告)*

int add(int a, int b);

*// Function Definition(函式定義)：定義函式本體：實際執行加法並回傳結果*

int add(int a, int b) {

return a + b;

}

int main() {

int x = 10;

int y = 20;

int result = add(x, y); *// Function Call(函式呼叫)*

cout << "x + y = " << result << endl; *// x + y = 30*

return 0;

}

## 向前申明(forward declaration)

有時候，我們會在函式呼叫之後才定義該函式，以提升程式的可讀性。在這種情況下，我們會在函式定義與呼叫之前先進行宣告，這種宣告稱為「前向宣告(forward declaration)」。

#include <iostream>

using namespace std;

*// Function Declaration(函式宣告)*

int add(int a, int b);

int main() {

int x = 10;

int y = 20;

int result = add(x, y); *// Function Call(函式呼叫)*

cout << "x + y = " << result << endl; *// x + y = 30*

return 0;

}

*// Function Definition(函式定義)：定義函式本體：實際執行加法並回傳結果*

int add(int a, int b) {

return a + b;

}

## 有回傳值的函數(單個參數)

函式會接收一個參數，並回傳一個計算結果給主程式。

#include <iostream>

using namespace std;

*// 定義函式：傳入一個整數，回傳它的平方*

int square(int x) {

return x \* x;

}

int main() {

int number = 5;

int result = square(number); *// 呼叫函式，接收回傳值*

cout << "平方為：" << result << endl;

return 0;

}

## 無回傳值的函數(單個參數)

函式接收一個參數，但不回傳任何資料，只做某種動作(如輸出)。

#include <iostream>

using namespace std;

*// 定義函式：傳入一個整數，顯示它的 1~9 乘法表*

void printTable(int n) {

for (int i = 1; i <= 9; i++) {

cout << n << " x " << i << " = " << n \* i << endl;

}

}

int main() {

int number = 5;

printTable(number); *// 呼叫無回傳值的函式*

return 0;

}

*// 5 x 1 = 5*

*// 5 x 2 = 10*

*// 5 x 3 = 15*

*// 5 x 4 = 20*

*// 5 x 5 = 25*

*// 5 x 6 = 30*

*// 5 x 7 = 35*

*// 5 x 8 = 40*

*// 5 x 9 = 45*

## 有回傳值的函數(多個參數)

函式接受多個參數，進行運算並回傳結果。

#include <iostream>

using namespace std;

*// 定義函式：傳入兩個整數，回傳它們的和*

int add(int a, int b) {

return a + b;

}

int main() {

int x = 3, y = 2;

int sum = add(x, y); *// 呼叫函式*

cout << "總和為：" << sum << endl; *// 總和為：5*

return 0;

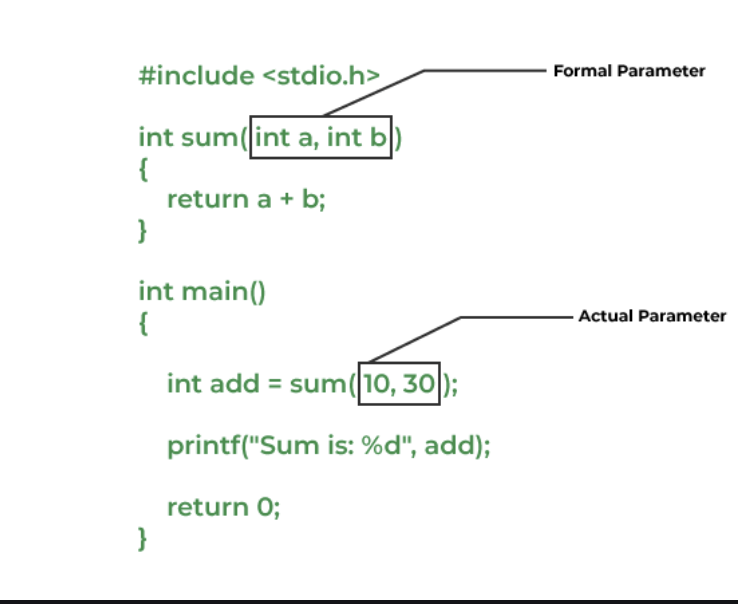
}

# 參數傳遞

C++的參數傳遞機制可以分為兩種：

* 形式參數(Formal Parameter)：Parameter：函式定義中所宣告的變數，用來接收傳入資料
* 實際參數(Actual Parameter)：Argument：呼叫函式時所傳入的值

Figure 4：形式參數與實際參數



// 查看形式參數與實際參數

#include <iostream>

using namespace std;

int square(int x) { *// x 是「形式參數」*

return x \* x;

}

int main() {

int num = 5;

int result = square(num); *// num 是「實際參數」*

cout << "平方為：" << result << endl;

return 0;

}

## 預設參數值(Default Parameter Value)

當呼叫函式時，若沒有傳入某些參數值，函式會使用預先設定好的預設值。這個功能讓函式呼叫變得更彈性、更方便，尤其在參數很多但大部分情況下使用預設行為時。

## 範例：單一預設參數值

#include <iostream>

using namespace std;

// 宣告與定義都有預設值也可以

void greet(string name = "使用者") {

    cout << "你好，" << name << "！" << endl;

}

int main() {

    greet();          // 你好，使用者！

    greet("小明");    // 你好，小明！

    return 0;

}

## 範例：多個預設參數值

#include <iostream>

using namespace std;

void display(int a = 1, int b = 2, int c = 3) {

    cout << a << " " << b << " " << c << endl;

}

int main() {

    display(); // 1 2 3

    display(10); // 10 2 3

    display(10, 20); // 10 20 3

    display(10, 20, 30);  // 10 20 30

    return 0;

}

# 傳值/傳位址/傳參考

## 範例：主程式與函數內的地址不同

#include <iostream>

using namespace std;

void test(int j) { // 函式定義，參數 j 是形式參數(formal parameter)

    cout << "變數 j 的值 = " << j << endl;

    cout << "變數 j 的地址 = " << &j << endl;

    j++;  // 修改 j，不會影響原始 i

}

int main() {

    int i = 5;

    cout << "變數 i 的值 = " << i << endl;

    cout << "變數 i 的地址 = " << &i << endl;

    test(i);  // 傳值呼叫(pass by value)

    cout << "變數 i 的值 = " << i << endl;  // 應該仍為原值

    cout << "變數 i 的地址 = " << &i << endl;

    return 0;

}

// 變數 i 的值 = 5

// 變數 i 的地址 = 0x7ffffcbfc

// 變數 j 的值 = 5

// 變數 j 的地址 = 0x7ffffcbd0

// 變數 i 的值 = 5

// 變數 i 的地址 = 0x7ffffcbfc

## 傳值呼叫(Pass by Value)

將值「複製」給函式的形式參數。函式接收到的是一份複製值，在函式中對參數的改動不會影響原變數。

#include <iostream>

using namespace std;

void setToTen(int x) {

    x = 10;

}

int main() {

    int a = 5;

    setToTen(a);

    cout << a << endl; // 👉 輸出仍是 5

}

## 傳位址呼叫(Pass by Address)

函式接收到的是變數的記憶體位址，必須使用指標操作(\*解參考)

#include <iostream>

using namespace std;

void setToTen(int\* x) {

    \*x = 10;

}

int main() {

    int a = 5;

    setToTen(&a);         // 傳址(&a)

    cout << a << endl;    // 👉 輸出為 10

}

## 傳參考呼叫(Call by Reference)

傳遞變數的「別名(引用)」給函式，函式內部的參數就是原變數的「別名」。不需使用指標語法，更直觀。

#include <iostream>

using namespace std;

void setToTen(int &x) {

    x = 10;

}

int main() {

    int a = 5;

    setToTen(a);

    cout << a << endl;  // 👉 輸出為 10

}

# 陣列參數傳遞與修改

函式無法一次回傳多個值，但可以通過陣列、指標、結構體達成。也因此，陣列在函數中是歸類於call by address，值是可以備修改的。

## 範例：傳遞陣列並印出內容

#include <iostream>

using namespace std;

#define ASIZE 3

#define BSIZE 5

// 印出陣列內容與位址

void printArray(int array[], int n) {

    cout << "array is at " << array << endl;

    for (int i = 0; i < n; i++) {

        cout << "array[" << i << "] = " << array[i] << endl;

    }

}

int main() {

    int a[ASIZE] = {3, 5, 7};

    int b[BSIZE] = {2, 4, 6, 8, 10};

    cout << "印出 a 陣列位址 = " << a << endl;

    cout << "印出 b 陣列位址 = " << b << endl;

    printArray(a, ASIZE);

    printArray(b, BSIZE);

    return 0;

}

// array is at 0x7ffffcbf4

// array[0] = 3

// array[1] = 5

// array[2] = 7

// array is at 0x7ffffcbe0

// array[0] = 2

// array[1] = 4

// array[2] = 6

// array[3] = 8

// array[4] = 10

## 範例：透過函式修改陣列內容(+1)

#include <iostream>

using namespace std;

#define ASIZE 5

// 印出陣列內容

void printArray(int array[], int n) {

    cout << "array is at " << array << endl;

    for (int i = 0; i < n; i++) {

        cout << "array[" << i << "] = " << array[i] << " ";

    }

    cout << endl;

}

// 陣列加一(直接修改原陣列內容)

void addArray(int array[], int n) {

    cout << "array is at " << array << endl;

    for (int i = 0; i < n; i++) {

        array[i]++;  // 修改原始記憶體內容

    }

}

int main() {

    int a[ASIZE] = {2, 4, 6, 8, 10};

    cout << "印出 a 陣列 = " << a << endl;

    printArray(a, ASIZE);  // 修改前

    addArray(a, ASIZE);    // 修改內容

    printArray(a, ASIZE);  // 修改後

    return 0;

}

// 印出 a 陣列 = 0x7ffffcbe0

// array is at 0x7ffffcbe0

// array[0] = 2 array[1] = 4 array[2] = 6 array[3] = 8 array[4] = 10

// array is at 0x7ffffcbe0

// array is at 0x7ffffcbe0

// array[0] = 3 array[1] = 5 array[2] = 7 array[3] = 9 array[4] = 11

# Inline Function(內聯函式)

很常跟Macro 比較

inline 函式是一種小型函式優化技巧，建議編譯器：

* 在呼叫這個函式的地方，直接插入程式碼本體(展開)
* 而不是像一般函式那樣「呼叫 → 跳轉 → 返回」

C++ 中的 inline是對編譯器的建議，而不是命令。編譯器可能會忽略你的 inline，特別是函式太大或有遞迴時。

Q：為什麼我寫了 inline 函式，還出現找不到定義

* 你可能把 inline 寫在 .cpp 檔中，但沒有給 header 看到它
* inline 應寫在 .h 檔(或 header 區)才會真正展開(因為必須「看到實體」)

## Inline 函式語法

inline 返回型別 函式名稱(參數) {

    // 函式內容

}

## 範例：使用 Inline 計算平方

#include <iostream>

using namespace std;

// 宣告 inline 函式

inline int square(int x) {

    return x \* x;

}

int main() {

    int n = 5;

    int result = square(n);  // 這裡會被展開成：n \* n

    cout << "平方是：" << result << endl; // 平方是：25

    return 0;

}

## 範例：使用 Inline找最大值與絕對值

#include <iostream>

using namespace std;

// 最大值

inline int max(int a, int b) {

    return (a > b) ? a : b;

}

// 絕對值

inline int abs(int x) {

    return (x < 0) ? -x : x;

}

int main() {

    cout << max(1, 2) << endl; // 2

    cout << abs(-3) << endl; // 3

    return 0;

}

Table 1：傳統函式呼叫 vs Inline

|  |  |
| --- | --- |
| **傳統函式呼叫問題** | **inline 解法** |
| 每次呼叫都要跳進函式再跳回來(有開銷) | 直接在呼叫點「展開」 |
| 小函式會呼叫很多次很浪費效能 | 展開後沒有 call → return |

Table 2：Macro vs Inline

|  |  |  |
| --- | --- | --- |
| **比較項目** | **Macro(巨集)** | **inline 函式** |
| 語法檢查 | ❌ 沒有 | ✅ 有型別檢查 |
| 調試 | ❌ 無法追蹤 | ✅ 可除錯 |
| 安全性 | ❌ 可能出錯 | ✅ 安全強 |
| 功能擴充 | ❌ 限制多 | ✅ 支援多參數、多型別 |

# 補充：可變參數函式(Variadic Functions)

一般函式的參數數量是固定的，例如int add(int a, int b);只能加兩個數。但我們能可以發現，以些函數他並無法確定要傳的參數數量，像 C 語言中的 printf。這種「參數數量可變」的函式就叫 Variadic Function

這些定義都來自 C++的<cstdarg>(繼承自 C 的`stdarg.h `)搭配這些工具：

* `va\_list`：宣告變數，用來存「參數清單」
* `va\_start`：開始讀取可變參數
* `va\_arg`：每次取出一個參數(並指向下一個)。你必須知道每個參數的型別，`va\_arg`不會自己判斷型別，錯誤會導致 undefined behavior
* `va\_end`：結束後要呼叫，避免記憶體問題

## 基本格式

#include <cstdarg>

return\_type function\_name(fixed\_arg, ...); // ... 表示後面可變參數

## 範例：可以加任意個數字的add() 函數

#include <iostream>

#include <cstdarg>

using namespace std;

// 可變參數函式

int add(int count, ...) {

    va\_list args;

    va\_start(args, count);

    int sum = 0;

    for (int i = 0; i < count; i++) {

        sum += va\_arg(args, int); // 每次拿一個 int

    }

    va\_end(args);

    return sum;

}

int main() {

    cout << add(2, 10, 20) << endl;          // ➜ 30

    cout << add(4, 1, 2, 3, 4) << endl;      // ➜ 10

    cout << add(1, 99) << endl;              // ➜ 99

    return 0;

}

## 範例：自訂參數數量的 print() 函式

#include <iostream>

#include <cstdarg>

using namespace std;

void print(int count, ...) {

    va\_list args;

    va\_start(args, count);

    for (int i = 0; i < count; i++) {

        int value = va\_arg(args, int);

        cout << value << " ";

    }

    va\_end(args);

    cout << endl;

}

int main() {

    print(3, 10, 20, 30);  // ➜ 10 20 30

    print(5, 1, 2, 3, 4, 5); // ➜ 1 2 3 4 5

    return 0;

}

## 範例：交錯輸出整數與浮點數

#include <iostream>

#include <cstdarg>

using namespace std;

void printMixed(int count, ...) {

    va\_list args;

    va\_start(args, count);

    for (int i = 0; i < count; i++) {

        if (i % 2 == 0) {

            int val = va\_arg(args, int);

            cout << "整數: " << val << endl;

        } else {

            double val = va\_arg(args, double);  // 注意浮點數用 double！

            cout << "浮點: " << val << endl;

        }

    }

    va\_end(args);

}

int main() {

    printMixed(4, 100, 3.14, 200, 2.71);

    return 0;

}

// 整數: 100

// 浮點: 3.14

// 整數: 200

// 浮點: 2.71

# Lifetime(生命週期)

生命週期指物件(變數實體)在程式執行期間「從出現到消失」的時間範圍。生命週期跟你在哪裡宣告名字沒直接等號；同一個名字可指向不同生命週期(例如回傳指標指向 dynamic 物件)。C/C++(你目前寫 C++)常見幾種：

* Local variable(區域變數) → 每次呼叫都重設
* Static local variable(靜態區域變數) → 只初始化一次，值會保留
* Global variable(全域變數) → 整個程式期間都存在

## 範例：變數生命週期

#include <iostream>

using namespace std;

// 全域變數(Global Variable)：整個程式執行期間都存在

int globalVar = 0;

void showLifetime() {

    // 區域變數(Local Variable)：每次進來都會重新建立與銷毀

    int localVar = 0;

    // 靜態變數(Static Local Variable)：只初始化一次，值會保留

    static int staticVar = 0;

    cout << "localVar = " << localVar << ", staticVar = " << staticVar << ", globalVar = " << globalVar << endl;

    // 每次都遞增

    localVar++;

    staticVar++;

    globalVar++;

}

int main() {

    showLifetime(); // localVar = 0, staticVar = 0, globalVar = 0

    showLifetime(); // localVar = 0, staticVar = 1, globalVar = 1

    showLifetime(); // localVar = 0, staticVar = 2, globalVar = 2

    return 0;

}

# Scope

Scope 是「名稱(identifier)在原始碼中可被參考的區域」。屬於編譯期概念；與它在執行期是否存在無必然關係。常見作用域(scope)層級：

* 區塊(block scope)：大括號 {} 裡宣告
* 函式參數 scope：參數名稱在函式本體中有效
* 檔案/全域 scope(namespace scope)：在所有函式外宣告
* 類別 scope：class/struct 內宣告的成員
* 命名空間 scope：namespace os { ... }
* for 迴圈 init scope：for (int i=0; ...) 中的 i

# static

static 關鍵字用來控制

* 變數或函式的生命週期(lifetime)
* 變數或函式的可見範圍(scope)

## 範例：記住函式中的變數值(區域 static)

普通區域變數在函式結束後會消失；而 static 區域變數只初始化一次，會保留值到程式結束。

#include <iostream>

using namespace std;

void counter() {

    static int count = 0; // 初始化一次，不會每次重新設為 0

    count++;

    cout << "Count: " << count << endl;

}

int main() {

    counter(); // Count: 1

    counter(); // Count: 2

    counter(); // Count: 3

    return 0;

}

## 範例：變數封裝(檔案內可見)

使用 static 修飾的全域變數，只能在該檔案內使用，屬於OOP中的「模組封裝」

├── utils.cpp

└── test.cpp (主檔案)

// utils.cpp

#include <iostream>

using namespace std;

static int sensorValue = 0; // 只在這個檔案內部可見

void readSensor() {

    sensorValue++;

    cout << "Sensor value: " << sensorValue << endl;

}

// test.cpp

#include <iostream>

using namespace std;

extern void readSensor();

int main() {

    readSensor(); // Sensor value: 1

    readSensor(); // Sensor value: 2

    return 0;

}

## 範例：函數封裝

static 函式只能在該檔案中使用，通常用來避免名稱衝突與限制存取。

├── utils.cpp

└── test.cpp (主檔案)

// utils.cpp

#include <iostream>

using namespace std;

static int add(int a, int b) { // 只能在本檔案使用

    return a + b;

}

void printSum(int x, int y) {

    cout << "Sum = " << add(x, y) << endl;

}

// test.cpp

#include <iostream>

using namespace std;

// extern int add(int, int); ❌ 錯誤，add 是 static

extern void printSum(int, int);

int main() {

    printSum(3, 4); // Sum = 7

    return 0;

}

## 範例：class中的 static 成員變數

static 成員變數屬於整個類別，不屬於某個物件。常用來計算物件數量、共享資源等

#include <iostream>

using namespace std;

class MyClass {

public:

    static int count; // 所有物件共享一份資料

    MyClass() {

        count++;

    }

    void showCount() {

        cout << "Count: " << count << endl;

    }

};

// 類別外定義 static 成員變數

int MyClass::count = 0;

int main() {

    MyClass a, b, c;

    c.showCount(); // Count: 3

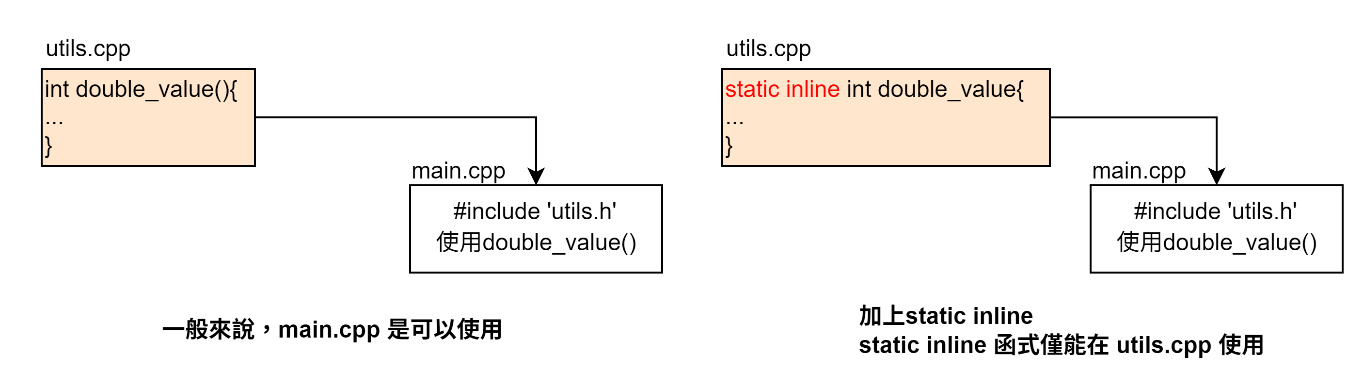
    return 0;

}

# static inline

* inline 建議編譯器展開
* static 表示這個函式只在本檔案中有效(內部連結)

## 範例 ：單一檔案中宣告與使用 static inline 函式



// utils.cpp

#include <iostream>

static inline int double\_value(int x) {

    return x \* 2;

}

int main() {

    int result = double\_value(10);  // 使用 static inline 函式

    std::cout << "結果為: " << result << std::endl;

    return 0;

}

## 錯誤寫法：謹寫static沒寫inline

// math\_utils.h

static int triple(int x) {

    return x \* 3;

}

如果沒寫inline，如果被多個 .cpp include，會重複產生symbol，但因沒inline，可能不會被編譯器優化，會出現「多重定義錯誤」。

# extern

用於跨檔案變數共享。extern 告訴編譯器：變數的定義在其他地方(通常在其他 .cpp 檔案中)。

├── utils.cpp

└── test.cpp (主檔案)

// utils.cpp

int counter = 42;  // 實際定義變數

// test.cpp

#include <iostream>

using namespace std;

extern int counter;  // 告訴編譯器：變數定義在別處

int main() {

    cout << "Counter = " << counter << endl; // Counter = 42

    return 0;

}

# 其他練習題目

## 陣列搜尋(Array Search)

|  |  |
| --- | --- |
| 題目描述  給定一個整數陣列和一個整數 key，請你回傳 key 在陣列中的索引位置。若找不到，則輸出 -1。 | |
| 輸入格式  第一行輸入一個整數 n(1 ≤ n ≤ 100)  第二行輸入 n 個整數，代表陣列內容  第三行輸入一個整數 key | 輸出格式  輸出 key 在陣列中的索引(從 0 開始)，若不存在則輸出 -1 |
| 範例輸入  6  3 4 6 2 8 10  2 | 範例輸出  3 |
| 程式碼 | |

## 判斷質數

|  |  |
| --- | --- |
| 題目描述  給定一個整數 N，請判斷它是否為質數。 | |
| 輸入格式  一行一個整數 N(1 ≤ N ≤ 10⁶) | 輸出格式  若為質數輸出 Yes，否則輸出 No |
| 範例輸入  10 | 範例輸出  No |
|  | |

## 區間內所有質數

|  |  |
| --- | --- |
| 題目描述  給定兩個整數 a 與 b，列出介於 [a, b) 區間內的所有質數。 | |
| 輸入格式  一行輸入兩個整數 a 與 b(0 ≤ a < b ≤ 10⁶) | 輸出格式  將所有質數以空白隔開，輸出在同一行 |
| 範例輸入  2 10 | 範例輸出  2 3 5 7 |
| 程式碼 | |