



# 06 函數與遞迴函數

Fundamental Computer Programming- C++ Lab(I)

元智大學 | C++ 程式設計實習

張維元

課程投影片：[請從元智個人 Portal 下載](#)

# C++ 程式設計實習

- 01 程式設計與開發環境
- 02 變數型態與運算
- 03 流程控制
- 04 陣列與向量
- 05 字元與字串
- 06 函數與遞迴函數
- 07 指標與引用
- 08 循序檔案和隨機檔案

2015



維元

(@v123582)

Web Development

Data Science

#遠端 #斜槓 #教學  
#資料科學 #網站開發

擅長網站開發與資料科學的雙棲工程師，熟悉的語言是 Python 跟 JavaScript。同時也是 資料科學家的工作日常 粉專及 資料科學家的 12 堂心法課 發起人，擁有多次國內大型技術會議講者經驗，持續在不同的平台發表對 #資料科學、#網頁開發 或 #軟體職涯 相關的分享。

- 元智大學 C++/CPE 程式設計課程 講師
- ALPHACamp 全端 Web 開發 / Leetcode Camp 課程講師
- CUPOY Python 網路爬蟲實戰研習馬拉松 發起人
- 中華電信學院 資料驅動系列課程 講師
- 工研院 Python AI 人工智慧資料分析師養成班 講師
- 華岡興業基金會 AI/Big Data 技能養成班系列課程 講師

site: v123582.tw / line: weiwei63

mail: weiyuan@saturn.yzu.edu.tw

2015



維元

(@v123582)

Web Development

Data Science

#遠端 #斜槓 #教學  
#資料科學 #網站開發

擅長網站開發與資料科學的雙棲工程師，熟悉的語言是 Python 跟 JavaScript。同時也是 資料科學家的工作日常 粉專及 資料科學家的 12 堂心法課 發起人，擁有多次國內大型技術會議講者經驗，持續在不同的平台發表對 #資料科學、#網頁開發 或 #軟體職涯 相關的分享。

- 2018 總統盃黑客松 冠軍隊伍
- 2017 資料科學愛好者年會(DSCONF) 講師
- 2017 行動科技年會(MOPCON) 講師
- 2016 微軟 Imagine Cup 台灣區冠軍

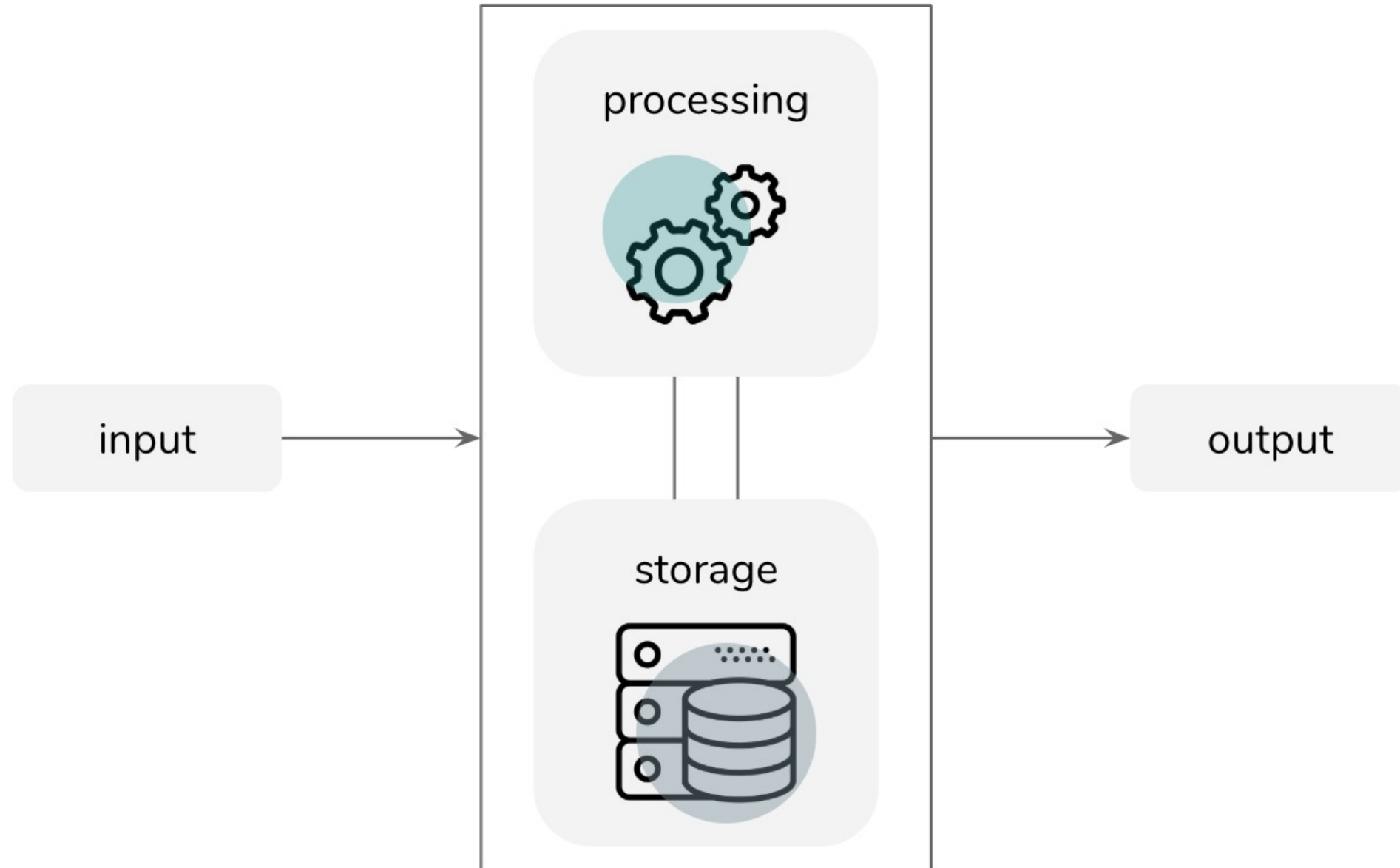
site: v123582.tw / line: weiwei63  
mail: weiyuan@saturn.yzu.edu.tw

# 什麼是程式？

程式語言是用來命令電腦執行各種作業的工具，是人與電腦溝通的橋樑。當電腦藉由輸入設備把程式讀入後，會儲存在主記憶體內，然後指令會依序被控制單元提取並解碼或翻譯成電腦可以執行的信號，並把信號送到各個裝置上，以執行指令所指派的動作。也就是說，人類與電腦溝通的語言稱為程式語言。

**程式 = 利用一系列的指令告訴電腦如何執行工作**





# 函數

## (Function)

函數用於程式碼的重複使用，定義參數與回傳值。可以將常用的程式碼進行封裝，透過呼叫重複調用。

一個函數有幾個部分組成：

- 函數名稱
- 回傳類型
- 傳入參數
- 函數主體

# 函數 (Function)

```
1  int max(int num1, int num2) {  
2      int result;  
3      if (num1 > num2)  
4          result = num1;  
5      else  
6          result = num2;  
7      return result;  
9  }  
10  
    int m = max(3, 5);
```



# 函數 (Function)

```
Function Name Parameters
1  int max(int num1, int num2) {
2      int result;
3      if (num1 > num2)
4          result = num1;
5      else
6          result = num2;
7
8      return result;
9  }
10      Return Value
      int m = max(3, 5);
              Arguments
```

# 函數 (Function)

```
1  int max(int num1, int num2) {  
2      int result;  
3      if (num1 > num2)  
4          result = num1;  
5      else  
6          result = num2;  
7      return result;  
8  }  
9  
10 int m = max(3, 5);
```

Define  
Declare

Block / Scope

Call / Invoke

# 函數 (Function)

```
1  int max(int num1, int num2) { ← ②
2      int result;
3      if (num1 > num2)
4          result = num1;
5      else
6          result = num2;
7
8      return result;
9  }
10 int m = max(3, 5); ← ①
```

The diagram illustrates the execution flow of a function call. Red arrows and numbered circles (①, ②, ③, ④) highlight key points:

- ①: Points to the function call `max(3, 5)` in the caller code.
- ②: Points to the opening curly brace of the function definition.
- ③: Points to the body of the function, specifically the `if` statement.
- ④: Points to the `return` statement, indicating the point where the function returns control to the caller.

# 函數 (Function)

```
1  int max(int num1, int num2) {  
2      int result;  
3      if (num1 > num2)  
4          result = num1;  
5      else  
6          result = num2;  
7      return result;  
8  }  
9  
10 int m = max(3, 5);
```

②

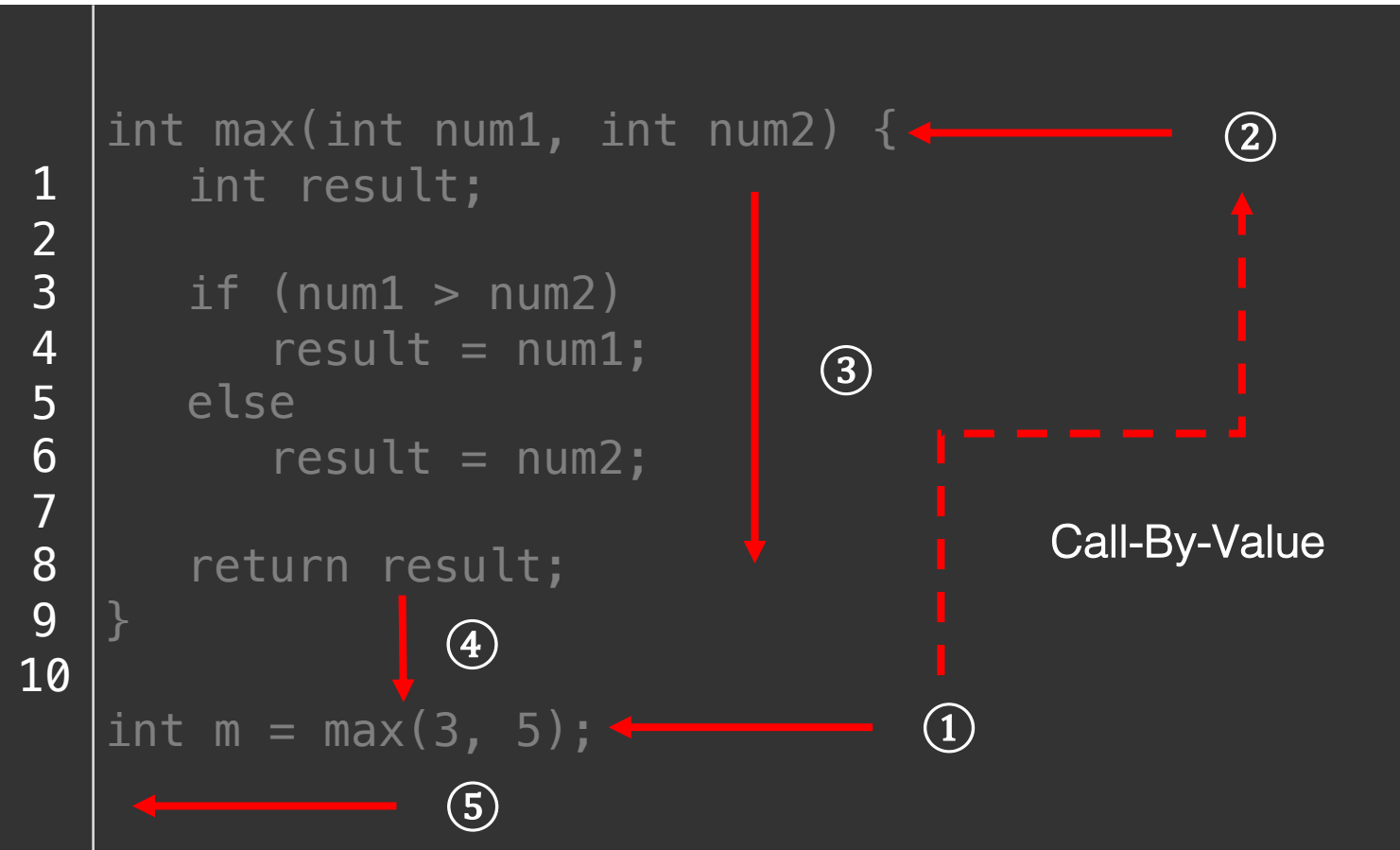
③

④

①

Call-By-Value

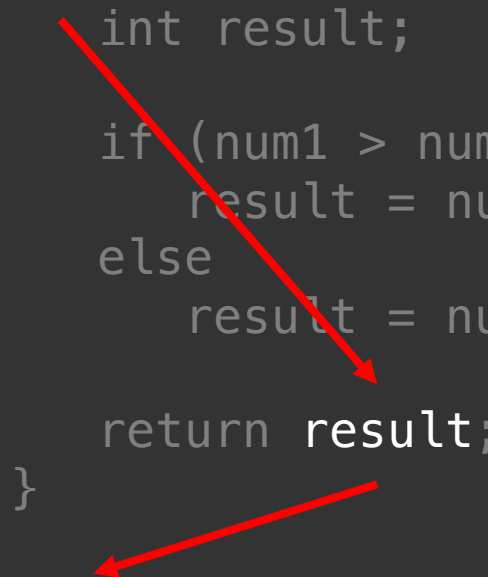
# 函數 (Function)



# 函數 (Function)

Return type

```
1  int max(int num1, int num2) {  
2      int result;  
3      if (num1 > num2)  
4          result = num1;  
5      else  
6          result = num2;  
7      return result;  
8  }  
9  
10 int m = max(3, 5);
```

A red arrow originates from the 'int' return type in the function signature on line 1 and points to the 'result' variable on line 2. Another red arrow originates from the closing brace of the function on line 9 and points to the 'max' function call on line 10.

- int, float, double
- void
- int \*

# Function

函數用於程式碼的重複使用，定義參數與回傳值。可以將常用的程式碼進行封裝，透過呼叫重複調用。

一般的函式特性：

- 用來封裝程式碼
- 有需要時才呼叫

# 函式的定義與呼叫

```
1  int max(int num1, int num2) {  
2      int result;  
3      if (num1 > num2)  
4          result = num1;  
5      else  
6          result = num2;  
7  
8      return result;  
9  }  
10  
    int m = max(3, 5);
```

Define  
Declare  
Call / Invoke



# 函式的宣告與定義

```
1  int max(int, int); ← 宣告 Declare  
2  int max(int num1, int num2) { (原型宣告prototype declaration)  
3      int result;  
4      ← 定義 Define  
5      if (num1 > num2)  
6          result = num1;  
7      else  
8          result = num2;  
9      return result;  
10 }  
  
int m = max(3, 5); ← 調用 / 呼叫
```

→ 函數的宣告跟定義可以分開，但要在宣告後才能調用<sup>17</sup>

# 作用域

```
1 #include <iostream>
2 using namespace std;
3 int m = 999; ← 全域變數
4
5 int foo() {
6     int m = 123; ← 區域變數
7     cout << "foo(): " << m << endl;
8 }
9
10 int main () {
11     cout << "main(): " << m << endl;
12     cout << foo() << endl;
13 }
```

# 作用域

```
1 #include <iostream>
2 using namespace std;
3 int m = 999; ← 全域變數
4
5 int foo() {
6     int m = 123; ← 區域變數
7     cout << "foo(): " << m << endl;
8 }
9
10 int main () {
11     cout << "main(): " << m << endl;
12     cout << foo() << endl;
13 }
```

→ 函數內宣告的變數稱為區域變數，會在函式結束時消失

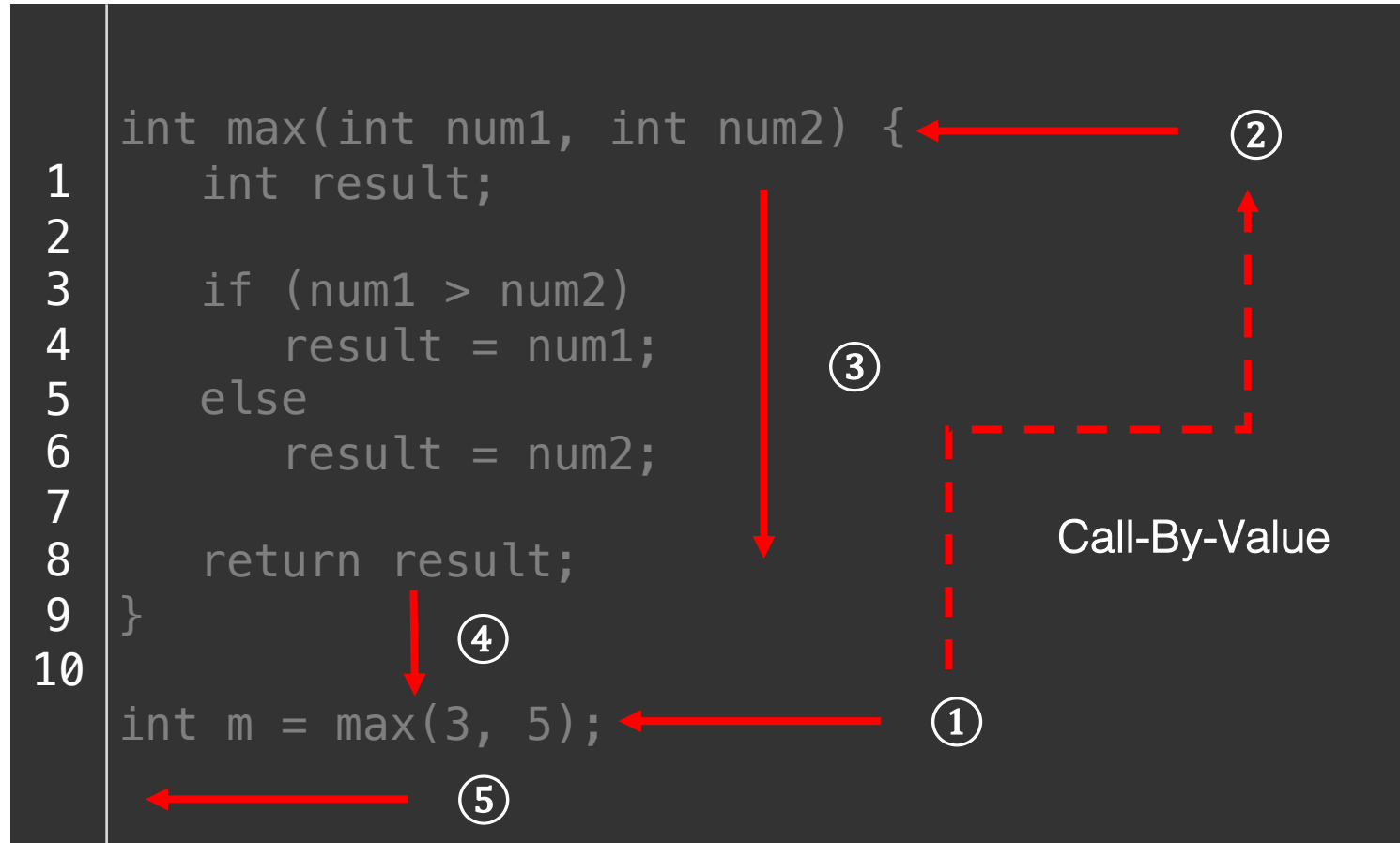
# 作用域

```
1 #include <iostream>
2 using namespace std;
3
4 int m = 999;
5 int foo() {
6     int m = 123;
7     return m;
8 }
9
10 int main (){
11     cout << foo() << endl;
12 }
```

```
1 #include <iostream>
2 using namespace std;
3
4 int m = 999;
5 int foo() {
6     int m = 123;
7     return ::m;
8 }
9
10 int main (){
11     cout << foo() << endl;
12 }
```

# 傳值的方式

- call-by-value
- call-by-reference
- call-by-address
- call-by-value/result
- call-by-name



# call-by-value

```
1 int f(int n) { ← ②
2
3     n = 10;
4     return n;
5 }
6
7 int a = 10;
8 cout << a << endl;
9 f(a); ← ①
10 cout << a << endl;
```

Call-By-Value

*int n = a;*

# call-by-address

```
1 int f(int *nPtr) {  
2  
3     *nPtr = 10;  
4     return n;  
5 }  
6  
7 int a = 10;  
8 cout << a << endl;  
9 f(&a);  
10 cout << a << endl;
```

- \* 依址取值運算子，表示後面的變數要存的是一個位置
- & 取址運算子，用來取出後面變數的位置

# call-by-reference

```
1 int f(int &nRef) {  
2  
3     nRef = 10;  
4     return n;  
5 }  
6  
7 int a = 10;  
8 cout << a << endl;  
9 f(a);  
10 cout << a << endl;
```

- \* 依址取值運算子，表示後面的變數要存的是一個位置
- & 取址運算子，用來取出後面變數的位置

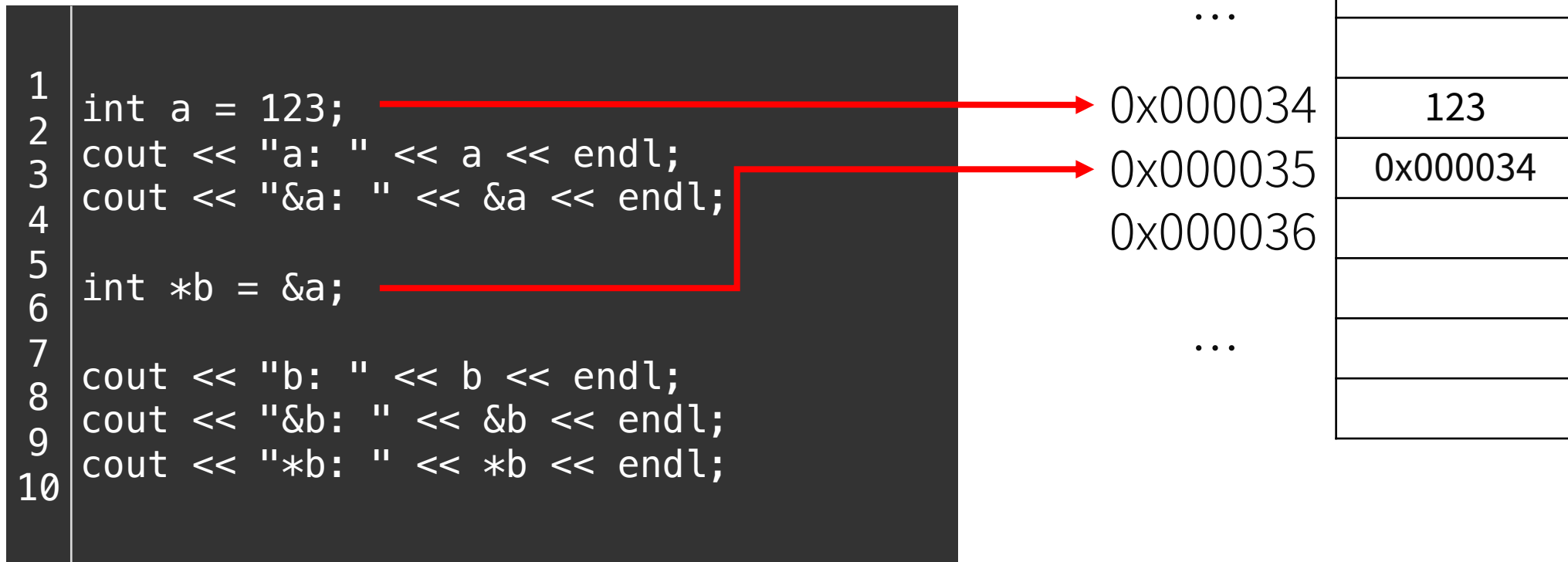


# call-by-value

```
1 int f(n) {  
2     n = 10;  
3     return n;  
4 }  
5  
6  
7 int a = 10;  
8 cout << a << endl;  
9 a = f(a);  
10 cout << a << endl;
```

# 「\*」和「&」運算子

- \* 依址取值運算子，表示後面的變數要存的是一個位置
- & 取址運算子，用來取出後面變數的位置



# 「\*」和「&」運算子

- \* 依址取值運算子，表示後面的變數要存的是一個位置
- & 取址運算子，用來取出後面變數的位置

```
1 int a = 123;
2 cout << "a: " << a << endl;
3 cout << "&a: " << &a << endl;
4
5 int *b = &a;
6
7 cout << "b: " << b << endl;
8 cout << "&b: " << &b << endl;
9 cout << "*b: " << *b << endl;
10
```

...

0x000034

0x000035

0x000036

...

← b 存放的數值是一個位址  
← b 變數的位址  
← 位址 b 存放的數值

123
0x000034

# 變數修飾

- inline
- static
- extern
- const
- volatile
- restrict
- ~~auto~~
- register

# inline function

```
1 #include <iostream>
2 using namespace std;
3
4 inline int max(int a, int b){
5     return a > b ? a : b;
6 }
7
8 int main(){
9     cout << max(55, 22) << endl;
10    cout << max(2, 214) << endl;
11
12    return 0;
12 }
```

```
1
2
3 #include <iostream>
4 using namespace std;
5
6 int main(){
7     cout << 55 > 22 ? 55 : 22 << endl;
8     cout << 2 > 214 ? 2 : 214 << endl;
9
10    return 0;
11 }
12
12
```

→ inline 會將簡單的函式在編譯時代入

# static

```
1 #include <iostream>
2 using namespace std;
3 void count(){
4     static int c = 1;
5     cout << c << endl;
6     c++;
7 }
8
9 int main(){
10     for(int i = 0; i < 10; i++) {
11         count();
12     }
12     return 0;
}
```

→ static 可以保留變數在同一個 function 使用

# extern

```
1
2
3
4 // a.cpp
5
6 int a = 123;
7 float b = 0.456;
8 // double c = a + b;
9 static double c = a + b;
10
11
12
13
```

```
1 // b.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main(void) {
6
7     extern int a;
8     extern float b;
9     // extern double c ;
10
11     cout << a << endl;
12     cout << b << endl;
13     cout << c << endl;
14 }
```

→ extern 可以存取其他的程式，static 可以限定變數的範圍

# const

```
1
2 #include <iostream>
3 using namespace std;
4
5 int main(void) {
6     const int a = 123;
7     // a = 123;
8     cout << a << endl;
9 }
10
11
12
12
```

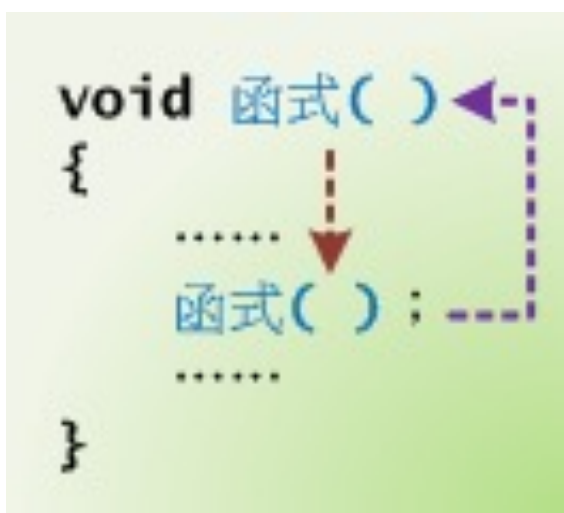
```
1
2
3
4 #include <iostream>
5 #define a 123
6 using namespace std;
7
8 int main(void) {
9     // a = 123;
10    cout << a << endl;
11 }
12
12
12
```

→ const 表示變數是不可修改 (immutable) 的內容

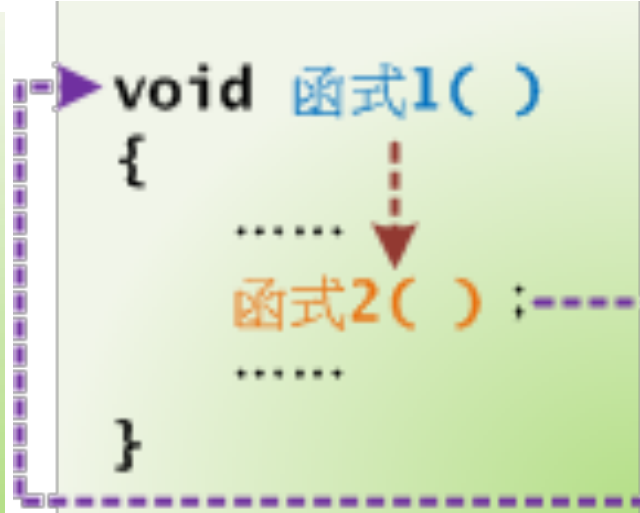


# 遞迴函式 (Recursion)

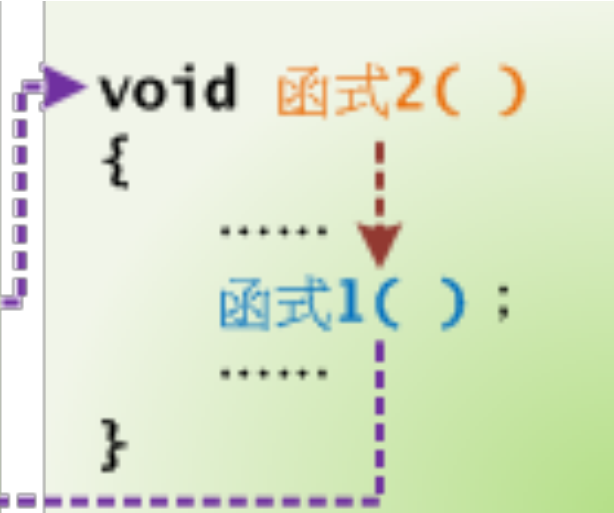
Function 當中含有自我呼叫（self-calling calling）敘述存在，稱為是遞迴函式。



直接遞迴



間接遞迴



尾端遞迴

# 遞迴演算法則的設計原則

```
1 function foo(parameters) {  
2  
3  
4     if (Base Case) ← // 達到終止條件時結束  
5         return 結果;    遞迴，需要時回傳結果  
6     else  
7         General Case; ← // 重複執行，遞迴呼叫  
8  
9 }  
10
```

# 遞迴的例子

$$n! = f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * f(n-1) & \text{if } n \geq 1 \end{cases}$$

```
1 int f(int n){  
2  
3     cout << "call f(), n = " << n << endl;  
4     if( n == 0 )  
5         return 1;  
6     int ret = n*f(n-1);  
7     cout << "return n*f(n-1) = " << ret << endl;  
8     return ret;  
9 }  
10
```

$$\begin{aligned} 4! &= 4 * 3! \\ &= 4 * 3 * 2! \\ &= 4 * 3 * 2 * 1! \\ &= 4 * 3 * 2 * 1 \end{aligned}$$

# 遞迴的例子

$$n! = f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * f(n-1) & \text{if } n \geq 1 \end{cases}$$

← Base case

← General case

```
1 int f(int n){
2
3     cout << "call f(), n = " << n << endl;
4     if( n == 0 )
5         return 1;
6     int ret = n*f(n-1);
7     cout << "return n*f(n-1) = " << ret << endl;
8     return ret;
9 }
10
```

$$4! = 4 * 3!$$

$$= 4 * 3 * 2!$$

$$= 4 * 3 * 2 * 1!$$

$$= 4 * 3 * 2 * 1$$

# 遞迴的例子

$$n! = f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * f(n-1) & \text{if } n \geq 1 \end{cases}$$

← Base case

← General case

```
1 int f(int n){
2
3     cout << "call f(), n = " << n << endl;
4     if( n == 0 )
5         return 1;
6     int ret = n*f(n-1);
7     cout << "return n*f(n-1) = " << ret << endl;
8     return ret;
9 }
10
```

$$\begin{aligned} 4! &= f(4) \\ &= 4 * f(3) \\ &= 4 * 3 * f(2) \\ &= 4 * 3 * 2 * f(1) \\ &= 4 * 3 * 2 * 1 * f(0) \\ &= 4 * 3 * 2 * 1 * 1 \end{aligned}$$

# 遞迴的例子

$$n! = f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * f(n-1) & \text{if } n \geq 1 \end{cases}$$

← Base case  
← General case

```
1 int f(int n){
2
3     cout << "call f(), n = " << n << endl;
4     if( n == 0 ) ← Base case
5         return 1;
6     int ret = n*f(n-1); ← General case
7     cout << "return n*f(n-1) = " << ret << endl;
8     return ret;
9 }
10
```

$$\begin{aligned} 4! &= f(4) \\ &= 4 * f(3) \\ &= 4 * 3 * f(2) \\ &= 4 * 3 * 2 * f(1) \\ &= 4 * 3 * 2 * 1 * f(0) \\ &= 4 * 3 * 2 * 1 * 1 \end{aligned}$$

# 遞迴的例子

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

```
1
2
3 int f(int n){
4
5     if( n <= 1 )
6         return 1;
7     return f(n-1) + f(n-2);
8 }
9
10
```

Fibonacci 為 1200 年代的歐洲數學家，在他的著作中曾經提到：「若有一隻兔子每個月生一隻小兔子，一個月後小兔子也開始生產。起初只有一隻兔子，一個月後就有兩隻兔子，二個月後有三隻兔子，三個月後有五隻兔子（小兔子投入生產）…」。

# 遞迴的例子

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

← Base case

← General case

```
1  
2  
3 int f(int n){  
4  
5     if( n <= 1 )  
6         return 1;  
7     return f(n-1) + f(n-2);  
8 }  
9  
10
```

1  
1  
1 + 1 = 2  
2 + 1 = 3  
2 + 3 = 5  
5 + 3 = 8  
...



# 遞迴的例子

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

← Base case

← General case

```
1
2
3 int f(int n){
4
5     if( n <= 1 ) ← Base case
6         return 1;
7     return f(n-1) + f(n-2); ← General case
8 }
9
10
```

1  
1  
1 + 1 = 2  
2 + 1 = 3  
2 + 3 = 5  
5 + 3 = 8  
...

# How Recursion Works

- 要保存 Function 當時執行的狀況，即 Push 資料到 Stack 中。
  - 參數值 (Parameter)
  - 區域/暫存變數值 (Local Variable)
  - 返回位址 (Return Address)
- 控制權轉移到另一個 Function
- Recursion 動作結束時，要 Pop Stack 取出資料，然後返回前一個 Function

# How Recursion Works

- 要保存 Function 當時執行的狀況，即 Push 資料到 Stack 中。
    - 參數值 (Parameter)
    - 區域/暫存變數值 (Local Variable)
    - 返回位址 (Return Address)
  - 控制權轉移到另一個 Function
  - Recursion 動作結束時，要 Pop Stack 取出資料，然後返回前一個 Function
- 大幅佔用記憶體空間

# 遞迴的例子

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

```
1  
2  
3 int f(int n){  
4  
5     if( n <= 1 )  
6         return 1;  
7     return f(n-1) + f(n-2);  
8 }  
9  
10
```

```
1  
2 int f(int n) {  
3     int fib[n+2];  
4     fib[0] = 1;  
5     fib[1] = 1;  
6     for (int i = 2; i <= n; i++)  
7         fib[i] = fib[i-1] + fib[i-2];  
8     return fib[n];  
9 }  
10
```

Thanks for listening.

元智大學 | C++ 程式設計實習

Wei-Yuan Chang