

程式設計 (**Programming**)

真理大學 資訊工程系 吳汶涓老師

CH05

函式 (function)



本章綱要

5-1 簡介

5-2 C語言中的程式模組

5-3 數學函式庫

5-4 函式

5-5 函式定義

5-6 函式原型

5-7 函式呼叫堆疊與活動紀錄

5-8 標頭

5-9 呼叫函式

5-10 亂數產生

5-11 範例：機會遊戲

5-12 儲存類別

5-13 範圍規則

5-14 遞迴

5-15 Fibonacci級數

5-16 遞迴 vs. 迭代

5.12 儲存類別

■ 識別字做為變數名稱、函式名稱

- 宣告名稱、型別、大小(值)

```
int counter = 0;
```

- 儲存類別：有 **auto, register, extern, static** 四種
- **佔用期間**：指該識別字存在記憶體中的時間
- **範圍(scope)**：指該識別字能夠被參考的範圍

■ 自動儲存

- **自動變數**在程式進入宣告他們的區塊或函式時會產生，而當程式離開此區塊或函式時，便會清除該變數。
- **auto**: **區域變數**預設為自動變數 (**auto** double x, y;)
- **register**: 建議將此變數放到電腦的硬體暫存器中，會加快執行速度 (**register** int counter = 1;)



增進效能的小技巧 5.1

自動儲存是節省記憶體的一種方法，因為自動變數只有在需要他們時才會存在。自動變數在程式進入函式時才會產生，而當程式離開此函式時，他們便會清除。



增進效能的小技巧 5.2

如果經常使用到的變數 (如計數器) 可以在硬體暫存器中操作，便可以去除重複地將變數由記憶體中載入到暫存器，以及重複地將結果存回記憶體。

使用範圍

使用範圍

```
3  #include <stdio.h>
5  void useLocal( void );
9  int x = 1;
10
12 int main( void )
13 {
14     int x = 5;
16     printf("local x in main is %d\n", x );
26     useLocal();
34     return 0;
35 }
38 void useLocal( void )
39 {
40     int x = 25;
42     printf( "local x in useLocal is %d \n", x );
45 }
```

全域變數 (global variable)

區域變數 (local variable)

區域變數 (local variable)

1

X

5

X

25

X

■ 靜態儲存

- 在程式的執行過程中一直都存在的變數
- 程式開始執行時，就配置好記憶體和**設好初值 0**
- **static**: 定義在函式中的區域變數
 - **離開函式後，還保有這些變數的值**
 - 只能在定義它的函式中使用
(**static** int counter = 1;)
- **extern**: 全域變數和函式名稱設為extern
 - 任何函式中都可以使用
 - **全域變數**：是將變數的宣告放在任何函式定義之外



軟體工程的觀點 5.12

只在某個函式內所使用的變數應宣告為此函式的區域變數，而不要將它宣告為全域變數。

5.13 範圍規則

■ 檔案範圍(file scope)：

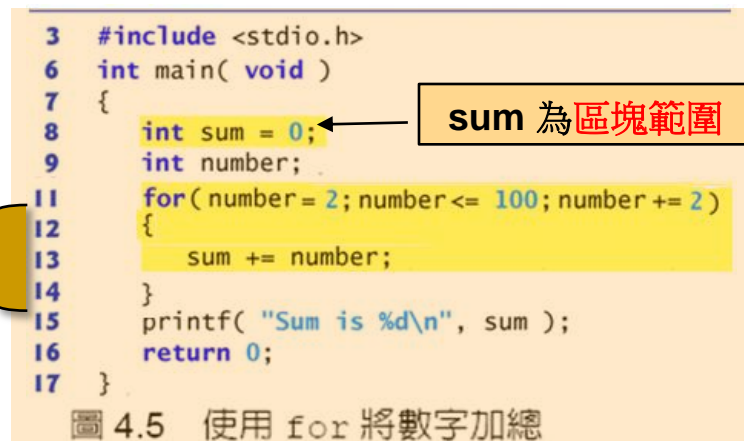
- 定義在函式外的識別字，所有函式都知道它的存在
- 全域變數、函式定義、函式原型都具有檔案範圍

■ 函式範圍(function scope)：

- 標籤是唯一具有函式範圍的識別字
- 標籤：識別字加冒號 (start:)

■ 區塊範圍(block scope)：

- 只能在該變數宣告的**區塊範圍**(`{ ... }`)中使用
- 若是碰到外層區塊的變數名稱與內層區塊的變數名稱相同時，外層的變數將會被**隱藏起來**，使用的是內層的變數。



```
3  #include <stdio.h>
6  int main( void )
7  {
8      int sum = 0;
9      int number;
11     for( number = 2; number <= 100; number += 2 )
12     {
13         sum += number;
14     }
15     printf( "Sum is %d\n", sum );
16     return 0;
17 }
```

Annotations in the image:

- A box labeled "sum 為**區塊範圍**" points to the declaration of `int sum = 0;` on line 8.
- A bracket labeled "區塊範圍" spans the code block from line 11 to line 17.

圖 4.5 使用 for 將數字加總

```

3 #include <stdio.h>
5 void useLocal( void );
6 void useStaticLocal( void );
7 void useGlobal( void );
9 int x = 1; /* global variable */
10
12 int main( void )
13 {
14     int x = 5; /* local variable to main */
16     printf("local x in outer scope of main is %d\n", x );
18     {
19         int x = 7;
21         printf( "local x in inner scope of main is %d\n", x );
22     }
24     printf( "local x in outer scope of main is %d\n", x );
26     useLocal();
27     useStaticLocal();
28     useGlobal();
29     useLocal();
30     useStaticLocal();
31     useGlobal();
32
33     printf( "\nlocal x in main is %d\n", x );
34     return 0;
35 }

```

具檔案範圍的全域變數

具區域範圍的變數

具區域範圍的變數

local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in main is 5

圖 5.12 範圍的例子

注意

課本pp. 5-32

```

38 void useLocal( void )
39 {
40     int x = 25; ← 具區域範圍的變數
42     printf( "local x in useLocal is %d after entering useLocal\n", x );
43     x++;
44     printf( "local x in useLocal is %d before exiting useLocal\n", x );
45 }

```

```

50 void useStaticLocal( void )
51 {
53     static int x = 50; ← 具區域範圍的靜態變數
55     printf( "local static x is %d on entering useStaticLocal\n", x );
56     x++;
57     printf( "local static x is %d on exiting useStaticLocal\n", x );
58 }

```

```

61 void useGlobal( void )
62 {
63     printf( "global x is %d on entering useGlobal\n", x );
64     x *= 10;
65     printf( "global x is %d on exiting useGlobal\n", x );
66 }

```

first call

local x in useLocal is 25 after entering useLocal
 local x in useLocal is 26 before exiting useLocal

local static x is 50 on entering useStaticLocal
 local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
 global x is 10 on exiting useGlobal



second call

local x in useLocal is 25 after entering useLocal
 local x in useLocal is 26 before exiting useLocal

local static x is 51 on entering useStaticLocal
 local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
 global x is 100 on exiting useGlobal

5.14 遞迴 (recursive)

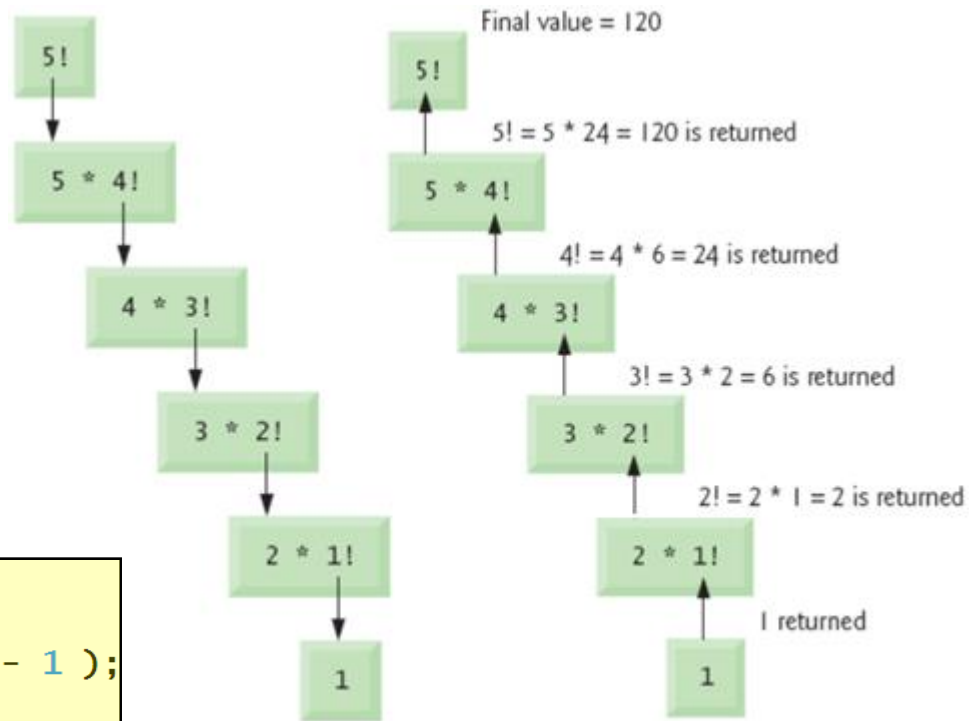
■ 遞迴函式：

- 呼叫自己的函式 (**recursive function**)
- 只曉得如何解決**基本情況 (base case)**
- 將問題分解成
 - 知道該怎麼作的部分
 - 不知道該怎麼作的部分
 - 類似原來的問題
 - 函式呼叫它自己來解決不知道該怎麼作的部分
- 最後基本情況會獲得解決
 - 解決完基本問題後，可以一路往上延伸，解決整個問題
- 範例：n階乘的計算, $n! = n * (n-1) * (n-2) * \dots * 1 = n * (n-1)!$
base case 為 $1! = 1, 0! = 1$

■ 階乘

- $5! = 5 * 4 * 3 * 2 * 1$
- 我們注意到
$$5! = 5 * 4!$$
$$4! = 4 * 3! \dots$$
- 可以用遞迴方式計算階乘
(遞迴呼叫 recursive call)

```
int factorial(int num){  
    ...  
    return num * factorial( num - 1 );  
}
```



(a) 遞迴呼叫的進行

(b) 每一個遞迴呼叫回傳的值


- 解決基本問題後，開始回傳結果
($1! = 1$, $0! = 1$)

課本pp. 5-36

```

1  /* Recursive factorial function */
3  #include <stdio.h>
5  long factorial( long number );
6
8  int main( void )
9  {
10     printf( "%2d! = %ld\n", i, factorial(10) );
14     printf( "%2d! = %ld\n", i, factorial(5) );
18     return 0;
19 }
20
22 long factorial( long number )
23 {
24     /* base case */
25     if ( number <= 1 ) {
26         return 1;
27     }
28     else { /* recursive step */
29         return ( number * factorial( number - 1 ) );
30     }
31 }

```


 10! = 3628800
 5! = 120

基本情況的解決
 遞迴呼叫解決不知道怎麼作的部分

圖 5.14 以遞迴函式計算階乘



常見的程式設計錯誤 5.13

於一個遞迴函式中，忘了在必須回傳值的位置回傳一個數值。

課本pp. 5-37

5.15 使用遞迴的例子: Fibonacci級數

■ Fibonacci 級數: 0, 1, 1, 2, 3, 5, 8, ...

- 每個數都是它的前兩個Fibonacci數之和
- 可以用遞迴來解決

`fib(n) = fib(n-1) + fib(n-2)`

`fib(0) = 0`

`fib(1) = 1`

基本情況的解決

- `fib`函式的程式碼

```
long fin(long n)
{
    if(n==0 || n==1)
        return n;
    else
        return fib(n-1) + fin(n-2);
}
```

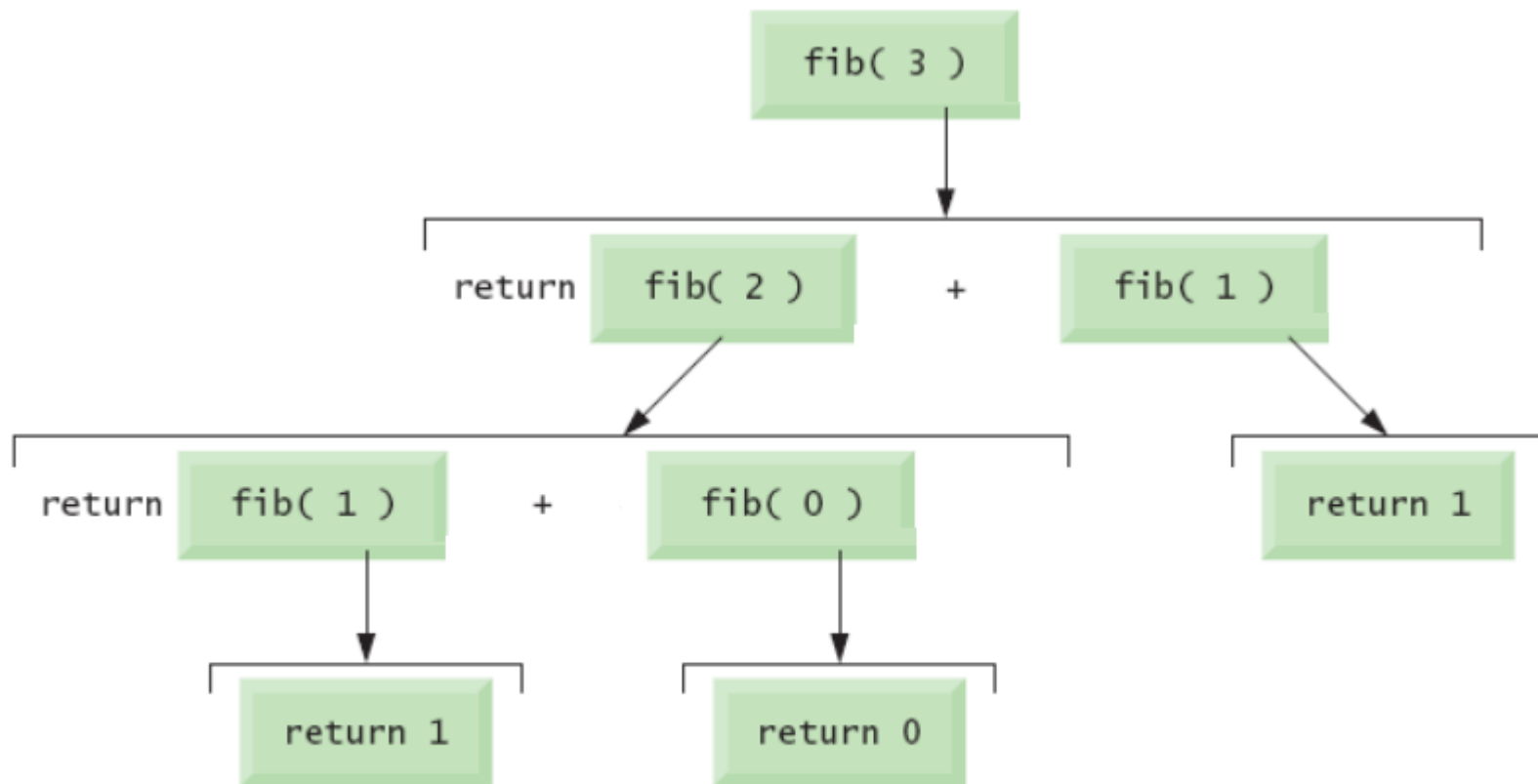


圖 5.16 呼叫 fibonacci(3) 的遞迴呼叫

```

3  #include <stdio.h>
5  long fibonacci( long n );
8  int main( void )
9  {
10     long result;
11     long number;
14     printf( "Enter an integer: " );
15     scanf( "%ld", &number );
16
18     result = fibonacci( number );
21     printf( "Fibonacci( %ld ) = %ld\n", number, result );
22     return 0;
23 }
25
26 long fibonacci( long n )
27 {
29     if ( n == 0 || n == 1 ) { /* base case */
30         return n;
31     }
32     else { /* recursive step */
33         return fibonacci( n - 1 ) + fibonacci( n - 2 );
34     }
35 }

```

Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

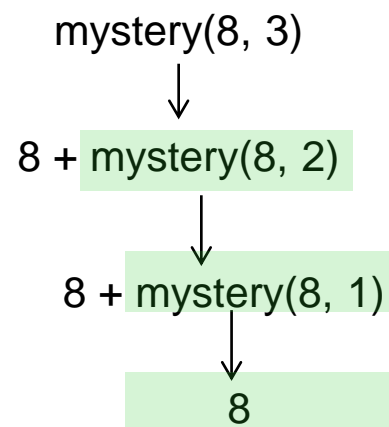
Enter an integer: 10
Fibonacci(10) = 55

圖 5.15 遞迴產生 Fibonacci 數

練習

- 請問下列程式結果為何？ (x = 8, y = 3)

```
1  #include <stdio.h>
2
3  int mystery( int a, int b );
6  int main( void )
7  {
8      int x;
9      int y;
11     printf( "Enter two integers: " );
12     scanf( "%d%d", &x, &y );
14     printf( "The result is %d\n", mystery( x, y ) );
16     return 0;
18 }
19
22 int mystery( int a, int b )
23 {
25     if ( b == 1 ) { /* base case */
26         return a;
27     }
28     else { /* recursive step */
29         return a + mystery( a, b - 1 );
30     }
32 }
```



課本pp. 5-63, ex. 5.43

5.16 遞迴與迭代

■ 重複性

- 迭代：明確的迴圈次數
- 遞迴：重複呼叫函式

■ 終止檢測

- 迭代：迴圈條件不符合
- 遞迴：變成基本狀況

■ 都可能造成無窮迴圈

■ 平衡

- 在效能(迭代)和良好的軟體工程(遞迴)之間作取捨

- 任何可用遞迴解決的問題，也一定可以用迭代（非遞迴）來解決。當使用遞迴方式較能夠自然地反映出問題，且能讓程式較容易瞭解時，我們便選用遞迴法，而不要用迭代法。另外一個選擇遞迴的理由是：迭代不是顯而易見的。



增進效能的小技巧 5.4

請避免如 Fibonacci 的遞迴程式，因為它會造成函式呼叫次數如指數般地迅速成長。

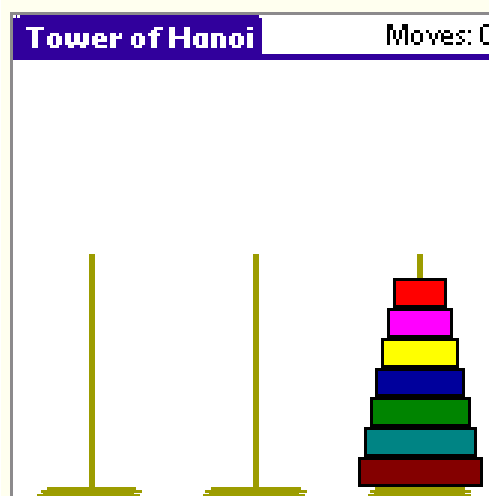


增進效能的小技巧 5.5

在要求效率的情況下，請避免使用遞迴。遞迴呼叫會花費很多時間和佔用額外的記憶體。

練習

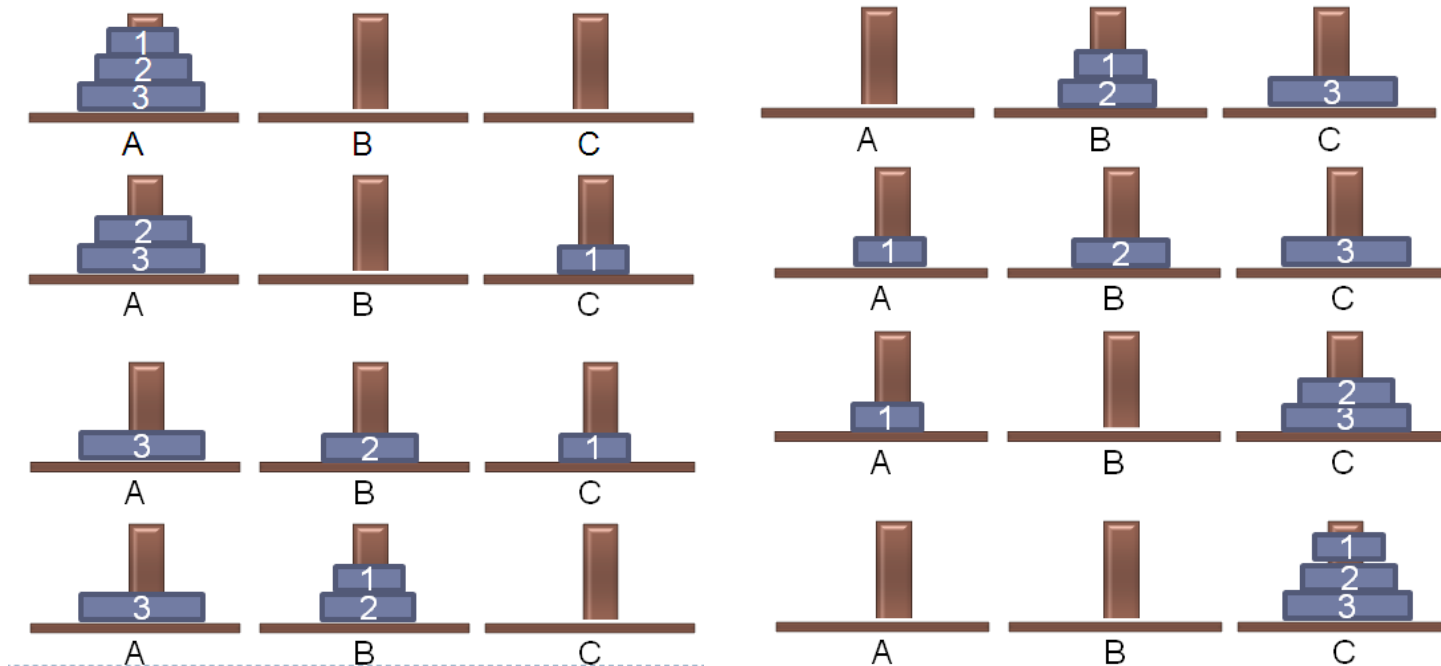
- 試撰寫一程式，將fibonacci的遞迴程式改成非遞迴的方式。
(迭代迴圈方式)
- (Hanoi塔) 撰寫一程式，將n個由大到小疊在一根柱子上的碟子，依序搬到另一根柱子中，搬移過程須遵照規則並可利用另一柱子。
 - 一次移動一個碟子
 - 小碟子壓大碟子



課本pp. 5-61, ex. 5.36

河內塔(Hanoi Tower)

- 河內塔問題：在A、B、C三個塔上有數個圓盤，請求出將圓盤從A塔全數搬移到C塔的順序，且大的圓盤不可以疊到小圓盤上。



練習

- 撰寫一個程式，遞迴計算指數。
函式 `integerPower(base, exponent)`，它可回傳 $\text{base}^{\text{exponent}}$ ，遞迴時可利用以下關係式

$$\text{base}^{\text{exponent}} = \text{base} * \text{base}^{\text{exponent}-1}$$

當 `exponent` 為1時，則是基本問題($\text{base}^1 = \text{base}$)

