

Chapter8-函數

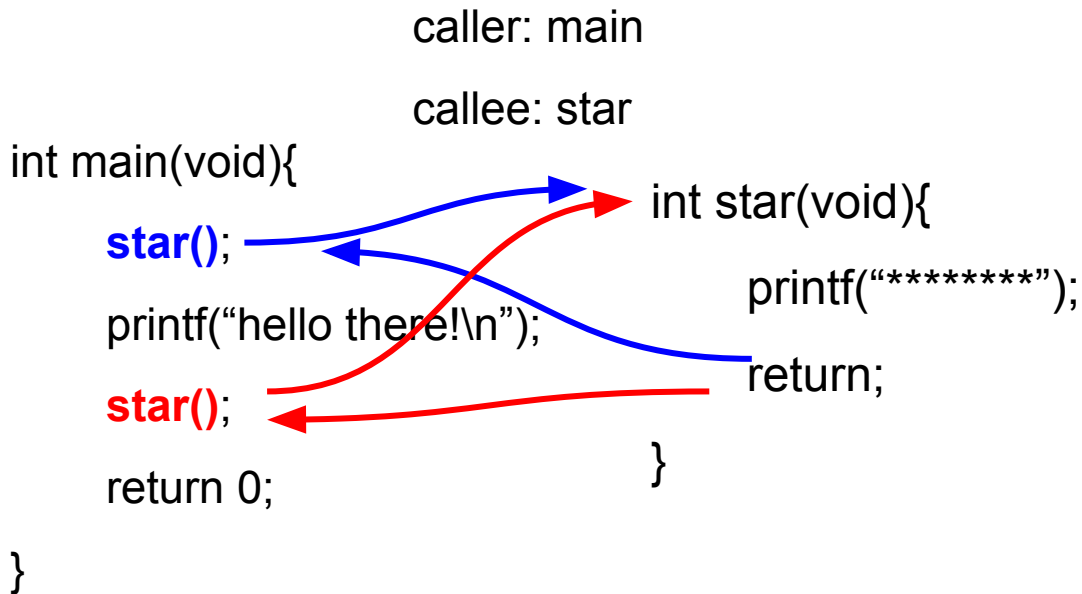
函數的目的

- 所有的 C 程式碼都是由函數所組成的
- 函數最大的目的是可以利用 **"模組化"** 的方式來簡化主程式
- 模組化優點:
 - 可以讓每一塊模組**完成各自特定的任務**
 - 可以減少重複寫程式碼
 - 較好管理



使用函數的程式執行順序

- 呼叫函式者: caller
- 被呼叫的函式: callee
- 程式執行權從 caller 到 callee 的過程會先儲存 caller 的資料後再切換到 callee, 這個準備過程與切換稱為 “function call”



函數的宣告 function declaration (aka prototype)

- 如同變數需要宣告, 使用函數前也須先告知編譯器
- 函數名稱命名規則與變數的命名規則一樣, 也不能使用 C 的關鍵字

語法:

回傳值型別 函數名稱(引數型別1, 引數型別2,引數型別n) ;

- 可以有多个引數, 但只能有一个回傳值

`int add(int, int) ;` // 輸入兩個為 int 數, 並回傳兩數和

函數的宣告 function declaration (aka prototype) (cont')

- 若不需要回傳值可以用 void 型別
- 同樣地如果不需要引數也可以用 void 型別

```
void star(void) ;
```

函數的定義 function definition

- 可以放在程式中的任一位置
- 若定義放在 caller 後須在 caller 前放宣告
- 若定義放在 caller 前則不須在 caller 前放宣告

語法:

```
回傳值型別 函數名稱(型別1 引數1, 型別2 引數2, ..... 型別n 引數n){  
    敘述主體 ;  
    return 運算式 ;  
}
```

函數的定義 function definition (cont')

回傳值型別 函數名稱(型別1 引數1, 型別2 引數2, 型別n 引數n){

敘述主體 ; // 可有可無

return 運算式 ;

}

```
int add (int num1, int num2){
```

```
    int a = num1 + num2;
```

```
    return a;
```

```
}
```

Return 敘述的作用

- return 用來回傳該函數的結果, return 的值必須含函數的回傳值型別相同
- 一旦執行 return 函數就不會繼續執行
- 若回傳型別為 void , return 敘述可省略, 但可以用 return 來當作提前中斷函數的一個方式

```
int  add (int num1, int num2){  
    int a = num1 + num2;  
    return a;  
}
```


在程式裡呼叫函數

- 呼叫函數的用法為, 輸入函數名稱後面加左右小括號, 並給參數(如果有的話)

`add(5, 10);` // 上一頁投影片的函數 `add`

- 若函數有回傳值可以搭配 `assign` 運算子來讓回傳值給某個變數接收

`int a;`

`a = add(5, 10);`

- 若不想接收回傳值會沒有回傳值的話, 就不用 `assign` 運算子以及用變數接收

利用函數來簡化程式結構 - 模組化

$$C_n^m = \frac{m!}{n!(m-n)!}$$

/lecture4/combination.c

區域、全域與靜態變數

- 在 C 語言中可以根據**活動範圍 (scope)** 與 **生命週期 (life cycle)** 來將變數分為**區域變數 (local variable)**、**全域變數 (global variable)** 與**靜態變數 (static variable)**
- 生命週期(life cycle): 變數內容 (記憶體裡的內容) 保留在記憶體的時間
- 活動範圍(scope): 可以使用變數名稱對照某記憶體內容的程式範圍

區域變數

- 若是在 block statement (由左右大括號夾住 statement 所形成的) 裡, 或是在函數的參數列裡面所宣告的變數, 若無特別指定, 則預設為**區域變數 (local variable)**
- 區域變數的 scope 以及 lifecycle 始於宣告該變數的地方, 終於該 block statement 的右大括號

區域變數 - 例子

```
int fac(int);  
int main(void){
```

```
    int ans;  
    ans = fac(5);  
    return 0;  
}
```

區域變數
ans 的
scope

```
int fac(int n){  
    int total = 1;  
    for(int i = 1 ; i <= n ; i++){  
        total *= i;  
    }  
    return total;  
}
```

區域變數
i 的
scope

區域變數
n 的
scope

區域變數
total 的
scope

當 main 執行 fac 時, ans 並不會消失, 因為 life time 會持續到右大括號, 但在 ans 並沒有涵蓋在 fac 裡所以在 fac 裡時不能使用 ans

不同 scope 中同樣的變數名稱

```
int fac(int);  
int main(void){
```

```
    int ans = 0;
```

```
    ans = fac(5);
```

```
    return 0;
```

```
}
```

區域變數
ans 的
scope

```
int fac(int n){
```

```
    int ans = 1;
```

```
    printf("%d", ans);
```

```
    for(int i = 1 ; i <= n ; i++)
```

```
        ans *= i;
```

```
    return ans;
```

```
}
```

區域變數
i 的
scope

區域變數
n 的
scope

區域變數
ans 的
scope

前面說過 scope 為該變數**名稱**對應到某記憶體內容的可使用範圍，因此兩個會有各自的記憶體空間，若在紅色範圍內則對應到紅色 ans 的，在綠色範圍則對應到綠色 ans 的

不同 scope 中同樣的變數名稱 (cont')

```
int fac(int);
```

```
int main(void){
```

```
    int ans = 0;
```

```
    ans = fac(5);
```

```
    return 0;
```

```
}
```

區域變數 ans 的
scope

```
int fac(int n){
```

```
    int ans = 1;
```

```
    printf("%d", ans);
```

```
    for(int i = 1 ; i <= n ; i++)
```

```
        ans *= i;
```

```
    return ans;
```

```
}
```

區域變數 i 的
scope

區域變數 ans
的 scope

區域變數 n 的
scope


- 更改 fac 的 ans，並不會影響 main 的 ans
- 當結束 fac 時 fac 的 ans 就會被刪除 (空間釋放)
- main 的 ans 則要到 main 結束時才會被刪除

全域變數


- 若變數定義在所有的 block statement 外 (所有的 function 外), 則該變數為全域變數 (global variable)
- 全域變數的 scope 為從該宣告變數開始一直到程式檔結束位置
- 全域變數的 lifetime 為從程式一開始一直到程式結束
- 全域變數可以當作 function 與 function 間互相溝通的管道
- 全域變數也可以跨檔案來使用, 之後章節會在詳細講
- 好習慣: 多用區域變數, 少用全域變數, 除非必要

全域變數 - 例子

```
int fac(int);  
  
int ans;  
  
int main(void){  
    fac(5);  
    printf("%d", ans);  
    return 0;  
}
```



```
void fac(int n){  
    ans = 1;  
    for(int i = 1 ; i <= n ; i++){  
        ans *= i;  
    }  
    return;  
}
```



全域變數 **ans**
的 scope

- main 可以使用 **ans**, 同樣地 fac 也可以使用 **ans**, 兩者共享同一個變數 **ans**

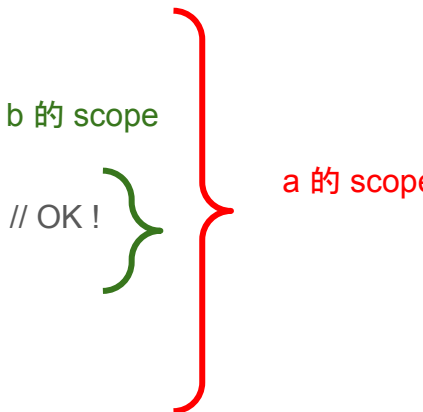
Scope 裡又有新的 Scope

原則:

外層的 scope 不能用內層 scope 的內容, 內層的 scope 可以用外層 scope 的內容

Scope 裡又有新的 Scope - 例子

```
int main(void){  
    if( /* some condition */){  
        int a = 1;  
        if( /* some condition */ ){  
            int b = 2;  
            printf("%d %d", a, b); // OK !  
        }  
        printf("%d", b); // ERROR !  
    }  
}
```



The diagram illustrates the scope of variables `a` and `b` in the provided C code. A green curly brace on the right side of the inner `if` block (lines 5-7) is labeled "b 的 scope" in green text, indicating that variable `b` is only defined within this inner scope. A red curly brace on the right side of the outer `if` block (lines 4-10) is labeled "a 的 scope" in red text, indicating that variable `a` is defined within the outer scope. This visualizes how a new scope can be created within an existing one, and how a variable declared in an inner scope is not accessible in the outer scope.

Scope 裡又有新的 Scope 且變數名稱重複

- 若內從的 scope 有變數名稱與外層 scope 重複, 則內層 scope 對該變數名稱時會使用內層的定義, 而非外層
- 在內層運算時, 外層的依然存在, 但只是無法指定到
- 當內層結束後, 使用該變數名稱時會使用到外層的

Scope 裡又有新的 Scope 且變數名稱重複 - 例子

```
int main(void){  
    if( /* some condition */){  
        int a = 1;  
        if( /* some condition */ ){  
            int a = 100;  
            a += 100;  
            printf("%d", a); // 結果為 200  
        }  
        printf("%d", a); // 結果為 1  
    }  
}
```

內層 a 的 scope

外層 a 的 scope

靜態變數 Static Variable

- 靜態變數與區域變數一樣在 block statement 進行宣告，且靜態變數的 scope 與區域變數一樣
- 但靜態變數的 life cycle 與 scope 範圍不同，靜態變數的 life cycle 與全域變數一樣，從程式執行開始一直到程式執行結束，也就是說變數 a 的儲存空間會從程式一開始一直保留到程式結束
- 需在變數名稱前加 static 關鍵字

靜態變數 Static Variable - 例子

```
int func(void);  
  
int main(void){  
    func(); // 印出 100  
    func(); // 印出 200  
    func(); // 印出 300  
    return 0;  
}
```

```
int func(void){  
    static int a = 100;  
    printf("%d\n", a);  
    a+=100;  
    return;  
}
```

- 當執行 static 變數初始化第二次時便不再執行
- 可以發現靜態變數即使在 func 結束時依然會保留，並等待之後可能的再次使用

const 修飾子

- 若一變數在其型別前加了 **const** 則此變數之後不能再被更改

```
const double pi = 3.14;
```

```
pi = 3.14159; // error !!!
```


引數傳遞的機制 - pass by value vs. pass by reference

- pass by value:

傳遞時函數會將引述**複製一份**新的在函數內部使用，外部的變數與複製過後用在函數內部的變數彼此互不影響彼此。

- pass by reference:

傳遞時函數時函數會使用 reference 來在函數內部稱此參數，在函數內部對變數操作會對外部的變數產生影響。

引數傳遞的機制 - 到底什麼是 reference

- reference 是變數的別稱, 但並不能代表真正記憶體空間的東西

A reference is an alias, an alternate name for an object. It is not an object itself (and in that way is *not* a pointer, even if some of their uses overlap with uses of pointers)

- C 沒有 reference 型別, 但 C++ 有, 所以留到 C++ 時在詳談

引數傳遞的機制 - 總而言之

- C 語言只有 pass by value, 但是我們可以指標 (pointer) 搭配 pass by value 達成 pass by reference 的效果

pass by value 特性

```
int swap(int, int);

int main(void){
    int a = 3, b = 5;

    printf("before swap: a = %d, b = %d", a, b);

    // a = 3, b = 5;

    swap(a, b);

    printf("after swap: a = %d, b = %d", a, b);

    // a = 3, b = 5;

    return 0;
}
```

```
int swap(int a, int b){
    int temp;

    temp = a;

    a = b;

    b = temp;

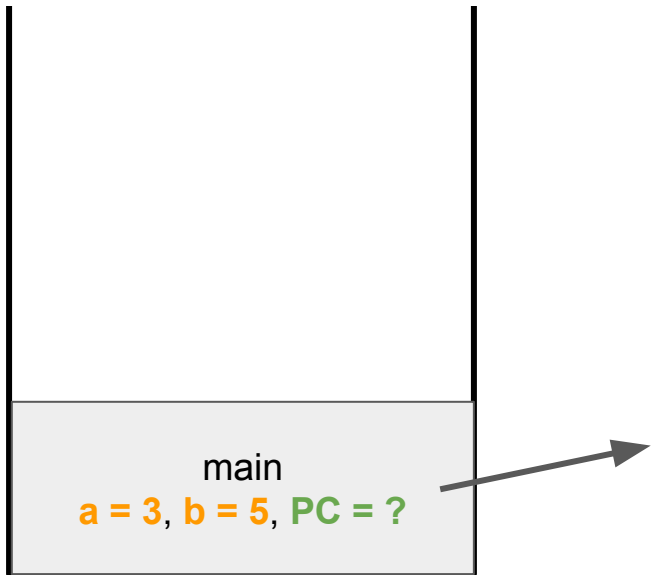
    printf("inside swap: a = %d, b = %d", a, b); // a = 5, b = 3
}
```

雖然我們傳 a, b 的值給 swap 函數, 但 swap 函數裡的 a, b 實際上的值是原本的 main 裡的 a, b **拷貝過去**的, 並非 main 裡的 a, b

因此無論我們在 swap 裡改變 a, b 的內容都不會影響到 main 裡的 a, b

program stack

執行 swap 函數之前



```
int swap(int, int);
```

```
int main(void){
```

```
    int a = 3, b = 5; // 1
```

```
    printf("before swap: a = %d, b = %d", a, b) // 2;
```

```
    swap(a, b); // 3
```

```
    printf("after swap: a = %d, b = %d", a, b); // 4
```

```
    return 0; // 5
```

```
}
```

```
int swap(int a, int b){
```

```
    int temp; // 15
```

```
    temp = a; // 16
```

```
    a = b; // 17
```

```
    b = temp; // 18
```

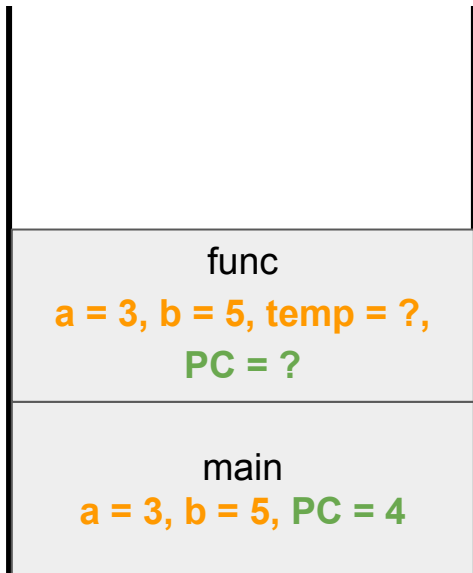
```
    printf("inside swap: a = %d, b = %d", a, b); // 19
```

```
}
```

stack frame 是一個記憶體區塊，裡面紀錄著該函數所用到的區域變數、要執行的下一行程式位置以及其他一些執行該函數會用到的重要資訊

program stack

執行 swap 函數之間,
a, b 互換之前



```
int swap(int, int);
```

```
int main(void){
```

```
    int a = 3, b = 5; // 1
```

```
    printf("before swap: a = %d, b = %d", a, b) // 2;
```

```
    swap(a, b); // 3
```

```
    printf("after swap: a = %d, b = %d", a, b); // 4
```

```
    return 0; // 5
```

```
}
```

```
int swap(int a, int b){
```

```
    int temp; // 15
```

```
    temp = a; // 16
```

```
    a = b; // 17
```

```
    b = temp; // 18
```

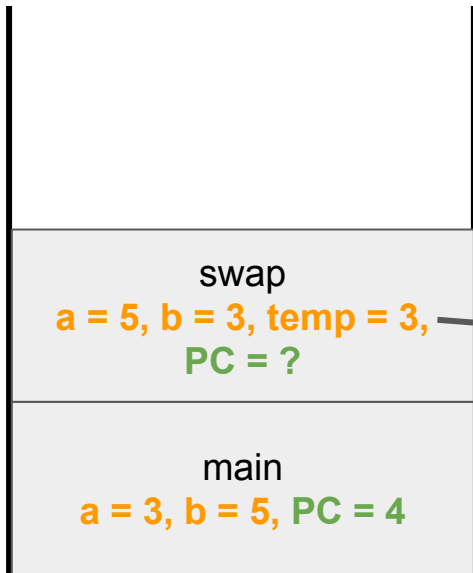
```
    printf("inside swap: a = %d, b = %d", a, b); // 19
```

```
}
```

呼叫函數 (進行 function call) 時, program stack 會
push (生出) 一個**新的 stack frame** 裡面有該函數的
local variable 以及 parameter 的值

program stack

執行 swap 函數之間,
a, b 互換之後



```
int swap(int, int);
```

```
int main(void){
```

```
    int a = 3, b = 5; // 1
```

```
    printf("before swap: a = %d, b = %d", a, b) // 2;
```

```
    swap(a, b); // 3
```

```
    printf("after swap: a = %d, b = %d", a, b); // 4
```

```
    return 0; // 5
```

```
}
```

```
int swap(int a, int b){
```

```
    int temp; // 15
```

```
    temp = a; // 16
```

```
    a = b; // 17
```

```
    b = temp; // 18
```

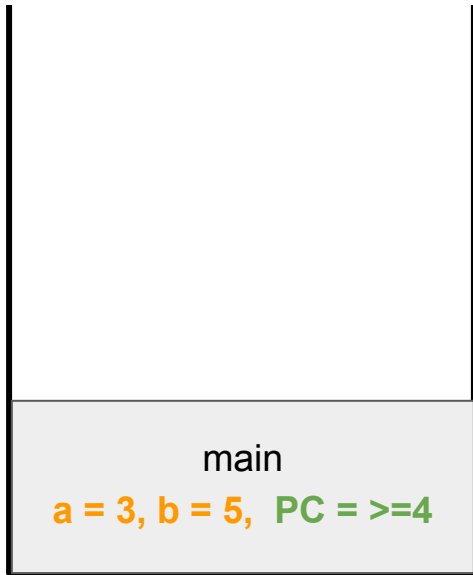
```
    printf("inside swap: a = %d, b = %d", a, b); // 19
```

```
}
```

由於 func 的 stack frame 與 main 的不同, 因此裡面各自的 a, b 互不影響, 因此在 func 裡修改 func 的區域變數 a, b 並不會影響到 main 的區域變數

program stack

完成 swap 函數之後



```
int swap(int, int);
```

```
int main(void){
```

```
    int a = 3, b = 5; // 1
```

```
    printf("before swap: a = %d, b = %d", a, b) // 2;
```

```
    swap(a, b); // 3
```

```
    printf("after swap: a = %d, b = %d", a, b); // 4
```

```
    return 0; // 5
```

```
}
```

```
int swap(int a, int b){
```

```
    int temp; // 15
```

```
    temp = a; // 16
```

```
    a = b; // 17
```

```
    b = temp; // 18
```

```
    printf("inside swap: a = %d, b = %d", a, b); // 19
```

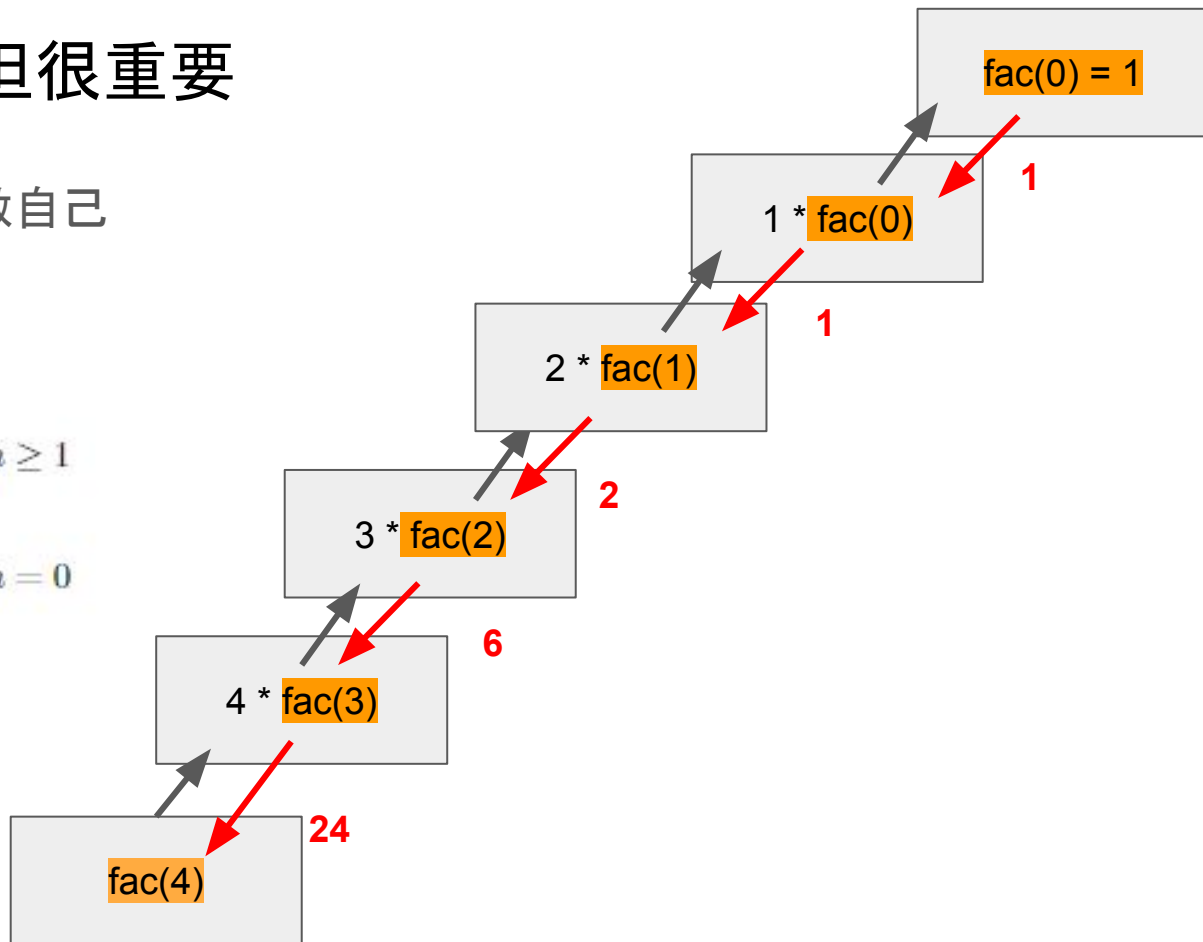
```
}
```

完成函數之後，最上層的 stack frame 並會被 pop (丟) 出去，程式會繼續執行最上層 stack frame 的內容

遞迴函數 - 不簡單但很重要

- 遞迴就是函數呼叫函數自己
- 舉例來說:

$$\boxed{fac(n)} = \begin{cases} n \times \boxed{fac(n-1)}, & \text{if } n \geq 1 \\ 1, & \text{if } n = 0 \end{cases}$$



遞迴函數 - 一定要有終止條件

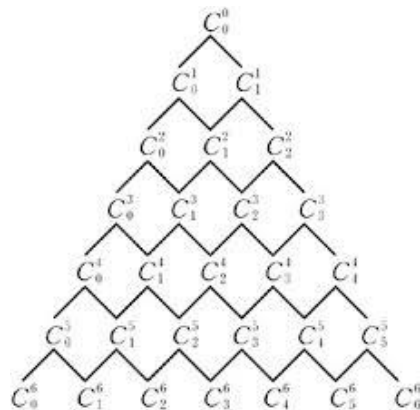
$$fac(n) = \begin{cases} n \times fac(n-1), & \text{if } n \geq 1 \\ 1, & \text{if } n = 0 \end{cases}$$

- 一定要有終止條件讓遞迴函數最終可以返回上一層的呼叫
- 否則每一層的函數都會佔據一段記憶體空間，最終導致記憶體不足



遞迴例子 - 上課實作

1. 巴斯卡定理 $C_n^m = C_n^{m-1} + C_{n-1}^{m-1}$



2. 最大公因數 (輾轉相除法)

$$\begin{array}{r|l|l|l} 3 & 34 & 10 & 2 \\ & 30 & 8 & \\ 2 & \hline & 4 & \textcolor{red}{2} & \\ & 4 & & \\ & \hline & 0 & & \end{array}$$

$$34 \div 10 = 3 \dots 4 \quad \rightarrow \quad 34 = 10 \times 3 + 4$$

$$10 \div 4 = 2 \dots 2 \quad \rightarrow \quad 10 = 4 \times 2 + 2$$

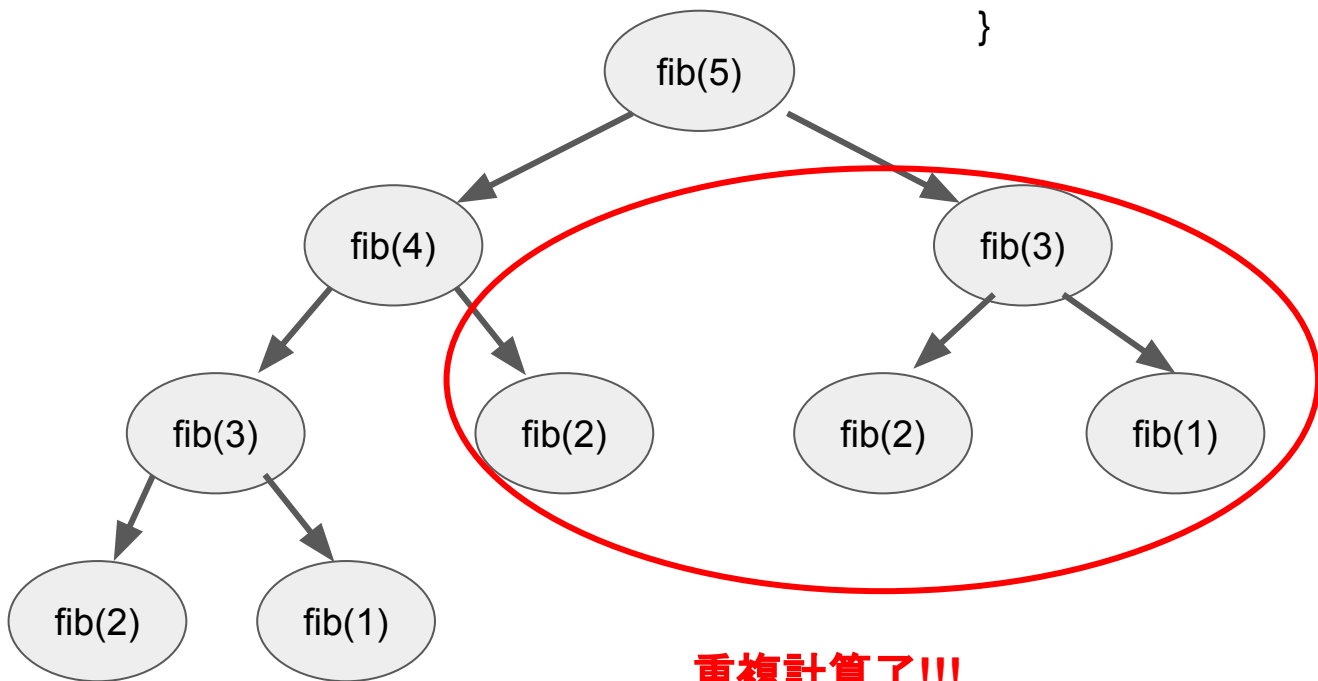
$$4 \div 2 = 2 \dots 0 \quad \rightarrow \quad 4 = 2 \times 2$$

3. 費氏數列

圖片來源: <http://www.mathland.idv.tw/fun/euclidean.htm>

遞迴效率 - 分析費氏數列

```
int fib(n){  
    if (n == 1 && n == 2) return 1;  
    return fib(n-1) + fib(n-2);  
}
```



- 可知並非以遞迴方式寫效率最好, 可能會出現重複計算的現象
- 可以利用動態規劃 (大魔王) 的方式避免重複計算, 之後談到演算法會在細談動態規劃

課堂練習題

- zerojudge - d487. Order's computation process
- zerojudge - e357. 遞迴函數練習
- zerojudge - a227. 三龍杯 -> 河內之塔

前置處理器 - #define

- 先前提過, C 程式在被編譯器(compiler)編譯前會先被前置處理器(preprocessor) 處理
- 以 # 開頭的指令都是呼叫前置處理器處理
- #include 為呼叫前置處理器先將函式庫的內容放進程式裡
- #define 可以將用一個識別名稱代替常用的字串、常數或函數(注意跟剛剛的函數有點小差別)

語法: #define 識別名稱 代換標記

- e.g. #define PI 3.14

前置處理器 - #define (cont')

- 若想要代換的內容太長, 可以利用 \ (反斜) 來分成兩行
- 已被定義的識別名稱不能重新在程式裡被賦值

PI = 3.1415;

- 因為會被前置處理器替換為 $3.14 = 3.1415$; 再交給編譯器做編譯, 自然會得到 error

使用巨集 macro

- 巨集只一個指令可以替代多個步驟
- 我們可以使用 `#define` 來替換一個程式區塊，也可以傳遞參數
- 在 macro 名稱後用一個小括號來代表要傳遞的參數

e.g. `#define SQUARE(X) X*X`

但請注意前置處理器只會複製貼上地置換因此

```
printf("%d", SQUARE(4+1));
```

會變成

```
printf("%d", 4+1*4+1);
```


使用巨集 macro (cont')

- 我們可以在代換標記有出現參數的地方都補上左右括號

e.g. `#define SQUARE(X) (X)*(X)`

如此一來

`printf("%d", SQUARE(4+1));`

就會變成

`printf("%d", (4+1)*(4+1));`