

程式設計 (一)

CH7. 函式 (下)

Ming-Hung Wang 王銘宏

tonymhwang@cs.ccu.edu.tw

Department of Computer Science and Information Engineering
National Chung Cheng University

Fall Semester, 2021

本章目錄

1. 變數的儲存類別
2. 函式呼叫堆疊
3. 遞迴 (recursive)
4. 使用遞迴的範例
5. 思考：河內塔

變數的儲存類別

變數的儲存類別

在之前，我們已經會使用識別字來作為變數名稱以及函式名稱，也討論過識別字當變數名稱時的範圍規則。但識別字還有一些其他特性，就是儲存類別 (storage class)。

識別字的儲存類別會影響儲存佔用期間 (storage duration)、能被參考的範圍 (scope) 等特性。

變數的儲存類別

C 提供了儲存類別指定詞 (storage class specifiers):
auto、register、extern 和 static。

- auto : 自動變數 (Automatic Variable)
- static : 靜態變數 (Static Variable)
- extern : 外部變數 (External Variable)
- register : 暫存器變數 (Register Variable)

auto：自動變數

或稱為內部變數，只可以是區域變數 (只能宣告在函式的參數列或函式本體)。

自動變數的有效範圍是由函式內變數的宣告處開始，一直到離開函式時，自動變數將釋放掉所佔用的記憶體空間，等到下次進入函式時，再重新配置記憶體位址給該變數使用，而無法保留其舊值。

變數的儲存類別

關鍵字 `auto` 用來明確指定宣告為自動變數。
函式的區域變數 (包括宣告在參數列或函式體的變數)
的儲存類別都預設為 `auto`，因此關鍵字 `auto` 幾乎不
會被使用。

```
auto int global; //ERROR! 全域範圍不能宣告auto  
  
void func() {  
    auto int local;  
}  
  
int main() {  
    auto int local;  
}
```

static：靜態變數

靜態變數可以是區域變數或全域變數。

屬於靜態儲存類別，與函式一樣，都是從程式開始執行至程式結束都存在（只會被宣告及初始化一次）。不過因為有範圍規則，這些變數並不是在程式的各個角落都可以被使用。

變數的儲存類別

區域靜態變數只會被宣告 1 次，
並在下次呼叫時延續之前的值

```
1 // stoage class
2 #include <stdio.h>
3
4 void counting() {
5     static int count = 0; //只宣告並初始化1次
6     printf("%d ", ++count);
7 }
8
9 int main() {
10     for (int i = 0; i < 5; i++)
11         counting();
12     printf("\n");
13 }
14 "D:\codeblocks\stoage class.exe"
15 1 2 3 4 5
```

(補充)extern：外部變數

屬於全域變數。全域變數與函式若沒有指明儲存類別，會被預設為 `extern`。

與 `static` 同屬於靜態儲存類別，與函式一樣，都是從程式開始執行至程式結束都存在。不過同樣因為有範圍規則，這些變數並不是在程式的各個角落都可以使用。

變數的儲存類別

因為範圍規則，第 5 行的變數 `global` 無法取得

```
1 // stoage class
2 #include <stdio.h>
3
4 void printGlobal() {
5     printf("%d\n", global);
6 }
7
8 int global = 10;
9
10 int main() {
11     printGlobal();
12 }
```

logs & others

Build messages X

File	Line	Message
D:\codeblocks...		=== Build file: "no target" in "no project" (compiler: unknown) ===
D:\codeblocks...		In function 'printGlobal':
D:\codeblocks...	5	error: 'global' undeclared (first use in this function)
D:\codeblocks...	5	note: each undeclared identifier is reported only once for each function it appears in
		=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

變數的儲存類別

extern 與全域變數的 static 不同的地方是，當在 extern 的全域變數的範圍規則之外的函式內，或是同專案的不同檔案，可藉由宣告同名的外部變數來取用。

變數的儲存類別

使用關鍵字 `extern` 可以宣告 (取用) 外部變數

```
1 // stoage class
2 #include <stdio.h>
3
4 void printGlobal() {
5     extern int global;
6     printf("%d\n", global);
7 }
8
9 int global = 10;
10
11 int main() {
12     printGlobal();
13 }
14 "D:\codeblocks\stoage class.exe"
15 10
```

變數的儲存類別

extern 變數一定要有在全域範圍的宣告，在區域宣告上也不能設定初始值

```
1 // _stoaage class
2 #include <stdio.h>
3
4 void printGlobal() {
5     extern int global;
6     printf("%d\n", global);
7 }
8
9 int main() {
10     extern int global;
11     global = 10;
12     printGlobal();
13 }
14
15
```

Logs & others

Build messages x

File	Line	Message
D:\codeblocks...		=== Build file: "no target" in "no i
D:\codeblocks...		undefined reference to 'global'
D:\codeblocks...		error: ld returned 1 exit status

```
1 // _stoaage class
2 #include <stdio.h>
3
4 void printGlobal() {
5     extern int global;
6     printf("%d\n", global);
7 }
8
9 int global;
10
11 int main() {
12     extern int global = 10;
13     printGlobal();
14 }
15
16
```

Logs & others

Build messages x

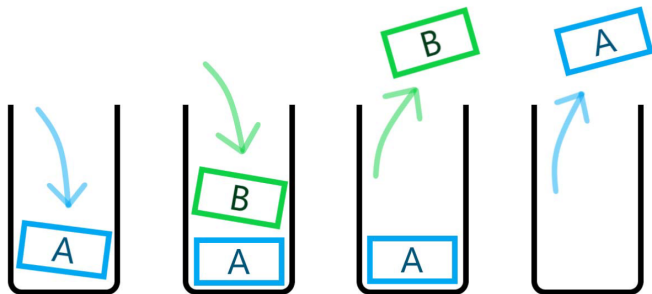
File	Line	Message
D:\codeblocks...		=== Build file: "no target" in "no project" (compile
D:\codeblocks...		In function 'main':
D:\codeblocks...	12	error: 'global' has both 'extern' and initializer
D:\codeblocks...		=== Build failed: 1 error(s), 0 warning(s) (0 minute

函式呼叫堆疊

函式呼叫堆疊

堆疊 (stack) 是一種後進先出 (last-in first-out, LIFO) 的結構。最後加入堆疊的項目會最先從堆疊取出。

而函式的呼叫就是一種堆疊的機制。



函式呼叫堆疊

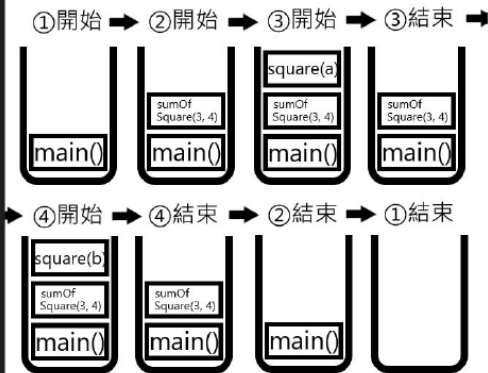
舉下面程式碼為例，執行流程會是：進入 main 函式
→ 進入 square 函式 → 結束 square 函式 → 結束 main
函式，這就式堆疊「後進先出」的概念。

```
1 // function call stack
2 #include <stdio.h>
3
4 int square(int n) {
5     return n * n;
6 }
7
8 int main() {
9     int a = square(5);
10    printf("%d\n", a);
11 }
```

函式呼叫堆疊

就算函式呼叫再複雜，
也符合堆疊「後進先出」的概念。

```
1 //function call stack
2 #include <stdio.h>
3
4 int square(int n) {
5     return n * n;
6 }
7
8 int sumOfSquare(int a, int b) {
9     return square(a) + square(b);
10 }
11
12 int main() {
13     int a = sumOfSquare(3, 4);
14     printf("%d\n", a);
15 }
```



遞迴 (recursive)

遞迴 (recursive)

遞迴函式 (recursive function) 就是一種可以直接或間接呼叫自己的函式。遞迴是進階電腦科學課程中所討論的一項複雜的課題。

用遞迴方法計算階乘

非負整數的階乘 n ，寫成 $n!$ ，如以下乘積：

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$$

而 $1!$ 等於 1 ， $0!$ 也定義為 1 。

遞迴 (recursive)

經過觀察，我們可以得到：

$$f(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } 1 \\ n \times (n-1)!, & \text{if } n > 1 \end{cases}$$

- 我們可以將任意非負整數代入上面式子驗證：

令 $n = 4$

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

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

$$4! = 4 * 3!$$

遞迴 (recursive)

我們可以把剛剛得到的數學式

$$f(x) = \begin{cases} 1, & \text{if } n = 0 \text{ or } 1 \\ n \times (n-1)!, & \text{if } n > 1 \end{cases}$$

寫成 C 語言

```
1 int factorial(int n) { //階乘
2     if(n <= 1)
3         return 1;
4     return n * factorial(n - 1);
5 }
```

遞迴 (recursive)

以呼叫 factorial(4) 為例
來看看這個函式如何運作的

```
1 | int factorial(int n){ //階乘
2 |     if(n <= 1)
3 |         return 1;
4 |     return n * factorial(n - 1);
5 | }
```

factorial(4)

return 4 * factorial(3)

設計遞迴觀念

遞迴函式的解決問題方法都有一個共同特點。遞迴函式只曉得如何解決最簡單的狀況，或稱**基本狀況** (base case)。

當遇到較複雜的狀況時，函式會將問題分成兩個概念性的小塊：「知道如何處理的部分」與「不知道如何處理的部分」，其中「不知道如何處理的部分」要近似於原來的問題，不過要比原來的問題還要簡單。

遞迴 (recursive)

比如剛才的階乘：

```
int factorial(int n){ //階乘
    if (n <= 1)        基本情況
        return 1;
    return n * factorial(n - 1);
}
```

知道如何出處理的部分 不知道如何處理的部分

遞迴 (recursive)

注意，如果無法讓遞迴函式一步一步簡化成基本情況，會造成無窮遞迴，最終造成記憶體耗盡。


練習

試著將用輾轉相除法求兩整數最大公因數的演算法，寫成遞迴函式。

- 輾轉相除法的運算方式是：
 1. 將兩整數代入到 $r = a \% b$ 的 a 與 b 中，求出 r
 2. 將上一步的 b 與 r 的值重新帶入 $r = a \% b$ 的 a 與 b 中，求出新的 r
 3. 重複步驟 2.，直到 r 等於 0
 4. 最後的 b 即是兩整數的最大公因數

遞迴 (recursive)

參考解答



```
int GCD(int a, int b) {  
    if (b == 0) {  
        return a;  
    }  
    return GCD(b, a % b);  
}
```

遞迴 (recursive)

迭代 (迴圈) 與遞迴比較：

```
int GCD(int a, int b) {  
    int r;  
    while (b != 0) {  
        r = a % b;  
        a = b;  
        b = r;  
    }  
    return a;  
}
```

```
int GCD(int a, int b) {  
    if (b == 0) {  
        return a;  
    }  
    return GCD(b, a % b);  
}
```

使用遞迴的範例

費氏數列

費氏數列 (fibonacci sequence)，是以 0,1 開始，之後的每一個 Fibonacci 數，都是它的前兩個 Fibonacci 數之和。

0, 1, 1, 2, 3, 5, 8, 13, 21,

費氏數列的數學式如下：

$$fib(n) = \begin{cases} n, & \text{if } n = 0 \text{ or } 1 \\ fib(n-1) + fib(n-2), & \text{if } n > 1 \end{cases}$$

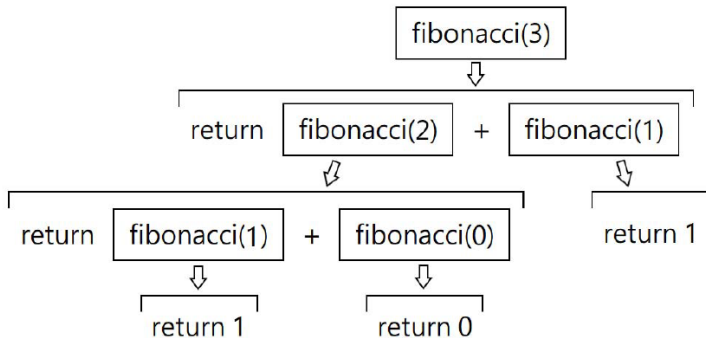
使用遞迴的範例

求出費氏數列第 n 項的函式：

```
int fibonacci(int n) { // 費氏數列
    if(n <= 1){ // 基本狀況
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

使用遞迴的範例

呼叫 fibonacci(3) 的遞迴呼叫



指數複雜度

關於遞迴程式 (例如我們用來產生 Fibonacci 數的程式) 有一點值得注意，就是它的複雜度等級的問題。

在 fibonacci 函式中，每一級的遞迴都會讓呼叫的總數變為 2 倍，亦即計算第 n 個 Fibonacci 數時，需要執行 2^n 次的遞迴呼叫。

使用遞迴的範例

當我們要計算 20 個 Fibonacci 數，便需要約 2^{20} 個函式呼叫，也就是一百萬個函式呼叫。


要計算第 30 個 Fibonacci 數，便需要約 2^{30} 個函式呼叫，也就是十億個函式呼叫

電腦科學家將這種現象稱為指數等級複雜度
(exponential complexity)

函式呼叫堆疊

利用函式呼叫堆疊概念，倒著印出輸入的字母

```
1 // recursive
2 #include <stdio.h>
3
4 void printReversely() {
5     char word;
6     if (scanf("%c", &word) != EOF && word != '\n') {
7         printReversely();
8         printf("%c", word);
9     }
10 }
11
12 int main() {
13     printReversely();
14     printf("\n");
15 }
```



使用遞迴的範例

main 函式遞迴

如同一般的函式，main 函式也可以自己呼叫自己

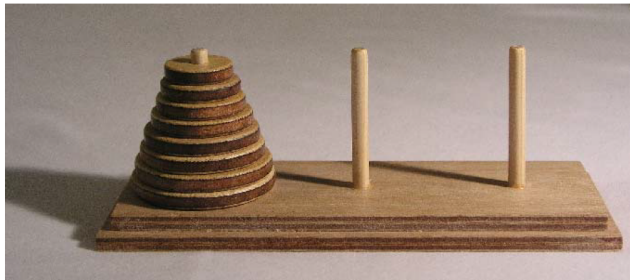
```
1 // main function recursive
2 #include <stdio.h>
3
4 int main() {
5     static int sum = 0; //靜態變數，只會被宣告一次
6     int input;
7     printf("輸入整數(輸入0結束輸入) : ");
8     scanf("%d", &input);
9     if (input) {
10         sum += input;
11         main(); //呼叫自己
12     }
13     else {
14         printf("總和為%d\n", sum);
15     }
16 }
```



```
"D:\codeblocks\main function recursive.exe"
輸入整數(輸入0結束輸入) : 12
輸入整數(輸入0結束輸入) : 50
輸入整數(輸入0結束輸入) : 33
輸入整數(輸入0結束輸入) : 40
輸入整數(輸入0結束輸入) : 0
總和為135
```

思考：河內塔

思考：河內塔




思考：河內塔

傳說中在東方的寺廟裡，僧侶試圖將一疊的碟子由一根柱子移到另一根。剛開始時有 64 個碟子，由大到小的疊在一根柱子上。僧侶們移動碟子的規定是，一次只能移動一個碟子，而且在任何時刻都不能發生大碟子壓在小碟子之上的情形。第三根柱子可以用來暫時置放碟子。因為當僧侶完成他們的工作的時候世界末日就會來到，這讓我們沒有多少動力來幫助他們工作。

思考：河內塔

請試著設計一支程式，首先讓使用者輸入一個整數 n ，代表河內塔有 n 個碟子，接下來印出河內塔由 1 號柱子全部移到 3 號柱子的過程 (過程請註明 X 號碟子由 Y 號柱移到 Z 號柱，1 號碟子是最小的碟子)。

 D:\codeblocks\Practice.exe

```
請輸入疊子數：3  
1號疊子由1號柱移到3號柱  
2號疊子由1號柱移到2號柱  
1號疊子由3號柱移到2號柱  
3號疊子由1號柱移到3號柱  
1號疊子由2號柱移到1號柱  
2號疊子由2號柱移到3號柱  
1號疊子由1號柱移到3號柱
```

參考資料： Deitel, H. M., & Deitel, P. J. (2015). C: How to program.
Upper Saddle River, N.J: Prentice Hall.