

# 高等C語言

Shengwen Cheng

Published  
with GitBook



# 目錄

Introduction	0
簡介	1
目錄	2
語言基礎	3
第一個程式 -- hello.js	3.1
hello.c	3.2
加總 -- sum.js	3.3
sum.c	3.4
函數 -- fsum.js	3.5
fsum.c	3.6
函數 -- max.js	3.7
max.c	3.8
陣列 -- array.js	3.9
array.c	3.10
字串 -- string.js	3.11
string.c	3.12
回呼 -- 微分 df.js	3.13
df.c	3.14
回呼 -- 積分 integral.js	3.15
integral.c	3.16
遞迴 -- recursive.js	3.17
recursive.c	3.18
字典 -- dict.js	3.19
dict.c	3.20
英翻中系統 -- mt.js	3.21
mt.c	3.22
英中互翻系統 -- mt2.js	3.23

mt2.c	3.24
中翻英系統 -- c2e.js	3.25
c2e.c	3.26
自動產生英文語句	3.27
genen.c	3.28
JS的模組化	3.29
C 的模組化	3.30
JS的物件導向	3.31
C的結構	3.32
字串處理	4
字串大小的問題	4.1
字串的格式化	4.2
sprintf 函數	4.3
sscanf 函數	4.4
標準字串函式庫	4.5
字串的誤用	4.6
動態字串物件	4.7
寬字串函數	4.8
寬窄字串間的轉換	4.9
字串的誤用	4.10
指標	5
指標算術	5.1
動態陣列物件	5.2
陣列大小的問題	5.3
函數指標	5.4
函數指標型態	5.5
變動參數	5.6
結構	6
結構的初始化	6.1
結構中的位元欄	6.2

結構的指標算術	6.3
鏈結串列	6.4
鏈結串列-基礎版	6.4.1
鏈結串列：內含物件版	6.4.2
鏈結串列：外包物件版	6.4.3
物件導向	7
物件導向的基本概念 -- 封裝，繼承，多型	7.1
封裝 — (Encapsulation)	7.2
繼承 — (Inheritance)	7.3
多型 — (Polymorphism)	7.4
類別結構 — (Define Class)	7.5
記憶體管理	8
C 語言的執行環境	8.1
記憶體漏洞	8.2
檔案系統	9
檔案緩衝區	9.1
臨時暫存檔	9.2
檔案錯誤	9.3
目錄管理	9.4
錯誤處理	10
錯誤代號與訊息	10.1
錯誤訊息列表	10.2
檔案錯誤	10.3
短程跳躍	10.4
長程跳躍	10.5
訊號機制	10.6
模擬 try ... catch	10.7
巨集處理	11
將函數巨集化	11.1
引用防護	11.2

條件編譯	11.3
編譯時期變數	11.4
編譯時期函數	11.5
編譯指示	11.6
字串化	11.7

## 序

本書起源於 2010 年我開始在 wikidot 上撰寫的一本《高等C語言》網路書，其實是自己《一邊學、一邊寫、一邊出版》的習慣所產生的作品，目的是重新複習一些 C 語言中自己較不熟悉的語法與技巧，並且趁這個機會深入學習 C 語言。

2011年 jserv 看到這本書之後，發現了一些錯誤，於是寄信給我，讓我修正了這些錯誤！

想不到在 2016 年 3 月 2 日，jserv 再度寄來十幾封勘誤信，最後一封告訴我希望採用這本書作為上課的參考教材，並建議我能放上 gitbook 。

在我修改了這些錯誤之後，由於我分身乏術，於是在 [facebook](#) 上發文請求協助，於是我的一位好友《鄭聖文》承擔了這個任務，然後這本在 gitbook 上的書籍就誕生了。

當初認識聖文的時候，他還是一位高中二年級的學生，在開源人年會 COSCUP 上，他自我介紹並拿了名片給我，手上還拿著一片 jserv 上課時用的 ARM 開發板，這真的讓我驚呆了！

一位高中二年級的學生，竟然就跟著 jserv 在寫嵌入式的程式，真是太強了！後來有一次我還跑去成大聽 jserv 上課，他也在場，變成我的同學。

在聖文的身上，我看到台灣資工領域的新希望，他常常每週在《台南、台中、新竹、台北》到處跑來跑去，到處修課，感覺好像火車票不用錢一樣。

我總是在想，假如台灣的資工系學生都像他一樣，那我們還需要擔心台灣的資工領域沒有未來，台灣的資訊產業沒有前途嗎？假如台灣有十萬個像鄭聖文這樣好學又肯動手做的學生，那我們的資訊軟體產業二十年後肯定是世界第一，哪需要我們這些 LKK 去替他們設想未來到底應該何去何從呢？

年紀比我小十歲的 jserv，能力比我強上十倍，但是請小心了，年紀比 jserv 小十歲的鄭聖文來了，想讓台灣電資產業發光發熱的企業經營者們，千萬別放過他！

陳鍾誠 2016/3/7 於金門大學

## 作者

初版作者 陳鍾誠，於金門大學資訊工程系，電子郵件：[ccc@nqu.edu.tw](mailto:ccc@nqu.edu.tw)，網站：<http://ccc.nqu.edu.tw>

共同作者兼技術指導 jserv 電子郵件：[jserv.tw@gmail.com](mailto:jserv.tw@gmail.com) 網站：<http://jserv.logdown.com/>

共同作者兼編輯 鄭聖文 電子郵件：[shengwen1997.tw@gmail.com](mailto:shengwen1997.tw@gmail.com) 網站：<https://github.com/shengwen1997>

共同作者兼編輯 郭璟瑋 電子郵件：[twfinderaz@gmail.com](mailto:twfinderaz@gmail.com) 網站：<https://github.com/findergithub>

## 授權

本文採用創作共用 (Creative Common) 3.0 版的 姓名標示—相同方式分享 授權條款，歡迎轉載或修改使用。

## 聲明

本書部份內容與大部份圖片修改自 維基百科，使用時請遵守 姓名標示、相同方式分享 授權。

# 簡介

## 前言

當我還是一個大學生的時候，總覺得 C 語言就是這樣了。但是在 10 年後我進入職場時，才發現原來我並不太認識這個語言。產業界所使用的 C 語言有許多是大學所沒有教授過的，像是 `#ifdef`、`make`、GNU 工具等等。又過了 10 年，當我研究嵌入式系統時，這個感覺又出現了，我仍然不太認識 C 語言，嵌入式系統中所使用的「記憶體映射輸出入、`volatile`、組合語言連接、`Linker Script`」等，又讓我耳目一新，我再度重新認識了 C 語言一次。然後，當我研讀 Linux 核心的程式碼時，看到 Torvalds 所使用的「鏈結串列、行程切換技巧」等，又再度讓我大為驚訝，C 語言竟然還可以這樣用。然後，當我開始研究 Google Android 手機平台的架構時，又看到了如何用 C 語言架構出網路、視窗、遊戲、瀏覽器等架構，於是我必須再度學習一次 C 語言。

當我翻閱坊間的書籍時，不禁如此想著，如果有人能直接告訴我這些 C 語言的學習歷程，那應該有多好。難道，我們真的必需花上數十年的時間去學習 C 語言，才能得到這些知識嗎？這些知識在初學者的眼中，看來簡直像是「奇技淫巧」。然而這些「奇技淫巧」，正是 C 語言為何如此強大的原因，我希望能透過這本書，告訴各位這些「奇技淫巧」，讓各位讀者不需要再像我一樣，花上二十年功夫，才能學會這些技術。

在我的眼中，C 語言就像一把鋒利的雙面刃，初出茅廬的人往往功力不夠深厚，反而將這個神兵利器往自己身上砍，因而身受重傷。但是在專家的手中，C 語言卻具有無比的威力，這種神兵利器具有「十年磨一劍、十步殺一人」的驚人力量。筆者希望能透過這本書，讓讀者能夠充分發揮 C 語言的力量，快速的掌握這個難以駕馭的神兵利器。

## C 語言的由來

從 1972 年 Dennis Ritchie 在貝爾實驗室發明 C 語言至今，已經超過四十年。在這個變化快速的電腦世界裡，C 語言彷彿成了不變的避風港。四十年來，C 語言的改變並不多，而且一直都是所有作業系統底層的主力語言。近來，由於 Linux 與開放原始碼的發展，C 語言的影響力更為增強。在這裡，我不禁要問一個問題，為何 C 語言可以經過四十年而幾乎毫不改變。



C 語言很快，這或許是原因之一，但是像 Pascal 或 Fortran 等語言也幾乎與 C 語言一樣快，那又為何非 C 語言不可。但是，C 語言不只是快，還具有指標，容易與組合語言連結，具有巨集、條件式編譯、inline 函數、結構化、可以使用記憶體映射輸出入，因此可以用高階語言撰寫低階輸出入驅動程式，還有撰寫作業系統。

這些特性，讓 C 語言特別適合撰寫嵌入式系統，在某些 deeply embedded 的資源受限型嵌入式系統中，只有很小的記憶體、很慢的 CPU、而且通常沒有硬碟。這種環境有點像相當於當年 Dennis Ritchie 所面對的環境，因為他們必須在很克難的資源中，發展出夠強大的作業系統。也正是因為這種環境的淬鍊、讓 C 語言在《嵌入式系統》和《作業系統》領域都有非常優異的表現。

UNIX 正是催生 C 語言的主要動力，當年 Ken Thompson 與 Dennis Ritchie 正是為了發展 UNIX 而設計出 C 語言的，這兩人也因為 UNIX/C 的貢獻而被 ACM 授予 Turing Award 這的電腦界的諾貝爾獎。

在 1978 年，Dennis 與另一位共同作者 Brian Wilson Kernighan 合力撰寫了第一本廣為流傳的 C 語言教科書，而這個版本的教科書由於影響深遠，成為人手一冊的 C 語言經典，因此後來我們這個版本的 C 語言教科書簡稱為 K&R 版本，這個經典書籍中所使用的 C 語言版本也因此而被稱為 K&R 版的 C 語言，以便與後來 1988 年的 ANSI C 版本，以及 1999 年的 ISO C99 版本有所區隔。(一個很容易誤會的點是，Ken Thompson 與 Brian Wilson Kernighan 是不同的兩個人，Ken Thompson 是發明 UNIX 與 C 語言的那個 Turing Award 得獎者，但是 Brian Wilson Kernighan 則是 C 語言書籍的作者，這兩個人的名字雖然都以 K 開頭，但是此 K 非比 K，請讀者切勿混淆)。

因此，學習 C 語言的人，如果只是將 C 當作是一般的程式語言，就會難以體會 C 語言的威力之所在，我們必須進入嵌入式與作業系統的領域，才能體會 C 語言的優點。一旦您能夠體會這些優點，C 語言將不再僅僅是一個普通的語言，您也將能體會為何 C 語言會經歷四十年而不墜。然後，您也才能發揮 C 語言的能力，並且體會這些設計背後的優點與缺點。

## C 語言的優缺點

C 語言並非沒有缺點的，實際上，C 語言的缺點非常的多，多到可以用罄竹難書來形容。舉例而言，用 C 語言寫程式很容易有 bug，特別是在記憶體分配與回收這部份更是如此。C 語言沒有自動記憶體回收機制，沒有垃圾收集功能，因此常常導致忘記釋放記憶體，或者將同一個記憶體釋放數次，因而造成錯誤。C 語言的字串很原始，使用起來非常不方便。C 語言的標準函式庫甚至沒有基本的資料結構，像是

陣列、串列、堆疊、字典等相關結構的函式庫。C 語言的條件式編譯讓程式看起來很冗長，使用標頭檔 \*.h 讓你必需重複撰寫函數表頭，浪費許多時間。更糟的是，由於 C 語言的標準函式庫很小，因此在不同的平台上，每個廠商都實作出完全不同的函式庫，這導致 C 語言的程式難以跨越平台執行，您必須位每個平台打造一份程式，而不像 Java 那樣可以 Write Once, Run Anywhere。

但是，即便有了這麼多的缺點，C 語言仍然歷經四十年而不衰，這又是為甚麼呢？

每個 C 語言的缺點，幾乎都是伴隨著其優點而來的，C 語言的記憶體難以管理，是因為 C 語言具有強大的指標功能。字串函數很原始，是為了讓您可以使用字元陣列的方式處理字串，而不需要使用動態記憶體配置。無法跨越平台，是因為 C 語言適合用來打造底層的嵌入式系統，可以直接連結組合語言協同工作。從這個角度看來，C 語言的設計其實是相當精巧的，這也是 C 語言為何經歷四十年而不衰的原因。

## 學習 C 語言的好處

C 語言幾乎是當今被廣泛使用的語言當中，唯一同時具有高階與低階特性的語言，這個特性主要是由指標所造成的。利用指標，您可以用記憶體映射的方法存取記憶體，這讓 C 語言可以直接與周邊裝置溝通，因此許多裝置驅動程式可以用 C 語言撰寫，而不需要全部用組合語言。

學習 C 語言的投資報酬率，必須以數十年甚至一輩子的眼光來看，而不是短視的。許多其他的語言，多如過江之鯽，每個兩三年就必須學習全新的語言，就像流行音樂或服飾一般，學會之後很快就會膩了。C 語言絕對不是流行的語言，而是一種經典的、長久的、耐用的語言，您在 C 語言的投資不會浪費，因為 C 語言將會陪伴您，走過數十年，甚至是一輩子。

## 語言的歷史

- 1972 -- 貝爾實驗室的 Dennis Richie 在 B 語言的基礎上發展出 C 語言。
- 1973 -- C 語言加入 struct 之後，UNIX 的上層整個以 C 語言改寫，這個 UNIX+C 的組合極為成功，開啟了《作業系統和程式語言》的新頁。
  - 原本我以為這版的 UNIX 是第一個用高階語言寫的作業系統，但是根據 Jserv 指出，從 UNIX v1 開始，工具程式就不全然用組合語言開發，B 語言解析器已經在 v2 使用。而用高階語言撰寫的作業系統更是不少，像是 Multics 就用 PL/I 這個遠比 C 語言高階、出現年代更早的程式語言開發。

- 1975 -- UNIX 第六版公布，C 語言開始引起人們的注意。
- 1978 -- UNIX 第七版相當成功，該版本的 C 成為標準的 C 語言。
- 1983 -- ANSI 協會制定了 ANSI C 的標準。
- 1990 -- ISO 組織制訂了 ISO C 的標準。

# 目錄

## C 語言 -- 切開作業系統的瑞士刀

主題	說明
簡介	Introduction
語言基礎	Language
字串處理	String
指標	Pointer
結構	Structure
物件導向	Object
記憶體管理	Memory
檔案系統	File
錯誤處理	Error Handling
巨集處理	Macro & Conditional Expansion
C 與組合語言	C & Assembly

## 基本範例

教學	JavaScript	C 語言	JS 教學錄影	C 教學錄影
習題	第一個程式 -- hello.js	hello.c		
習題	加總 -- sum.js	sum.c		
習題	函數 -- fsum.js	fsum.c		
習題	函數 -- max.js	max.c		
習題	陣列 -- array.js	array.c		
習題	字串 -- string.js	string.c		
習題	回呼 -- 微分 df.js	df.c		
習題	回呼 -- 積分 integral.js	integral.c		
習題	遞迴 -- recursive.js	recursive.c		
習題	字典 -- dict.js	dict.c		
習題	英翻中系統 -- mt.js	mt.c	影片	
習題	英中互翻系統 -- mt2.js	mt2.c	影片	
習題	中翻英系統 -- c2e.js	c2e.c	影片	
習題	自動產生英文語句	genen.c	影片	
文章	JS的模組化	C 的模組化		
文章	JS的物件導向	C的結構		
文章		C的指標		

# 第一個程式

檔案：**hello.js**

```
console.log('hello 你好!');
```

執行結果

```
$ node hello.js  
hello 你好!!
```

# hello.c

檔案：**hello.c**

```
#include <stdio.h>

int main() {
    printf("hello 你好!\n");
}
```

執行結果

```
D:\code>gcc hello.c -o hello
D:\code>hello
hello 你好！
```

注意：在 windows 8 當中我必須將 hello.c 儲存成 ansi 編碼，中文字才能正確印出。

## 範例 -- 加總

### 使用 **while** 迴圈

檔案：**wsum.js**

```
sum=0;
i=1;
while (i<=10) {
    sum = sum + i;
    console.log("i=", i, " sum=", sum);
    i = i + 1;
}
```

執行結果：

```
D:\jsbook>node wsum.js
i= 1  sum= 1
i= 2  sum= 3
i= 3  sum= 6
i= 4  sum= 10
i= 5  sum= 15
i= 6  sum= 21
i= 7  sum= 28
i= 8  sum= 36
i= 9  sum= 45
i= 10 sum= 55
```

### 使用 **for** 迴圈

檔案：**sum.js**



```
sum=0;
for (i=1;i<=10;i++) {
    sum = sum + i;
    console.log("i=", i, " sum=", sum);
}
```

執行結果：

```
D:\jsbook>node sum.js
```

```
i= 1  sum= 1
i= 2  sum= 3
i= 3  sum= 6
i= 4  sum= 10
i= 5  sum= 15
i= 6  sum= 21
i= 7  sum= 28
i= 8  sum= 36
i= 9  sum= 45
i= 10 sum= 55
```

## 範例 -- 加總 (sum.c)

### 使用 **while** 迴圈

檔案：**wsum.c**

```
#include <stdio.h>
int main() {
    int i=1, sum=0;
    while (i<=10) {
        sum = sum + i;
        printf("i=%d sum=%d\n", i, sum);
        i = i + 1;
    }
}
```

執行結果：

```
D:\Dropbox\cccw\db\c\code>gcc wsum.c -o wsum

D:\Dropbox\cccw\db\c\code>wsum
i=1 sum=1
i=2 sum=3
i=3 sum=6
i=4 sum=10
i=5 sum=15
i=6 sum=21
i=7 sum=28
i=8 sum=36
i=9 sum=45
i=10 sum=55
```

### 使用 **for** 迴圈

檔案：**sum.c**

```
#include <stdio.h>

int main() {
    int i, sum=0;
    for (i=1;i<=10;i++) {
        sum = sum + i;
        printf("i=%d sum=%d\n", i, sum);
    }
}
```

執行結果：

```
D:\Dropbox\cccwd\db\c\code>gcc sum.c -o sum

D:\Dropbox\cccwd\db\c\code>sum
i=1 sum=1
i=2 sum=3
i=3 sum=6
i=4 sum=10
i=5 sum=15
i=6 sum=21
i=7 sum=28
i=8 sum=36
i=9 sum=45
i=10 sum=55
```

## 計算總和的函數

檔案：**fsum.js**

```
function sum(n) {  
    s=0;  
    for (i=1; i<=n; i++) {  
        s = s+i;  
    }  
    return s;  
}  
  
sum10 = sum(10);  
console.log("1+...+10="+sum10);
```

執行結果：

```
nqu-192-168-61-142:code mac020$ node fsum.js  
1+...+10=55
```

## 計算總和的函數

檔案：**fsum.c**

```
#include <stdio.h>

int sum(n) {
    int i, s=0;
    for (i=1; i<=n; i++) {
        s = s+i;
    }
    return s;
}

int main() {
    int sum10 = sum(10);
    printf("1+...+10=%d\n", sum10);
}
```

執行結果：

```
D:\Dropbox\cccwd\db\c\code>gcc fsum.c -o fsum
D:\Dropbox\cccwd\db\c\code>fsum
1+...+10=55
```

## 函數：取大值 `max(a,b)`

檔案：`max.js`

```
function max(a,b) {  
    if (a>b)  
        return a;  
    else  
        return b;  
}  
  
m = max(9,5);  
console.log("max(9,5)="+m);
```

執行結果：

```
$ node max.js  
max(9,5)=9
```

在此必須說明一點，函數呼叫時，參數的傳遞與名稱無關，而是與參數的位置有關。

舉例而言，上面我們用 `max(9,5)` 呼叫 `max` 函數，此時 `a`, `b` 分別會帶入 `a=9`, `b=5` 的值進去，因此判斷式會得到 `if (9>5) return 9`; 於是會傳回 9。

但是如果我們用下列的呼叫方式，那麼就會得到 `z=7`。

```
x=3; y=7;  
z = max(x,y); // 此時 x 會代入 a, y 會代入 b 於是得到 max(x,y) = max(3,
```

假如呼叫時有 `a`, `b` 等變數，也不會因為名稱相同而代入，而是按照位置填入，舉例而言，在以下程式中，

```
a=8; b=2;  
c = max(b,a); // 此時 b=2 會代入參數 a, 而 a=8 會代入參數 b 於是得到 max
```

總而言之，函數是根據呼叫時的位置代入的，而非根據名字。

## 函數：取大值 `max(a,b)`

### 採用函數的作法

檔案：`max.c`

```
#include <stdio.h>

int max(int a, int b) {
    if (a>b)
        return a;
    else
        return b;
}

int main() {
    int m = max(9,5);
    printf("max(9,5)=%d\n", m);
}
```

執行結果：

```
D:\Dropbox\cccw\db\c\code>gcc max.c -o max

D:\Dropbox\cccw\db\c\code>max
max(9,5)=9
```

### 採用巨集的作法

C 語言屬於靜態語言，所有變數都要宣告型態，這樣編譯器才能正確進行編譯。

但是如果你想寫一個《通用的函數》，不想綁訂在某些型態上，在 C 語言裡應該怎麼辦呢？

一個可能的方法是使用巨集！



巨集和函數不同，會被事先展開，然後在進行編譯，以下是一個用 C 語言設計通用的 `max(a,b)` 巨集的範例！

檔案：**maxMacro.c**

```
#include <stdio.h>

#define max(a,b) ((a>b)?a:b)

int main() {
    int m = max(9,5);
    printf("max(9,5)=%d\n", m);

    printf("max(3,8)=%d\n", max(3,8));
}
```

執行結果：

```
D:\Dropbox\cccw\db\c\code>gcc maxMacro.c -o maxMacro

D:\Dropbox\cccw\db\c\code>maxMacro
max(9,5)=9
max(3,8)=8
```

這種巨集會被先展開成沒有巨集的程式，方法是在每次呼叫時都《將整段程式碼貼上並取代參數》。

我們可以透過 gcc 的 `-E` 參數，來觀察展開的情況，指令如下：

```
D:\Dropbox\cccw\db\c\code>gcc -E maxMacro.c -o maxMacro.i
```

以下是展開後的程式碼

```
// ... 前面有一大堆註解
int main() {
    int m = ((9>5)?9:5);
    printf("max(9,5)=%d\n", m);

    printf("max(3,8)=%d\n", ((3>8)?3:8));
}
```

## 範例：陣列 -- array.js

JavaScript 的陣列宣告非常簡單，就是用 [...] 所框起來的一連串資料，或者您也可以使用 `new Array()` 語句來建立一個空的陣列，而且可以用 `length` 屬性來取得陣列大小。

檔案：**array.js**

```
var a=[1,6,2,5,3,6,1];

for (i=0;i<a.length;i++) {
    console.log("a[%d]=%d", i, a[i]);
}
```

執行結果

```
D:\jsbook>node array.js
a[0]=1
a[1]=6
a[2]=2
a[3]=5
a[4]=3
a[5]=6
a[6]=1
```

## 範例：陣列 -- array.c

C 語言的宣告陣列時，通常必須給定陣列大小，除非是將整個陣列初始值宣告進去時，才能交由編譯器自動決定大小。

檔案：**array.c**

```
#include <stdio.h>

int main() {
    int i;
    int a[]={1,6,2,5,3,6,1};
    for (i=0;i<7;i++) {
        printf("a[%d]=%d\n", i, a[i]);
    }
}
```

執行結果

```
D:\Dropbox\cccwd\db\c\code>gcc array.c -o array
```

```
D:\Dropbox\cccwd\db\c\code>array
```

```
a[0]=1
```

```
a[1]=6
```

```
a[2]=2
```

```
a[3]=5
```

```
a[4]=3
```

```
a[5]=6
```

```
a[6]=1
```

## 字串運算

```
> x = "hello"
'hello'
> x.length
5
> x[3]
'l'
> x[2]
'l'
> x[1]
'e'
> x[0]
'h'
> x[5]
undefined
> x[4]
'o'
> x=x+" world"
'hello world'
> x
'hello world'
>
```

## 範例：字串 -- string.c

C 語言宣告的字串，其實只是一種以 ASCII 編碼 0 的字母作為結果的字元陣列而已。(不像 javascript 的字串是一個物件，C 語言沒有物件的概念)。

檔案：**string.c**

```
#include <stdio.h>
#include <string.h>

int main() {
    int i;
    char s[]="hello!";
    for (i=0;i<strlen(s)+1;i++) {
        printf("s[%d]=%c ascii=%d\n", i, s[i], s[i]);
    }
}
```

執行結果

```
D:\Dropbox\cccwd\db\c\code>gcc string.c -o string

D:\Dropbox\cccwd\db\c\code>string
s[0]=h ascii=104
s[1]=e ascii=101
s[2]=l ascii=108
s[3]=l ascii=108
s[4]=o ascii=111
s[5]=! ascii=33
s[6]=
```

說明：C 語言編譯器會自動在結尾多補一個 ASCII 代號 0 的字元，所以 s[6] 才會印出空的內容。

# 微分

檔案：**df.js**

```
function df(f, x) {  
  var dx = 0.001;  
  var dy = f(x+dx) - f(x);  
  return dy/dx;  
}  
  
function square(x) {  
  return x*x;  
}
```

```
console.log('df(x^2,2)='+df(square, 2));  
console.log('df(x^2,2)='+df(function(x){ return x*x; }, 2));  
console.log('df(sin(x/4),pi/4)='+df(Math.sin, 3.14159/4));
```

執行結果

```
D:\Dropbox\cccweb\db\js\code>node df.js  
df(x^2,2)=4.0009999999999699  
df(x^2,2)=4.0009999999999699  
df(sin(x/4),pi/4)=0.7067535793015001
```

# 微分

檔案：**df.c**

```
#include <stdio.h>
#include <math.h>

double dx = 0.001;

double df(double (*Fx)(double), double x) {
    double dy = f(x+dx) - f(x);
    return dy/dx;
}

double square(double x) {
    return x*x;
}

int main() {
    printf("df(x^2,2)=%f\n", df(square, 2.0));
    printf("df(sin(x),pi/4)=%f\n", df(sin, 3.14159/4));
}
```

執行結果

```
D:\Dropbox\cccwd\db\c\code>gcc df.c -o df

D:\Dropbox\cccwd\db\c\code>df
df(x^2,2)=4.001000
df(sin(x/4),pi/4)=0.706754
```



# 微分

檔案：**df.js**

```
function df(f, x) {  
  var dx = 0.001;  
  var dy = f(x+dx) - f(x);  
  return dy/dx;  
}  
  
function square(x) {  
  return x*x;  
}
```

```
console.log('df(x^2,2)='+df(square, 2));  
console.log('df(x^2,2)='+df(function(x){ return x*x; }, 2));  
console.log('df(sin(x/4),pi/4)='+df(Math.sin, 3.14159/4));
```

執行結果

```
D:\Dropbox\cccweb\db\js\code>node df.js  
df(x^2,2)=4.0009999999999699  
df(x^2,2)=4.0009999999999699  
df(sin(x/4),pi/4)=0.7067535793015001
```

# 積分

檔案：**integral.c**

```
#include <stdio.h>
#include <math.h>
#define dx 0.001

double integral(double (*f)(double), double a, double b) {
    double x, area = 0.0;
    for (x=a; x<b; x=x+dx) {
        area = area + f(x)*dx;
    }
    return area;
}

double square(double x) {
    return x*x;
}

int main() {
    printf("integral(x^2,0,1)=%f\n", integral(square,0,1));
    printf("integral(sin(x),0,pi)=%f\n", integral(sin,0,3.14159));
}
```

執行結果

```
D:\Dropbox\cccwd\db\c\code>gcc integral.c -o integral

D:\Dropbox\cccwd\db\c\code>integral
integral(x^2,0,1)=0.332834
integral(sin(x),0,pi)=2.000000
```

## 遞迴範例 1

範例：**recursive.js**

```
function sum(n) {  
    var s=0;  
    for (var i=1; i<=n; i++)  
        s += i;  
    return s;  
}  
  
function s(n) {  
    if (n==1) return 1;  
    return s(n-1)+n;  
}  
  
function f(n) {  
    if (n==0) return 1;  
    if (n==1) return 1;  
    return f(n-1)+f(n-2);  
}  
  
var log = console.log;  
log("f(5)=%d", f(5));  
log("sum(10)=%d", sum(10));  
log("s(10)=%d", s(10));
```

## 簡化後的版本

檔案：**s10.js**

```
function s(n) {
    if (n==1) return 1;
    var sn = s(n-1)+n;
    return sn;
}

console.log("s(10)=%d", s(10));
```

執行結果：

```
nqu-192-168-61-142:code mac020$ node s10
s(10)=55
```

## 追蹤遞迴過程

將函數 s(n) 修改如下以印出中間結果，並觀察執行過程：

檔案：**sum\_recursive.js**

```
function s(n) {
    if (n==1) return 1;
    var sn = s(n-1)+n;
    console.log("s(%d)=%d", n, sn);
    return sn;
}

console.log("s(10)=%d", s(10));
```

執行結果：

```
$ node sum_recursive
s(2)=3
s(3)=6
s(4)=10
s(5)=15
s(6)=21
s(7)=28
s(8)=36
s(9)=45
s(10)=55
s(10)=55
```

## C 的遞迴函數

### 用遞迴計算 $s(n)=1+..+n$

檔案：**s10.c**

```
#include <stdio.h>

int s(int n) {
    if (n==1) return 1;
    int sn = s(n-1)+n;
    return sn;
}

int main() {
    printf("s(10)=%d", s(10));
}
```

執行結果：

```
D:\Dropbox\cccwd\db\c\code>gcc s10.c -o s10

D:\Dropbox\cccwd\db\c\code>s10
s(10)=55
```

### 印出中間過程

檔案：**recursive.c**

```
#include <stdio.h>

int s(int n) {
    if (n==1) return 1;
    int sn = s(n-1)+n;
    printf("s(%d)=%d\n", n, sn);
    return sn;
}

int main() {
    printf("s(10)=%d", s(10));
}
```

執行結果：

```
D:\Dropbox\cccwd\db\c\code>gcc recursive.c -o recursive

D:\Dropbox\cccwd\db\c\code>recursive
s(2)=3
s(3)=6
s(4)=10
s(5)=15
s(6)=21
s(7)=28
s(8)=36
s(9)=45
s(10)=55
s(10)=55
```

## 字典 (物件)

雖然上述這些 JavaScript 的語法很像 C 語言，但是 JavaScript 本質上仍然是個動態語言，其特性比較像 Python、Ruby 等語言，因此預設就有提供更高階的資料結構，其中最重要的一個就是字典 (dictionary)，字典中的基本元素是 (key, value) 的配對，我們只要將 key 傳入就可以取得 value 的值，以下是一個 JavaScript 的字典範例。

檔案：**dict.js**

```
var dict={
  name:"john",
  age:30
};

dict["email"] = "john@gmail.com";
dict.tel = "02-12345678";

for (var key in dict) {
  console.log("key=", key, " value=", dict[key]);
}

console.log("age=", dict.age);
console.log("birthday=", dict["birthday"]);
```

執行結果

```
D:\js\code>node dict.js
key= name  value= john
key= age  value= 30
key= email  value= john@gmail.com
key= tel  value= 02-12345678
age= 30
birthday= undefined
```



## 範例：字典 -- dict.c

C 語言沒有物件，也沒有字典結構與對應的函式庫。

C 語言可以說是高階語言裡面的低階語言，很多東西你都要自己來實作，或者採用其他人所寫的函式庫。

人員查詢 -- 姓名年齡分成兩個陣列

以下是一個實現用《姓名》搜尋《年齡》的範例，這種搜尋法是線性搜尋，比較慢但程式相對簡單！

檔案：dict.c

```
#include <stdio.h>
#include <string.h>

int size = 3;
char *name[] = { "john", "mary", "george" };
int age[] = { 20, 30, 40 };

int findPeople(char *pName, int pSize) {
    int i;
    for (i=0; i<size; i++) {
        if (strcmp(name[i], pName)==0) {
            return i;
        }
    }
    return -1;
}

int main() {
    int mi = findPeople("mary", size);
    if (mi < 0) {
        printf("not found!\n");
    } else {
        printf("people[%d]: name=%s, age=%d\n", mi, name[mi], age[mi]);
    }
}
```

執行結果

```
D:\Dropbox\cccwd\db\c\code>gcc dict.c -o dict
```

```
D:\Dropbox\cccwd\db\c\code>dict
```

```
people[1]: name=mary, age=30
```

## 人員查詢 -- 使用結構 (struct)

以下改使用結構 `struct` 來儲存人的姓名和年齡，結構和物件有點像，可以用 `object.data` 存取資料，但是通常不會將函數宣告在結構裡面。(因為 C 沒有支援物件的觀念，所以試圖模仿物件導向時，寫起來會很囉唆且麻煩)。

檔案：**dict2.c**

```
#include <stdio.h>
#include <string.h>

#define SIZE 3

typedef struct {
    char *name;
    int age;
} People;

People peoples[] = {
    { .name="john", .age=20},
    { .name="mary", .age=30},
    { .name="george", .age=40}
};

int findPeople(char *pName, int pSize) {
    int i;
    for (i=0; i<pSize; i++) {
        if (strcmp(peoples[i].name, pName)==0) {
            return i;
        }
    }
    return -1;
}

int main() {
    int mi = findPeople("mary", SIZE);
    if (mi < 0) {
        printf("not found!\n");
    } else {
        printf("people[%d]: name=%s, age=%d\n", mi, peoples[mi].name, p
    }
}
```

執行結果

```
D:\Dropbox\cccwd\db\c\code>gcc dict2.c -o dict2
```

```
D:\Dropbox\cccwd\db\c\code>dict2
```

```
people[1]: name=mary, age=30
```

## 英翻中系統

程式：**mt.js**

```
var e2c = { dog:"狗", cat:"貓", a: "一隻", chase:"追", eat:"吃" };

function mt(e) {
  var c = [];
  for (i in e) {
    var eword = e[i];
    var cword = e2c[eword];
    c.push(cword);
  }
  return c;
}

var c = mt(process.argv.slice(2));
console.log(c);
```

執行結果：

```
$ node mt a dog chase a cat
[ '一隻', '狗', '追', '一隻', '貓' ]
```

## 英翻中系統

程式：**mt.c**

```
#include <stdio.h>
#include <string.h>

char *e[] = {"dog", "cat", "a", "chase", "eat", NULL};
char *c[] = {"狗", "貓", "一隻", "追", "吃", NULL};

int find(char *nameArray[], char *name) {
    for (int i=0; nameArray[i] != NULL; i++) {
        if (strcmp(nameArray[i], name)==0)
            return i;
    }
    return -1;
}

void mt(char *words[], int len) {
    for (int i=0; i<len; i++) {
        int ei = find(e, words[i]);
        if (ei < 0)
            printf(" _ ");
        else
            printf(" %s ", c[ei]);
    }
}

int main(int argc, char *argv[]) {
    mt(&argv[1], argc-1);
}
```

執行結果：

```
$ gcc mt.c -std=c99 -o mt
```

```
$ ./mt a dog chase a cat
```

```
一隻 狗 追 一隻 貓
```



# 英中互翻

檔案：**mt2.js**

```
var log = console.log;

var dic = { dog:"狗", cat:"貓", a: "一隻", chase:"追", eat:"吃",
           "狗":"dog", "貓":"cat", "一隻":"a", "追":"chase", "吃":"eat"

function mt(w) {
  var array = [];
  for (i in w) {
    var word = w[i];
    var toWord = dic[word];
    array.push(toWord);
  }
  return array;
}

var a = mt(process.argv.slice(2));
log(a);
```

執行結果

```
nqu-192-168-61-142:code mac020$ node mt2.js a dog chase a cat
[ '一隻', '狗', '追', '一隻', '貓' ]
nqu-192-168-61-142:code mac020$ node mt2.js 一隻 狗 追 一隻 貓
[ 'a', 'dog', 'chase', 'a', 'cat' ]
nqu-192-168-61-142:code mac020$ node mt2.js 一隻 狗 chase 一隻 貓
[ 'a', 'dog', '追', 'a', 'cat' ]
nqu-192-168-61-142:code mac020$ node mt2.js 一隻 狗 咬 一隻 貓
[ 'a', 'dog', undefined, 'a', 'cat' ]
```

# 英中互翻系統

程式：**mt2.c**

```
#include <stdio.h>
#include <string.h>

char *e[] = {"dog", "cat", "a", "chase", "eat", NULL};
char *c[] = {"狗", "貓", "一隻", "追", "吃", NULL};

int find(char *nameArray[], char *name) {
    int i;
    for (i=0; nameArray[i] != NULL; i++) {
        if (strcmp(nameArray[i], name)==0) {
            return i;
        }
    }
    return -1;
}

void mt(char *words[], int len) {
    int i;
    for (i=0; i<len; i++) {
        int ei = find(e, words[i]);
        int ci = find(c, words[i]);
        if (ei >= 0) {
            printf(" %s ", c[ei]);
        } else if (ci >= 0) {
            printf(" %s ", e[ci]);
        } else {
            printf(" _ ");
        }
    }
}

int main(int argc, char *argv[]) {
    mt(&argv[1], argc-1); // 從 argv (例如 : mt a dog chase a cat) 中取
```

執行結果：

```
D:\Dropbox\cccwd\db\c\code>gcc mt2.c -o mt2
```

```
D:\Dropbox\cccwd\db\c\code>mt2 a dog chase a cat  
一隻 狗 追 一隻 貓
```

```
D:\Dropbox\cccwd\db\c\code>mt2 一隻 狗 追 一隻 貓  
a dog chase a cat
```

```
D:\Dropbox\cccwd\db\c\code>mt2 a 狗 chase 一隻 cat  
一隻 dog 追 a 貓
```

## 習題：中翻英系統

這題和 [英翻中系統](#) 有個差別，因為中文的詞彙間沒有空白，因此要查詢詞彙需要一些技巧。

由於中文詞彙通常在四個字以下，所以可以嘗試從 4 字，3 字，2 字，1 字 這樣一直查下來，查到就進行翻譯動作。

這種方法隱含了長詞優先規則。

用法範例：

```
$ node c2e.js 一隻狗追一隻貓  
a dog chase a cat
```

## 解答 1：c2e.js

```
var log = console.log;

var dic = { "狗":"dog", "貓":"cat", "一隻":"a", "追":"chase", "吃":"e"

function mt(s) {
  var array = [];
  for (var i=0; i<s.length; i++) {
    for (var len=4; len>0; len--) {
      var str = s.substr(i, len);
      var toWord = dic[str];
      if (typeof toWord !== "undefined") {
        array.push(toWord);
        break;
      }
    }
  }
  return array;
}

var a = mt(process.argv[2]);
log(a);
```

上面版本會列出所有詞的翻譯，但有可能重疊，所以我又改成了下列版本：

## 解答 2：c2e2.js

```

var log = console.log;

var dic = { "狗":"dog", "貓":"cat", "一隻":"a", "追":"chase", "吃":"e"

function mt(s) {
  var array = [];
  for (var i=0; i<s.length; ) {
    for (var len=4; len>0; len--) {
      var str = s.substr(i, len);
      var toWord = dic[str];
      if (typeof toWord !== "undefined") {
        array.push(toWord);
        break;
      }
    }
    i=i+Math.max(1, len);
  }
  return array
}

var a = mt(process.argv[2]);
log(a);

```

# 中翻英系統

程式：**c2e.c**

```
#include <stdio.h>
#include <string.h>

char *e[] = {"dog", "cat", "a", "chase", "eat", NULL};
char *c[] = {"狗", "貓", "一隻", "追", "吃", NULL};

int find(char *nameArray[], char *name) {
    int i;
    for (i=0; nameArray[i] != NULL; i++) {
        if (strcmp(nameArray[i], name)==0) {
            return i;
        }
    }
    return -1;
}

// 注意，一個中文字佔兩個 byte，也就是兩個 char
void mt(char *s) {
    int i, len;
    for (i=0; i<strlen(s); ) {
        for (len=8; len>0; len-=2) {
            char word[9];
            strncpy(word, &s[i], 9);
            word[len] = '\0';
            int ci = find(c, word);
            if (ci >= 0) {
                printf(" %s ", e[ci]);
                i+=len;
                break;
            }
        }
        if (len <=0) {
            printf(" _ ");
            i+=2; // 跳過一個中文字
        }
    }
}
```



```

    }
}
}

int main(int argc, char *argv[]) {
    mt(argv[1]); // 從 argv (例如：mt 一隻狗追一隻貓) 中取出參數一 (例如：一
}

```

執行結果：

```

D:\Dropbox\cccwd\db\c\code>gcc c2e.c -o c2e

D:\Dropbox\cccwd\db\c\code>c2e 一隻狗
a dog

D:\Dropbox\cccwd\db\c\code>c2e 一隻狗追一隻貓
a dog chase a cat

```

### 習題：自動產生英文語句

提示：先用簡單的幾個字加上基本語法就行了，不用一下企圖心太大。

#### 簡易語法

```
S = NP VP
NP = DET N
VP = V NP
N = dog | cat
V = chase | eat
DET = a | the
```

#### 產生過程的範例

```
S = NP VP = (DET N) (V NP)
  = (a dog) (chase DET N)
  = a dog chase a cat
```

解答：

```
function rand(a,b) {
    return a+Math.random()*(b-a);
}

function randInt(a,b) {
    return Math.floor(a+Math.random()*(b-a));
}

function randSelect(a) {
    return a[randInt(0,a.length)];
}
/*
for (var i=0; i<10; i++) {
    var animal = randSelect(['dog', 'cat']);
    console.log("%s", animal);
}
*/
/*
```

```

S = NP VP
NP = DET N
VP = V NP
N = dog | cat
V = chase | eat
DET = a | the
*/

function S() {
    return NP()+" "+VP();
}

function NP() {
    return DET()+" "+N();
}

function VP() {
    return V()+" "+NP();
}

function N() {
    return randSelect(["dog", "cat"]);
}

function V() {
    return randSelect(["chase", "eat"]);
}

function DET() {
    return randSelect(["a", "the"]);
}

console.log(S());

```

## 自動產生英文語句

程式：**gen.c**

```
#include <stdio.h>
#include <stdlib.h>

char *randPrint(char *a[], int size) {
    int r = rand()%size;
    printf(" %s ", a[r]);
}

/*
S = NP VP
NP = DET N
VP = V NP
N = dog | cat
V = chase | eat
DET = a | the
*/

char *n[] = {"dog", "cat"};
void N() {
    randPrint(n, 2);
}

char *v[] = {"chase", "eat"};
void V() {
    randPrint(v, 2);
}

char *det[] = {"a", "the"};
void DET() {
    randPrint(det, 2);
}

void NP() { DET(); N(); }
```

```
void VP() {    V(); NP(); }

void S() { NP(); VP(); }

int main() {
    srand(time(NULL));
    S();
}
```

執行結果：

```
D:\Dropbox\cccwd\db\c\code>gcc gen.c -o gen
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
the cat chase the cat
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
a dog eat the dog
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
the dog eat the cat
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
a cat eat a cat
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
a cat eat the dog
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
the dog eat the dog
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
the dog eat the dog
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
the cat chase a cat
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
the dog eat a dog
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
a dog eat a cat
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
a cat eat the cat
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
the cat eat a dog
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
a dog eat a dog
```

```
D:\Dropbox\cccwd\db\c\code>gen
```

```
the dog chase a cat
```

### 模組定義：**math.js**

```
var math = {  
  PI:3.14,  
  square:function(n) {  
    return n*n;  
  }  
}  
  
module.exports = math;
```

接著您可以使用 `require` 這個指令動態的引入該模組，注意 `require` 必須採用相對路徑，即使在同一個資料夾底下，也要加上 `./` 的前置符號，代表在目前資料夾之下。以下是一個引用上述模組的範例。

### 模組使用：**mathTest.js**

```
var m=require("./math");  
  
console.log("PI=%d square(3)=%d", m.PI, m.square(3));
```

執行結果：

```
D:\Dropbox\Public\pmag\201306\code>node mathTest  
PI=3.14 square(3)=9
```

## 動態模組：匯出建構函數

以下是一個定義圓形 `circle` 的物件。

### 模組定義：**circle.js**

```
var PI = 3.14;
Circle = function (radius) {
    this.radius = radius
    this.area = function() {
        return PI * this.radius * this.radius;
    }
};

module.exports = Circle;
module.exports.PI = PI;
```

在引用「匯出建構函數」的程式當中，由於取得的是建構函數，因此必須再使用 `new` 的方式建立物件之後才能使用 (例如以下的 `c = new cir(5)` 這個指令，就是在透過建構函數 `cir()` 建立物件。

模組使用：**CircleTest.js**

```
var cir = require('./circle');           // 注意，./ 代表 circle
var c = new cir(5);
console.log("cir.PI="+cir.PI);
console.log("c.PI="+c.PI);
console.log("c.area( )="+c.area());
```

執行結果

```
D:\code\node>node circleTest.js
cir.PI=3.14
c.PI=undefined
c.area( )=78.5
```

您現在應該可以理解為何我們要將 `Circle` 定義為一個函數了吧！這只不過 `Circle` 類別的建構函數而已，當他被 `module.exports = Circle` 這個指令匯出時，就可以在 `var cir = require('./circle')` 這個指令 接收到建構函數，然後再利用像 `var c = new cir(5)` 這樣的指令，呼叫該建構函數，以建立出物件。



然後，您也應該可以看懂為何我們要用 `module.exports.PI = PI` 將 `PI` 獨立塞到 `module.exports` 裏了吧！因為只有這樣才能讓外面的模組在不執行物件建構函數 (不建立物件) 的情況之下就能存取 `PI`。

跨平台模組：定義各種平台均能使用的 JavaScript 模組

在很多開放原始碼的 JavaScript 專案模組中，我們會看到模組的最後有一段很複雜的匯出動作。舉例而言，在 `marked.js` 這個將 Markdown 語法轉為 HTML 的模組最後，我們看到了下列這段感覺匪夷所思的匯出橋段，這種寫法其實只是為了要讓這個模組能夠順利的在「瀏覽器、`node.js`、CommonJS 與其 Asynchronous Module Definition (AMD) 實作版的 RequireJS」等平台當中都能順利的使用這個模組而寫的程式碼而已。

```
/**
 * Expose
 */

marked.Parser = Parser;
marked.parser = Parser.parse;

marked.Lexer = Lexer;
marked.lexer = Lexer.lex;

marked.InlineLexer = InlineLexer;
marked.inlineLexer = InlineLexer.output;

marked.parse = marked;

if (typeof exports === 'object') {
  module.exports = marked;
} else if (typeof define === 'function' && define.amd) {
  define(function() { return marked; });
} else {
  this.marked = marked;
}

}).call(function() {
  return this || (typeof window !== 'undefined' ? window : global);
})();
```

對這個超複雜匯出程式有興趣的朋友可以看看以下的文章，應該就可以大致理解這種寫法的來龍去脈了。

- <http://www.angrycoding.com/2012/10/cross-platform-wrapper-function-for.html>

# C 語言的模組化 – 學校沒教的那些事兒！ -- 以矩陣相加為例

有朋友在「程式人雜誌」社團上問到有關 C 語言的參數傳遞問題，其討論帖如下：

- <https://www.facebook.com/groups/programmerMagazine/permalink/622690621080991/> 筆者想藉此問題在此進行釐清與回答。

## 問題 1：C 語言的模組化問題

- 吳秉穎：不好意思可以問一個小問題嗎？
  - 老師給我的題目:以同一程式主類別之外部二維矩陣相加方式將 Sales A&B 每個月相加，其結果再以不在同一程式方式與 Sales C 每個月相加和輸出結果。
  - 因為要寫 C 跟JAVA，JAVA我是寫完了，但是不懂 C 要以不在同一程式方式寫是什麼意思？C 可以這樣寫嗎？
- 陳鍾誠：就是用兩個以上的 .c 檔，加上 .h 檔的方式，例如：main.c module1.c module2.c module1.h module2.h 的方式。

## 問題 2：矩陣參數與回傳值問題

- 吳秉穎：不好意思 請問一下 陣列怎回傳到主程式

```
#include <iostream>
#include<stdio.h>
#include<stdlib.h>
/* run this program using the console pauser or add your own getch,
int sum(int arr[4][3],int arr1[4][3]);
int sum1(int arr[4][3],int arr1[4][3]);
void show(int arr[4][3]);
int main(int argc,char *argv[]) {
    int a[4][3],b[4][3],c[4][3],d[4][3];
    printf("請輸入A&B&C報表: \n");
    for(int i=0;i<4;i++)
    {
```

```

        for(int j=0;j<3;j++)
        {
            scanf("%d",&a[i][j]);
            scanf("%d",&b[i][j]);
            scanf("%d",&c[i][j]);
            d[i][j]=a[i][j]+b[i][j];
        }
        printf("\n");
    }
    sum1(a,b);
    sum(c,d);
    system("pause");
    return 0;
}

int sum1(int arr[4][3],int arr1[4][3])
{
    int i,j,d[4][3];
    for(i=0;i<4;i++)
    {
        for(j=0;j<3;j++)
        {
            d[i][j]=arr[i][j]+arr1[i][j];
        }
    }
    show(d);
    //return (d[4][3]);
}

```

- - 吳秉穎：現在卡在是 我想要把 SUM1 的 D 再傳到 SUM 裡跟 C 家再一起
- 李○○：你在函數參數內再放入一個陣列指標參數 將arr + arr1的值指派給他 不用迴傳就可以拿到此值了
- 吳秉穎：恩我試試看
- 李○○：你的函數宣告有很大的問題 參數型太沒人這樣用的吧@@
- 李○○：感覺你對於記憶體概念跟指標不是很熟
- 吳秉穎：對呀!還在學 ...
- 若○○：你丟指標過去到函式裡面就可以傳陣列了，2 維 = (ptr)

- 若○○：
- 在 c 用標準的 stdio 涵式要小心很容易會爆掉 memory, 我舉例：scanf <= 這個涵式是要靠 \n 或是 \0 斷尾，不然它會無止盡的去換算下去... 就是說如果你在 a[y][x] 裡面的值不是標準的 byte 或是整數的結尾的話，那麼... %d 可能會轉成亂七八糟的東西出來，爆 mem 機率很大，而 & 取指符號的用法是要一開始用的，我不知道在 a[y][x] 的地方可不可以取址，因為 c 沒有很高階，要先在 &a 這裡取址，然後 a[y][x] 的位置要用算的，= (ptr[x] ptr[y])，不知道我這樣說對不對... \n 二維的意思是方形的區塊，但是在 mem 裡面是 (x y) 這樣代表 a[x][y]，所以一般都會先把維給轉成指標再去運算，在宣告的時候 int a[y][x] 這樣的用法在 c 語言是每一個 a[i][j] 都是 int 的區斷，並不是整個 int 如果你用 sizeof() 去取得就是得到 int (x y) 那麼大的一塊 mem 區域，然後 a 和 b 的 array 不能夠直接去做 sum 運算... \n 因為 int (x y) 在 c 裡面是看成單獨的 int (x \* y) 而不是一整個可以放單一個數值的，不知道我這樣會不會越說越亂... 最後要回傳的值，必須要符合你一開始給函數所宣告的遞迴值，你用 int 去宣告，就要符合 int 型態，你可以傳 int 的指標。
- 劉○○：請問樓上的大大 我該如何釐清所有程式的概念阿..
- 若○○：好問題... 我也弄不懂...
- 劉敦桐：我光 C 就1個頭兩個大了
- 若○○：我說一個經驗談好了，asm 是最基礎的機械語言，而 c 語言是把 asm 語言做成很多的 marco 巨集處理出來的二階語言，而更高階的 c++ 和 c# 語言則是將 c + 標準涵式庫再做一次 macro 處理的三階語言，所以在處理 memory 上有極高效率與穩定的特性，因此複雜度會高許多，但是因為高階語言所能夠容納的標處理器是更多，因此更貼合人類語言，所以感覺比較好學一點，其實許多觀念會比 asm 來的複雜不簡單的特性，我本身的感覺是透過基本的機械語言和 c 能夠去更了解實體的高階語言可能會有一些不一樣的想法...
- 吳秉穎 還有一些程式範例是 屏科老師的 我去他網站抓的
- 若○○ <http://programming.im.ncnu.edu.tw/Chapter9.htm>
- 若○○ \*(p+10) = 100; // 相當於x[10] = 100
- 劉○○ 恩Q\_Q
- 若○○ 2 維就是 ((ptr))
- 若○○ 用成的就可以指過去
- 若○○ 問一個小問題喔... int a[10]，在 a[1] - a[2] 的 mem 區域是怎麼編排的?! 是 int a[1] + \0 + a[2] 或是 a[1] + a[2] 無空隙?!還是 a[1] 的實際位址在 0x000001 而 a[2] 是在 0xffff00，而 int a[1] 裡面裝的到底是什麼?!0000 0000

0000 0000 ?!

- 李○○ 位置應該會連續吧 我指的是MMU來說 實體記憶體應該不一定會連續 裡面裝的值應該是亂碼嗎? 我有說錯嗎~~
- 陳鍾誠 是  $a[1] + a[2]$  無空隙 紀○○ 陣列是一段連續的記憶體 $a[1]$ - $a[2]$ 的位址距離是`sizeof(int)`的大小
- 陳鍾誠 只有字串才需要以 `\0` 結尾，陣列是連續的空間。
- 陳鍾誠 對於有 MMU 的情況而言，在邏輯位址空間上仍然是連續的。但在實體位址空間就有可能有斷裂的情況。

### 問題 3 :矩陣參數與回傳值問題 (續)

- 吳秉穎：若再不用指標的方式將 $A[4][3]$ 跟 $B[4][3]$ 的兩個二維陣列傳到函數裡相加，也就是

```
for(int i = 0 ; i < 4 ; i++)
{
    for(int j = 0 ; j < 3 ; j++)
    {
        D[i][j]=A[i][j]+B[i][j];
    }
}
```

我要如何將D陣列回傳到main裡，請問這方法可不可行。

- 若○○ : `int arr[10][10] main()`...
- 吳秉穎：謝謝各位解說，所以如果我寫 `void main(void)` 這樣就無法接收回傳值對吧

### 筆者回覆與釐清

許多 C 語言的學習者都會有同樣的問題，但是卻往往學了很久之後還是不知道答案，有時摸索了很久自己找出一個答案，卻不見得就是好的方法。

以上的問題大致可歸結為兩個，第一個是「陣列的參數傳遞問題」，第二個是「程式的模組化問題」。

首先讓我們解答「陣列的參數傳遞問題」。

標準 C 語言的參數傳遞，基本上只有兩種，一種是傳值的參數 (通常用在基本型態上)，另一種是傳遞位址的參數 (通常用在陣列或結構上)。

舉例而言，在下列程式當中，函數 max 的參數 a, b 都是傳遞值的參數。而函數 sum 的參數 len 仍然是傳值的參數，但 a 則是傳遞位址的參數。

檔案：**param.c**

```
#include <stdio.h>

int max(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}

int sum(int a[], int len) {
    int i, s=0;
    for (i=0; i<len; i++) {
        s += a[i];
    }
    return s;
}

int main() {
    int x = 3, y = 5;
    int z = max(x, y);
    int array[] = { 1, 2, 3, 4, 5};
    int s = sum(array, 5);
    printf("x=%d y=%d z=%d s=%d\n", x, y, z, s);
}
```

執行結果：

```
D:\Dropbox\Public\c>gcc param.c -o param
```

```
D:\Dropbox\Public\c>param
```

```
x=3 y=5 z=5 s=15
```

對於傳值的參數，在傳遞時會複製一份該參數傳進函數，因此函數中的任何的修改都將不會影響到原來的正本，因此下列程式中 `modify(a)` 函數呼叫後，`a` 的值仍然是 5，並不會改變。

檔案：**modify1.c**

```
#include <stdio.h>

int modify(int x) {
    x += 3;
}

int main() {
    int a = 5;
    modify(a);
    printf("a=%d\n", a);
}
```

執行結果：

```
D:\Dropbox\Public\c>gcc modify1.c -o modify1

D:\Dropbox\Public\c>modify1
a=5
```

但是對於傳遞位址的參數，如果我們採用類似的方法進行修改，那麼其內容將會被改變，因此下列程式中的 `modify(a)` 執行完之後，`a[0]` 的值將會變成 8。

檔案：**modify2.c**



```
#include <stdio.h>

int modify(int x[]) {
    x[0] += 3;
}

int main() {
    int a[] = { 5 };
    modify(a);
    printf("a[0]=%d\n", a[0]);
}
```

執行結果：

```
D:\Dropbox\Public\c>gcc modify2.c -o modify2

D:\Dropbox\Public\c>modify2
a[0]=8
```

事實上、傳址參數也有進行參數複製後傳遞的動作，但複製的是「記憶體位址」，而非參數內容，假如上述程式中 `a` 的記憶體位址為 100，那麼當 `modify(a)` 被呼叫時，會將 100 複製一份後傳給 `x`，因此當 `x[0] += 3` 執行的時候，事實上是將記憶體位址 100 的內容 5 取出，然後加上 3，於是記憶體位址 100 的內容就變成了 8，因此在程式執行完之後，`a[0]` (也就是記憶體位址 100 的內容) 的值也就變成 8 了。

對於上述的程式，如果我們改成指標的寫法，那麼程式碼將改寫如下：

```
#include <stdio.h>

int modify(int *x) {
    x[0] += 3;
}

int main() {
    int a[] = { 5 };
    modify(a);
    printf("a[0]=%d\n", a[0]);
}
```

執行結果：

```
D:\Dropbox\Public\c>gcc modify3.c -o modify3

D:\Dropbox\Public\c>modify3
a[0]=8
```

您可以看到這兩個版本，除了 `int x[]` 改為 `int *x` 之外，其餘的部分完全沒有改變，因為在 C 語言當中，陣列形態的參數事實上就是用「位址」的方式傳遞的，而這也正是傳址參數的底層實作方式。

接著讓我們看看矩陣參數的傳遞問題，首先讓我們看看以下的二維陣列寫法。

檔案：**matrix1.c**

```
#include <stdio.h>

#define ROWS 4
#define COLS 3

void print(char *name, double M[ROWS][COLS]) {
    int i, j;
    if (name != NULL)
        printf("===== %s =====\n", name);
    for(i = 0; i < ROWS; i++) {
        for(j = 0; j < COLS; j++) {
            printf("%4.1f ", M[i][j]);
        }
        printf("\n");
    }
}

void add(double A[ROWS][COLS], double B[ROWS][COLS], double M[ROWS][COLS]) {
    int i, j;
    for(i = 0 ; i < ROWS ; i++) {
        for(j = 0 ; j < COLS ; j++) {
            M[i][j]=A[i][j]+B[i][j];
        }
    }
}

int main() {
    double X[ROWS][COLS] = { {1, 2, 3}, {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };
    double Y[ROWS][COLS] = { {1, 1, 1}, {1, 1, 1}, {1, 1, 1}, {1, 1, 1} };
    double Z[ROWS][COLS];

    add(X,Y,Z);
    print("X", X);
    print("Y", Y);
    print("Z", Z);
}
```

執行結果

```
D:\Dropbox\Public\c\matrix>gcc matrix1.c -o matrix1
```

```
D:\Dropbox\Public\c\matrix>matrix1
```

```
===== X =====
1.0  2.0  3.0
1.0  2.0  3.0
1.0  2.0  3.0
1.0  2.0  3.0
===== Y =====
1.0  1.0  1.0
1.0  1.0  1.0
1.0  1.0  1.0
1.0  1.0  1.0
===== Z =====
2.0  3.0  4.0
2.0  3.0  4.0
2.0  3.0  4.0
2.0  3.0  4.0
```

您可以看到這樣的寫法感覺好像很正常，但事實上卻很沒有彈性，因為陣列大小都定死了，沒有辦法同時宣告 43 的陣列與 35 的陣列。

假如我們改用以下寫法，彈性就會大多了，因為可以宣告任意大小的二維陣列。

檔案：**matrix2.c**

```
#include <stdio.h>

void matrixPrint(char *name, double *M, int rows, int cols) {
    int i, j;
    if (name != NULL)
        printf("===== %s =====\n", name);
    for(i = 0; i < rows; i++) {
        for(j = 0; j < cols; j++) {
            printf("%4.1f ", M[i*cols+j]);
        }
        printf("\n");
    }
}
```

```

void add(double *A, double *B, double *M, int size) {
    int i;
    for(i = 0 ; i < size ; i++)
        M[i] = A[i] + B[i];
}

#define matrixAdd(A, B, M, rows, cols) add(A, B, M, rows*cols)

int main() {
    double X[4][3] = { {1, 2, 3}, {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };
    double Y[4][3] = { {1, 1, 1}, {1, 1, 1}, {1, 1, 1}, {1, 1, 1} };
    double Z[4][3];
    double *x = X[0], *y = Y[0], *z = Z[0];

    matrixAdd(x, y, z, 4, 3);
    matrixPrint("X", x, 4, 3);
    matrixPrint("Y", y, 4, 3);
    matrixPrint("Z", z, 4, 3);

    double A[2][2] = { {1, 2}, {3, 4} };
    double B[2][2] = { {1, 1}, {1, 1} };
    double C[2][2];
    double *a = A[0], *b = B[0], *c = C[0];

    matrixAdd(a, b, c, 2, 2);
    matrixPrint("A", a, 2, 2);
    matrixPrint("B", b, 2, 2);
    matrixPrint("C", c, 2, 2);
}

```

註：讀者可能會感覺到奇怪，為何我們用 `double x = X[0]` 這樣的語法，筆者原本是寫成 `double x = (double) X` 這樣的方式，但是這樣的語法在 `gcc` 中是正確的，但有人在 `Visual C++` 當中編譯就錯了。而用 `double x = X[0]` 這樣寫，因為 `X` 是二維陣列，`X[0]` 自然就是一維陣列，與 `double *x` 的形態相容，而且位址也對，所以就採用了這種寫法。

執行結果：

```
D:\Dropbox\Public\c\matrix>gcc matrix2.c -o matrix2
```

```
D:\Dropbox\Public\c\matrix>matrix2
```

```
===== X =====
1.0  2.0  3.0
1.0  2.0  3.0
1.0  2.0  3.0
1.0  2.0  3.0
===== Y =====
1.0  1.0  1.0
1.0  1.0  1.0
1.0  1.0  1.0
1.0  1.0  1.0
===== Z =====
2.0  3.0  4.0
2.0  3.0  4.0
2.0  3.0  4.0
2.0  3.0  4.0
===== A =====
1.0  2.0
3.0  4.0
===== B =====
1.0  1.0
1.0  1.0
===== C =====
2.0  3.0
4.0  5.0
```

現在我們應該已經回答完 C 語言參數的傳遞問題了，接著讓我們來回答第二個問題，也就是 C 語言的模組化問題。

針對以上程式，我們通常不應該將主程式與矩陣函數撰寫在同一個檔案裏面，因此我們可以分開成兩個檔案：main.c 與 matrix.c 如下所示：

檔案：**matrix.c**

```
void matrixPrint(char *name, double *M, int rows, int cols) {
    int i, j;
    if (name != NULL)
        printf("===== %s =====\n", name);
    for(i = 0; i < rows; i++) {
        for(j = 0; j < cols; j++) {
            printf("%4.1f ", M[i*cols+j]);
        }
        printf("\n");
    }
}

void add(double *A, double *B, double *M, int size) {
    int i;
    for(i = 0 ; i < size ; i++)
        M[i] = A[i] + B[i];
}

#define matrixAdd(A, B, M, rows, cols) add(A, B, M, rows*cols)
```

檔案：**main.c**

```

#include <stdio.h>
#include "matrix.c"

int main() {
    double X[4][3] = { {1, 2, 3}, {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };
    double Y[4][3] = { {1, 1, 1}, {1, 1, 1}, {1, 1, 1}, {1, 1, 1} };
    double Z[4][3];
    double *x = X[0], *y = Y[0], *z = Z[0];

    matrixAdd(x, y, z, 4, 3);
    matrixPrint("X", x, 4, 3);
    matrixPrint("Y", y, 4, 3);
    matrixPrint("Z", z, 4, 3);

    double A[2][2] = { {1, 2}, {3, 4} };
    double B[2][2] = { {1, 1}, {1, 1} };
    double C[2][2];
    double *a = A[0], *b = B[0], *c = C[0];

    matrixAdd(a, b, c, 2, 2);
    matrixPrint("A", a, 2, 2);
    matrixPrint("B", b, 2, 2);
    matrixPrint("C", c, 2, 2);
}

```

編譯時您只要將兩個檔案放在同一個資料夾，就可以用下列指令編譯完成後執行，執行結果仍然與原本一樣。

```
gcc main.c -o matrix
```

matrix 雖然以上的方式已經可以「分開撰寫、合併編譯」，但是仍然有一些缺陷，那就是每次要使用矩陣函式庫時，都要用 `#include "matrix.c"` 指令將整個 `matrix.c` 引入並且重新編譯，這樣會造成編譯速度緩慢的問題，因此若要加快速度，可以將 `matrix.c` 先行編譯為目的檔 `matrix.o`，然後再將 `matrix.c` 當中的函數原型抽取出來獨立成一個檔案 `matrix.h`，這樣不用每次都勞動編譯器重新編譯 `matrix.c` 檔，而且可以讓編譯器將 `main.c` `matrix.h` `matrix.o` 等檔案順利編譯連結成執行檔。



因此我們可以將上述程式改寫如下。

檔案：**matrix.h**

```
#define matrixAdd(A, B, M, rows, cols) add(A, B, M, rows*cols)

void matrixPrint(char *name, double *M, int rows, int cols);
void add(double *A, double *B, double *M, int size);
```

檔案：**matrix.c**

```
#include <stdio.h>
#include "matrix.h"

void matrixPrint(char *name, double *M, int rows, int cols) {
    int i, j;
    if (name != NULL)
        printf("===== %s =====\n", name);
    for(i = 0; i < rows; i++) {
        for(j = 0; j < cols; j++) {
            printf("%4.1f ", M[i*cols+j]);
        }
        printf("\n");
    }
}

void add(double *A, double *B, double *M, int size) {
    int i;
    for(i = 0 ; i < size ; i++)
        M[i] = A[i] + B[i];
}
```

檔案：**main.c**

```
#include <stdio.h>
#include "matrix.h"

int main() {
    double X[4][3] = { {1, 2, 3}, {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };
    double Y[4][3] = { {1, 1, 1}, {1, 1, 1}, {1, 1, 1}, {1, 1, 1} };
    double Z[4][3];
    double *x = X[0], *y = Y[0], *z = Z[0];

    matrixAdd(x, y, z, 4, 3);
    matrixPrint("X", x, 4, 3);
    matrixPrint("Y", y, 4, 3);
    matrixPrint("Z", z, 4, 3);

    double A[2][2] = { {1, 2}, {3, 4} };
    double B[2][2] = { {1, 1}, {1, 1} };
    double C[2][2];
    double *a = A[0], *b = B[0], *c = C[0];

    matrixAdd(a, b, c, 2, 2);
    matrixPrint("A", a, 2, 2);
    matrixPrint("B", b, 2, 2);
    matrixPrint("C", c, 2, 2);
}
```

然後用下列指令編譯並執行 (其中的 -c 參數用來告訴編譯器只要編譯成目的檔就好，不需要進一步連結成執行檔)。

```
gcc -c matrix.c -o matrix.o

gcc main.c matrix.o -o main

main
```

但是上述程式可能還會引起一些小問題，因為有些編譯器可能會對重複引用的行為產生警告或編譯錯誤的形況，因此最好再加上引用防護，將 matrix.h 改寫如下。

檔案：**matrix.h**

```
#ifndef __MATRIX_H__
#define __MATRIX_H__

#define matrixAdd(A, B, M, rows, cols) add(A, B, M, rows*cols)

void matrixPrint(char *name, double *M, int rows, int cols);
void add(double *A, double *B, double *M, int size);

#endif
```

如此就完成了將矩陣函數模組化的動作，事先將 `matrix.c` 編譯成目的檔 `matrix.o`，以縮短編譯時間，並且方便 使用者引用。

當然、如果您有很多模組，分別都編譯成目的檔，最好還是用 `ar` 指令將這些目的合併成函式庫，這樣就不用每次編譯 都要引用一大堆目的檔，只要引用對應的函式庫就可以了。

## 結語

對於初學 C 語言的朋友們而言，參數傳遞的誤解與型態的混淆是很常見的問題，不會使用 `*.h` 或撰寫引用防護也是很正常的事情。C 語言是一個很困難的語言 (事實上、C 語言是筆者所學過的語言當中，我認為最困難的一個。不過最近發現 JavaScript 這個語言 有後來居上的趨勢 -- 或者說有些語法更不容易理解)，所以初學者在學習 C 語言時感到困難是很正常的事情。

我還記得自己在先前寫的一份網誌電子書上寫下了這一段話，在此與讀者分享：

當我還是一個大學生的時候，總覺得 C 語言就是這樣了。但是在 10 年後我進入職場時，才發現原來我並不太認識這個語言。產業界所使用的 C 語言有許多是大學所沒有教授過的，像是 `#ifdef`、`make`、GNU 工具等等。又過了 10 年，當我研究嵌入式系統時，這個感覺又出現了，我仍然不太認識 C 語言，嵌入式系統中所使用的「記憶體映射輸出入、`volatile`、組合語言連接、Link Script」等，又讓我耳目一新，我再度重新認識了 C 語言一次。然後，當我研讀 Linux 核心的程式碼時，看到 Torvalds 所使用的「鏈結串列、行程切換技巧」等，又再度讓我大為驚訝，C 語言竟然還可以這樣用。然後，當我開始研究 Google Android 手機平台的架構時，又看到了如何用 C 語言架構出網路、視窗、遊戲、瀏覽器等架構，於是我必須再度學習一次 C 語言。

當我翻閱坊間的書籍時，不禁如此想著，如果有人能直接告訴我這些 C 語言的學習歷程，那應該有多好。難道，我們真的必需花上數十年的時間去學習 C 語言，才能得到這些知識嗎？這些知識在初學者的眼中，看來簡直像是「奇技淫巧」。然而這些「奇技淫巧」，正是 C 語言為何如此強大的原因，我希望能透過這本書，告訴各位這些「奇技淫巧」，讓各位讀者不需要再像我一樣，花上二十年功夫，才能學會這些技術。

在我的眼中，C 語言就像一把鋒利的雙面刃，初出茅廬的人往往功力不夠深厚，反而將這個神兵利器往自己身上砍，因而身受重傷。但是在專家的手中，C 語言卻具有無比的威力，這種神兵利器具有「十年磨一劍、十步殺一人」的驚人力量。筆者希望能透過這本書，讓讀者能夠充分發揮 C 語言的力量，快速的掌握這個難以駕馭的神兵利器。

## 參考文獻

- 免費電子書：高等 C 語言(專業限定版) -- <http://ccckmit.wikidot.com/cp:main>

## 補充

在本文預覽版刊出之後，柏諺兄在 程式人雜誌社團 當中寫了一段精準的解說，於是我將這段解說放在這裏，作為補充：

在 C 語言裡面沒有 "傳遞陣列" 給函式的概念，一直都是 call by value.

## 預備知識

當你在將 陣列名稱 傳遞給函式或是拿去作數值運算([], +)時，你的運算元實際上是 "第一個元素的位址值"，我們在語言裡稱它為 decay(退化). 陣列不是指標，但能夠退化成指標.

在 C/C++ 中型別是很重要的，例如陣列 `int array[ 2 ][ 3 ]`: (array) 的數值跟 `array[ 0 ]`、`&array[ 0 ][ 0 ]` 一樣，但是他們各自擁有不同的型別，在運算上的差異甚大. 若以 "數值" 來理解這個語言那麼有可能在撰寫跨平台程式時產生非預期的bugs.

陣列和函式的型態跟一般內建型態不同，是在變數的左右兩側，例如一維陣列: `int array[ 10 ]`, array 自己本身的型態是 `int[ 10 ]`, 而成員型態則是 `int`; 二維陣列: `int matrix[ 4 ][ 3 ]`, matrix 本身的型態是 `int[ 4 ][ 3 ]`, 成員型態則是 `int[ 3 ]` (所代表的意義是: matrix 是一個 `int[ 3 ]` 陣列, 大小為 4 )

有這樣的理解我們就可以看懂更複雜的語法:

```
int add( int, int ), abs( int );
```

上面程式碼是在 "宣告" 兩個函式: 1) 第一個是 add(), 有兩個 int 參數, 回傳值型態是 int 2) 第二個是 abs(), 只有一個 int 參數, 回傳值型態是 int

因為宣告時每個名稱能夠共用的只有型態的左邊部分, 但是右邊部分無法共用所以必須分開寫.

本文開始

在範例中的函式宣告(a):

```
void print(char *name, double M[ROWS][COLS]);
```

有的人說 "最高維可以忽略", 所以延伸了第二種宣告方式(b):

```
void print(char *name, double M[][COLS]);
```

但最正確的宣告則是第三種(c):

```
void print(char *name, double (*M)[COLS]);
```

三者被 名稱修飾(Name Mangling) 出來的結果是一樣的, 但是你卻無法傳遞 int[ ROWS+1 ][ COLS ] 給 (a) --- 因為撰碼的人在宣告的地方自己加上最高維的限制使然.

當叫用 print( "Y", Y ) 時, Y 會被 decay(退化) 成指標, 因為 Y 的型態是 double[ROWS][COLS], 它的成員型態是 double[COLS], 那麼第一個元素的位址型態就是 double(\*)[COLS] 所以我們參數應該寫成 (c) 而不是其它兩種. 一樣的概念適用在任何維度的陣列.

在 print() 內可以直接使用 M[ i ][ j ] 的語法來存取成員, M[ i ] 是對傳進來的連續 double[COLS] 中取第 i 個小陣列, [ j ] 則是拿出小陣列的第 j 個值, 想要達到陣列 slice 也不無可能.

用不適當的方法去傳遞多維陣列導致 `M[i*cols+j]` 這寫法反而有可能拖慢間接位址計算的速度.

順帶一提下面的宣告:

```
void matrixPrint(char *name, double *M, int rows, int cols);
```

語意上是 "給予一塊連續的 `double` 記憶體, 並且用額外的必要資訊 `rows`, `cols` 來存取該記憶體". 這邊我們看到的就只剩數個 `double` 小個體而不是整體概念 "矩陣", 所以傳遞的時候最好用 `struct` 包裝起來 "語意" 才會一樣.

檔案：**complex.js**

```
function add(c1, c2) {
    return { r:c1.r+c2.r, i:c1.i+c2.i };
}

function sub(c1, c2) {
    return { r:c1.r-c2.r, i:c1.i-c2.i };
}

function mul(c1, c2) {
    return { r:c1.r*c2.r-c1.i*c2.i,
            i:c1.r*c2.i+c1.i*c2.r };
}

function toStr(c) {
    return c.r+" "+c.i+"i";
}

var o1 = { r:1, i:2 }, o2={ r:2, i:1 };

var add12 = add(o1, o2);
var sub12 = sub(o1, o2);
var mul12 = mul(o1, o2);

var c = console;

c.log("o1=%s", toStr(o1));
c.log("o2=%s", toStr(o2));
c.log("add(c1,c2)=%s", toStr(add12));
c.log("sub(c1,c2)=%s", toStr(sub12));
c.log("mul(c1,c2)=%s", toStr(mul12));
```

## 執行結果

```
D:\Dropbox\cccwd\db\js\code>node complex.js
o1=1+2i
o2=2+1i
add(c1,c2)=3+3i
sub(c1,c2)=-1+1i
mul(c1,c2)=0+5i
```

您可以看到這種寫法也很模組化，看起來相當不錯，只是函數是函數，資料是資料，函數只是用來處理資料的程式，此種寫法還沒有用到物件導向的技術。

接著、讓我們來看一個簡化後的物件導向版本，這個簡化後的版本只有一種運算函數，那就是加法 `add`。

## 物件寫法 1 : `ocomplex1.js`

檔案：`ocomplex1.js`



```
var Complex = {
  add:function(c2) {
    return createComplex(this.r+c2.r, this.i+c2.i);
  }
}

var createComplex=function(r,i) {
  var c = Object.create(Complex);
  c.r = r;
  c.i = i;
  return c;
}

var c = console;

var c1=createComplex(1,2), c2=createComplex(2,1);

var c3 = c1.add(c2).add(c2).add(c2).add(c2);

c.log("c1=%j", c1);
c.log("c2=%j", c2);
c.log("c1.add(c2)=%j", c1.add(c2));
c.log("c3=%j", c3);
```

### 執行結果

```
D:\Dropbox\cccwd\db\js\code>node ocomplex1.js
c1={"r":1,"i":2}
c2={"r":2,"i":1}
c1.add(c2)={"r":3,"i":3}
c3={"r":9,"i":6}
```

在上述程式中，我們透過 `Object.create(Complex)` 創造一個物件之後，立刻在其中塞入 `r`, `i` 等欄位，此時雖然物件看來只有兩個欄位，但事實上還有一些隱藏的物件資訊沒有被印出來，其中的一個隱藏物件資訊就是原型，在 JavaScript 中的物件都有一個原型 `prototype` 的欄位，這個欄位在執行完 `Object.create(Complex)` 後，會指向 `Complex` 物件。

```
var createComplex=function(r,i) {  
    var c = Object.create(Complex);  
    c.r = r;  
    c.i = i;  
    return c;  
}
```

上述程式中我們在 `log()` 函數中用了 `%j` 的格式，這代表要將該物件以 `json` 的方式印出來。

因此當我們後來呼叫 `c1.add(c2)` 這樣的指令時，JavaScript 的解譯系統才能夠從 `c1` 這個物件中找到 `add` 這個欄位，然後將其當成函數使用。

這種用點符號 `.` 串起來的寫法可以一直串下去，成為一種串式語法，因此我們可以用 `c1.add(c2).add(c2).add(c2).add(c2)` 進行連續的加法。

## 物件寫法 2 : `ocomplex2.js`

在物件的原型 `prototype` 裏通常還有些其他未顯示出來的函數，像是 `toString()` 就是一個很好用的函數，假如我們為物件加上 `toString()` 函數的話，那麼在該物件需要被轉換成字串的時候，就會自動呼叫 `toString()`，以下是我們為上述 `ocomplex1.js` 程式加上 `toString()` 函數之後的結果，這個版本稱為 `ocomplex2.js`。

檔案：**`ocomplex2.js`**

```
var Complex = {
  add:function(c2) {
    return createComplex(this.r+c2.r, this.i+c2.i);
  },
  toString:function() {
    return this.r+" "+this.i+"i"
  }
}

var createComplex=function(r,i) {
  var c = Object.create(Complex);
  c.r = r;
  c.i = i;
  return c;
}

var c = console;

var c1=createComplex(1,2), c2=createComplex(2,1);

var c3 = c1.add(c2).add(c2).add(c2).add(c2);

c.log("c1=%s", c1);
c.log("c2=%s", c2);
c.log("c1.add(c2)=%s", c1.add(c2));
c.log("c3=%s", c3);
```

### 執行結果

```
D:\Dropbox\cccwd\db\js\code>node ocomplex2.js
c1=1+2i
c2=2+1i
c1.add(c2)=3+3i
c3=9+6i
```

您可以看到由於我們加入了 `toString()` 函數，而且在印出來的語法上採用了 `%s` 這個字串式印法，於是在印到螢幕前 `console.log` 會先呼叫這些複數物件的 `toString()` 函數，結果印出來的格式就好看很多了。

## 物件寫法 3 : ocomplex3.js

當然、我們也可以把減法 `sub()` 和乘法 `mul()` 函數放到這個物件導向版的複數程式中，這樣就和前面的非物件導向版功能相當了，以下是這個比較完整的版本。

檔案：**ocomplex3.js**

```
var Complex = {
  add:function(c2) {
    return createComplex(this.r+c2.r, this.i+c2.i);
  },
  sub:function(c2) {
    return createComplex(this.r-c2.r, this.i-c2.i);
  },
  mul:function(c2) {
    return createComplex(this.r*c2.r-this.i*c2.i,
                        this.r*c2.i+this.i*c2.r);
  },
  toString:function() {
    return this.r+" "+this.i+"i"
  }
}

var createComplex=function(r,i) {
  var c = Object.create(Complex);
  c.r = r;
  c.i = i;
  return c;
}

var c = console;

var c1=createComplex(1,2), c2=createComplex(2,1);

var c3 = c1.add(c2).sub(c2).add(c2).sub(c2);

c.log("c1=%s", c1);
c.log("c2=%s", c2);
c.log("c1.add(c2)=%s", c1.add(c2));
c.log("c1.sub(c2)=%s", c1.sub(c2));
c.log("c1.mul(c2)=%s", c1.mul(c2));
c.log("c3=%s", c3);
```

執行結果

```
D:\Dropbox\cccwd\db\js\code>node ocomplex3.js
c1=1+2i
c2=2+1i
c1.add(c2)=3+3i
c1.sub(c2)=-1+1i
c1.mul(c2)=0+5i
c3=1+2i
```

上述程式雖然很完整了，但是在語法上 `createComplex()` 沒有和 `Complex` 物件直接綁釘在一起，感覺怪怪的。為了讓語法更漂亮，我們乾脆將該函數直接塞回 `Complex` 物件內，成為 `Complex.create()` 函數，這樣感覺就更「物件化」了一些。請看以下的版本！

## 物件寫法 4 : `ocomplex4.js`

檔案：`ocomplex4.js`

```
var Complex = {
  add:function(c2) {
    return Complex.create(this.r+c2.r, this.i+c2.i);
  },
  sub:function(c2) {
    return Complex.create(this.r-c2.r, this.i-c2.i);
  },
  mul:function(c2) {
    return Complex.create(this.r*c2.r-this.i*c2.i,
                          this.r*c2.i+this.i*c2.r);
  },
  toString:function() {
    return this.r+" "+this.i+"i"
  }
}

Complex.create=function(r,i) {
  var c = Object.create(Complex);
  c.r = r;
  c.i = i;
  return c;
}

var c = console;

var c1=Complex.create(1,2), c2=Complex.create(2,1);

var c3 = c1.add(c2).sub(c2).add(c2).sub(c2);

c.log("c1=%s", c1);
c.log("c2=%s", c2);
c.log("c1.add(c2)=%s", c1.add(c2));
c.log("c1.sub(c2)=%s", c1.sub(c2));
c.log("c1.mul(c2)=%s", c1.mul(c2));
c.log("c3=%s", c3);
```

執行結果

```
D:\Dropbox\cccwd\db\js\code>node ocomplex4.js
c1=1+2i
c2=2+1i
c1.add(c2)=3+3i
c1.sub(c2)=-1+1i
c1.mul(c2)=0+5i
c3=1+2i
```

必須注意的是，這種寫法仍然必須把 `Complex.create` 提出來到外面寫，否則在 `Complex` 都尚未創建完成時就要用 `Object.create(Complex)` 創建 `Complex` 物件的話，就會產生錯誤了。

## 結語

以上是我們對 JavaScript 物件導向機制的一個簡單入門，但是並不完整，因為我們還沒有看到更深入的《原型鏈》機制，關於《原型鏈》的概念我們將在後續的文章中再來探討。



# C 的結構與物件

## 整個結構傳遞

C 語言沒有物件導向，只有一種稱為《結構》(struct) 的組織模式，可以讓你把很多個欄位放在一起，形成一種《多欄位結構》。

舉例而言，假如我們要表達《複數》，可以將《實部 r 和虛部 i》組合之後，形成一個《複數結構》，以下是其程式範例：

檔案：**complex.c**

```
#include <stdio.h>

typedef struct {
    double r, i;
} Complex;

Complex add(Complex c1, Complex c2) {
    Complex c;
    c.r = c1.r+c2.r;
    c.i = c1.i+c2.i;
    return c;
}

Complex sub(Complex c1, Complex c2) {
    Complex c;
    c.r = c1.r-c2.r;
    c.i = c1.i-c2.i;
    return c;
}

Complex mul(Complex c1, Complex c2) {
    Complex c;
    c.r = c1.r*c2.r-c1.i*c2.i;
    c.i = c1.r*c2.i+c1.i*c2.r;
    return c;
}
```

```
void print(char *name, Complex c) {
    printf("%s=%6.2f+%6.2fi\n", name, c.r, c.i);
}

int main() {
    Complex o1={ .r=1.0, .i=2.0 };
    Complex o2={ .r=2.0, .i=1.0 };

    print("o1", o1);
    print("o2", o2);

    Complex add12 = add(o1, o2);
    Complex sub12 = sub(o1, o2);
    Complex mul12 = mul(o1, o2);

    print("add(o1,o2)", add12);
    print("sub(o1,o2)", sub12);
    print("mul(o1,o2)", mul12);
}
```

## 執行結果

```
D:\Dropbox\cccwd\db\c\code>gcc complex.c -o complex

D:\Dropbox\cccwd\db\c\code>complex
o1=  1.00+  2.00i
o2=  2.00+  1.00i
add(o1,o2)=  3.00+  3.00i
sub(o1,o2)= -1.00+  1.00i
mul(o1,o2)=  0.00+  5.00i
```

## 只有傳遞指標，不須複製內容

檔案：**complex.c**

```
#include <stdio.h>
```

```
typedef struct {
    double r, i;
} Complex;

Complex add(Complex *c1, Complex *c2) {
    Complex c;
    c.r = c1->r+c2->r;
    c.i = c1->i+c2->i;
    return c;
}

Complex sub(Complex *c1, Complex *c2) {
    Complex c;
    c.r = c1->r-c2->r;
    c.i = c1->i-c2->i;
    return c;
}

Complex mul(Complex *c1, Complex *c2) {
    Complex c;
    c.r = c1->r*c2->r-c1->i*c2->i;
    c.i = c1->r*c2->i+c1->i*c2->r;
    return c;
}

void print(char *name, Complex *c) {
    printf("%s=%6.2f+%6.2fi\n", name, c->r, c->i);
}

int main() {
    Complex o1={ .r=1.0, .i=2.0 };
    Complex o2={ .r=2.0, .i=1.0 };

    print("o1", &o1);
    print("o2", &o2);

    Complex add12 = add(&o1, &o2);
    Complex sub12 = sub(&o1, &o2);
    Complex mul12 = mul(&o1, &o2);
}
```

```
    print("add(o1,o2)", &add12);  
    print("sub(o1,o2)", &sub12);  
    print("mul(o1,o2)", &mul12);  
}
```

### 執行結果

```
D:\Dropbox\cccwd\db\c\code>gcc complex2.c -o complex2
```

```
D:\Dropbox\cccwd\db\c\code>complex2
```

```
o1=  1.00+  2.00i
```

```
o2=  2.00+  1.00i
```

```
add(o1,o2)=  3.00+  3.00i
```

```
sub(o1,o2)= -1.00+  1.00i
```

```
mul(o1,o2)=  0.00+  5.00i
```

問題是，到底指標是甚麼？接下來我們將花比較多的時間，講述這個 C 語言當中令人又愛又恨的奇特概念！

## 字串處理

主題	說明
字串大小的問題	如何決定字串的大小，防止緩衝區溢位。
字串的格式化	printf 與 scanf 都用到的 format 字串，這是 C 語言字串輸出入的核心
sprintf 函數	sprintf 是很好用的格式化工具
sscanf 函數	sscanf 是很好用的字串剖析工具，並且支援類似 Regular Expression 的功能
標準字串函式庫	標準 C 語言的字串大都是靜態函數，也就是不會在函數中分配新的記憶體
字串的誤用	無法決定長度的字串，請不要用標準函式庫，要改用動態字串
動態字串物件	可以動態增長的字串物件，讓您不用再為字串長度傷腦筋
字串函數	寬字串的處理，在 C 語言中，通常寬字串指的是 Unicode (但不限定於 Unicode)
寬窄字串間的轉換	Unicode 轉為一般字串，或將一般字串轉為 Unicode
字串的誤用	無法決定長度的字串，請不要用標準函式庫，要改用動態字串

## 字串大小的問題 -- 如何決定字串的大小，防止緩衝區溢位。

在 C 語言當中，最惱人的莫過於如何決定字串或陣列大小的這個問題了，舉例而言，在下列程式當中，我們宣告 input 大小為 5，但是我們永遠不會知道 5 到底夠不夠，萬一不夠就會造成當機，甚至被有心人士透過「緩衝區溢位」方法攻擊，這是使用 C 語言陣列時經常遇到的困擾。

### 程式一：很容易「緩衝區溢位」的程式

```
#include <stdio.h>

int main() {
    char input[5];
    scanf("%s", input);
    printf("Your input : %s", input);
}
```

即使我們宣告 char input[100] 也有可能不夠，但是如果宣告 char input[10000]，會不會太浪費記憶體了，到底應該如何處理呢？

其實在 scanf 這樣的函數中，可以用 %s 指定大小，以下範例就改進了上述問題，因此當輸入字串長度超過 5 時，就會被截掉，

### 程式二：不會造成「緩衝區溢位」的程式

```
#include <stdio.h>

int main() {
    char input[6]; // 注意，這裡必須宣告 6=5+1，因為還有結束字元 \0。
    scanf("%5s", input);
    printf("Your input : %s", input);
}
```

## 執行結果

```
D:\cp>gcc arraySize.c -o arraySize
```

```
D:\cp>arraySize
```

```
Hello!John.
```

```
Your input : Hello
```

## 字串的格式化

在 C 語言當中，輸出格式化依賴 `printf()` 類的指令，而輸入的格式化則仰賴 `scanf` 類的指令。這兩個函數都用到的 `format` 字串，這是 C 語言字串輸出的核心。

事實上，這兩類指令當中最重要函數是 `sscanf()` 與 `sprintf()`，`sprintf()` 可以將複雜的參數格式化成字串，而 `sscanf()` 函數則是 C 語言版本的正規表達式，幾乎可以做到大部分 Regular Expression 能做到的功能。

在這些函數當中，都會有一個 `char *format` 這樣的格式化參數，其中以 `%` 開頭的稱為格式描述區 (Format specifiers)，格式描述區有複雜的參數型態，稱為描述元 (specifier)，下表說明了描述元當中符號的意義，並列出其使用範例。

符號	說明	範例
c	字元 (char)	a
d	整數 (Decimal integer)	372
i	整數 (Decimal integer) (同 d)	372
f	浮點數 (Floating Point)	372.56
e	科學記號 (Scientific notation)	3.7256e+2
E	科學記號 (Scientific notation)	3.7256E+2
g	取浮點數或科學記號當中短的那個	372.56
G	取浮點數或科學記號當中短的那個	372.56
o	八進位 (Octal Integer)	735
s	字串 (String)	372
u	無號數 (unsigned integer)	372
x	十六進位 (Hexadecimal integer)	3fb
X	十六進位 (Hexadecimal integer)	3FB
p	指標位址	B800:0000
n	不列印, 用來取得目前輸出長度	%n
%	印出 % 符號	%%



另外，有時在格式描述區當中還會指定變數長度欄位，說明取出後的變數長度，數字類型的欄位才會需要這樣指定，欄位的內容可能為 h, l 或 L，其意義如下所示。

長度符號	說明
h	2 byte 短整數 short int (針對 i, d, o, u, x and X)
l	4 byte 長整數 int , long (針對 i, d, o, u, x and X)
L	8 byte 浮點數 double (e, E, f, g and G)

## **sprintf** 函數 — **sprintf** 是很好用的格式化工具。

`sprintf()` 函數是C 語言用來格式化的主要方法，其函數原形如下所示。由於其中的格式化參數稍微複雜，因此很多學習者並不知道該如何正確的使用這些格式化參數。

```
int sprintf ( char * str, const char * format, ... );
```

str: 格式化後的輸出字串

format: 格式化的規格字串

在 format 字串中, 以 % 起頭者為格式段落, 其格式語法如下:

格式段落的語法: %[flags][width][.precision][length]specifier

% 代表變數開始

[flag]

-: 靠左

+: 輸出正負號

(space): 當不輸出正負號時, 就輸出空白

#: 在 8 或 16 進位 (o, x, X) 時, 強制輸出 0x 作為開頭,

在浮點數 (e, E, f) 時, 強制輸出小數點,

在浮點數 (g, G) 時, 強制輸出小數點, 但尾端的 0 會被去掉。

0: 在開頭處(左側) 補 0, 而非補空白。

[width]

最小輸出寬度 (或 \*)

[.precision]

精確度, 小數點之後的輸出位數

[length]

長度符號 h, l, L

[specifier]

型態描述元, 可以是 c, d, e, E, f, g, G, o, s, u, x, X 等基本型態。

sprintf() 函數的用法與 printf() 很類似, 只是第一個參數為輸出字串 str 而已

檔案: printf.c

```
#include <stdio.h>

int main()
{
    printf ("Characters: %c %c \n", 'a', 65);
    printf ("Decimals: %d %ld\n", 1977, 650000L);
    printf ("Preceding with blanks: %10d \n", 1977);
    printf ("Preceding with zeros: %010d \n", 1977);
    printf ("Some different radixes: %d %x %o %#x %#o \n", 100, 100,
    printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
    printf ("Width trick: %*d \n", 5, 10);
    printf ("%s \n", "A string");
    return 0;
}
```

以上程式的輸出結果：

```
D:\cp\code>gcc printf.c -o printf

D:\cp\code>printf
Characters: a A
Decimals: 1977 650000
Preceding with blanks:          1977
Preceding with zeros: 0000001977
Some different radixes: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick:    10
A string
```

## 參考文獻

- <http://www.cplusplus.com/reference/clibrary/cstdio/sprintf/>

## sscanf 函數

Sscanf 是很好用的字串剖析工具，並且支援類似 Regular Expression 的功能。

C 語言中的 scanf 函數，是初學者都會使用的，但也是大部分人都會誤用，或者是無法充分發揮其功能的。

C 語言的 sscanf() 與 sprintf() 這兩個函數，採用的是一種既創新又好用的設計法，

事實上，函數 sscanf() 比 scanf() 更為好用，sscanf() 甚至支援了類似 Regular Expression 的功能，可以讓我們輕易的剖析格式化的字串。

sscanf 的函數原形如下，其中的 format 格式字串具有複雜的格式指定功能，以下我們將詳細說明這些格式的用途。

```
int sscanf ( const char * str, const char * format, ...);
```

str : 被剖析的字串

format: 剖析格式

在 format 字串中, 以 % 起頭者為剖析段落, 通常在剖析完成後會指定給後面的變數, :

剖析段落的語法 : %[\*][width][modifiers]type

% 代表變數開始

\* 代表省略不放入變數中

width 代表最大讀取寬度

modifier 可以是 {h|I|L} 之一

說明 : 其中 h 代表 2 byte 的變數 (像 short int),

I 代表 4 byte 的變數 (像 long int),

L 是 8 byte 的變數 (像 long double)

type 則可以是 c, d,e,E,f,g,G,o, s, u, x, X 等基本型態,

也可以是類似 Regular Expression 的表達式。

說明: c : 字元 (char);

d : 整數 (Decimal integer);

f : 浮點數 (Floating Point);

e,E : 科學記號 (Scientific notation);

g,G : 取浮點數或科學記號當中短的那個;

o : 八進位 (Octal Integer);

u : 無號數 (unsigned integer);

x, X : 十六進位 (Hexadecimal integer)

為了說明 sscanf 函數的用法, 我們寫了以下程式, 以示範 format 欄位的各種寫法。

檔案 : **sscanf.c**

```
#include <stdio.h>

int main() {
    char name[20], tel[50], field[20], areaCode[20], code[20];
    int age;
    sscanf("name:john age:40 tel:082-313530", "%s", name);
    printf("%s\n", name);
    sscanf("name:john age:40 tel:082-313530", "%8s", name);
    printf("%s\n", name);
    sscanf("name:john age:40 tel:082-313530", "%[^:]", name);
    printf("%s\n", name);
    sscanf("name:john age:40 tel:082-313530", "%[^:]:%s", field, name);
    printf("%s %s\n", field, name);
    sscanf("name:john age:40 tel:082-313530", "name:%s age:%d tel:%s",
    printf("%s %d %s\n", name, age, tel);
    sscanf("name:john age:40 tel:082-313530", "%*[^:]:%s %*[^:]:%d %*",
    printf("%s %d %s\n", name, age, tel);

    char protocol[10], site[50], path[50];
    sscanf("http://ccckmit.wikidot.com/cp/list/hello.txt",
        "%[^:]:%*2[/]%[^/]/%[a-zA-Z0-9._/-]",
        protocol, site, path);
    printf("protocol=%s site=%s path=%s\n", protocol, site, path);
    return 1;
}
```

其編譯執行結果如下所示。

```
D:\oc>gcc sscanf.c -o sscanf
```

```
D:\oc>sscanf
```

```
name:john
```

```
name:joh
```

```
name
```

```
name john
```

```
john 40 082-313530
```

```
john 40 082-313530
```

```
protocol=http site=ccckmit.wikidot.com path=cp/list/hello.txt
```

## 參考文獻

- <http://www.cplusplus.com/reference/cstdio/sprintf/>
- <http://www.cplusplus.com/reference/cstdio/sscanf/>



# 標準字串函式庫 — (String Library) 標

準 C 語言的字串大都是靜態函數，也就是不會在函數中分配新的記憶體。

## K&R 對字串的設計理念

為何 C 語言要設計出像 `strcpy()`, `strcat()`, `strtok()`, `strcmp()`, `strlen()` 這樣的字串函式庫，而不直接使用像上述的動態字串函式庫取代就好了呢？關於這個問題，我們必須回到當初 K & R 兩人設計 C 語言的初始環境，才能看出其原因。

K & R 設計 C 語言時，還沒有物件導向的概念，因此不太可能設計出像範例五這樣具有物件導向概念的字串函式庫。當時電腦的記憶體極為有限，而且 K&R 一心只想設計出 UNIX 作業系統，因此才設計出像 `strcpy()`、`strcmp()` 這樣的函數，可以同時支援字串陣列與指標。

K & R 所設計的字串函式庫可不是用來支援應用程式的，而是用在設計作業系統上的，特別是在基本文字檔的處理上。當時的環境不像我們現在這麼奢侈，有好幾 GB 的記憶體可以用，幾乎可以將任何文字檔都全部讀到記憶體中再進行處理，而是必須一個字一個字的讀，或者是一行一行的讀，因此這些靜態的字串函式庫就已經相當夠用了。

這些字串函式庫所處理的對象，就是從檔案中讀出來，不超過一行長度的字串，就當時的環境，我們可以假設檔案中任一行的長度不會超過 127 個字元，因此您只要宣告像 `char line[128]` 這樣的變數，就足以處理任何的文字檔了。

理解了這個背景，相信讀者應該能夠感覺到為何 K&R 沒有發展出動態字串函式庫了吧？

但是在 C 語言漫長的發展過程中，為何動態字串函式庫沒有被納入標準函式庫中，這或許才是真正的問題，也是學習者之所以誤用 C 語言的原因之所在，關於這點，或許牽涉到了太多的歷史，筆者也無從考證起了。

## 來自 jserv 的建議

code => 沒有「C 語言的字串函式庫」的說法 整體是 C Library:

[http://www.tutorialspoint.com/cstandardlibrary/string\\_h.htm](http://www.tutorialspoint.com/cstandardlibrary/string_h.htm)

==> "但是在 C 語言漫長的發展過程中，為何動態字串函式庫沒有被納入標準函式庫中，" 這句值得商榷 /code

## 字串的誤用 — (String Misuse)

無法決定長度的字串，請不要用標準函式庫，要改用動態字串。

在 Java 這樣的語言當中，字串的長度是可以改變的，您可能會使用下列程式，很自由的讓字串大小增大，Java 會自動的幫您分配字串所需要的記憶空間，而不會產生太大的問題。

### 範例一、字串連接的 Java 程式

```
String s = "";  
for (i=0; i<100; i++)  
    s = s + token[i];
```

但是在 C 語言當中，您就會遇到一個困擾，假如我要撰寫一個類似的程式，那麼字串 s 的長度應該要設定為多長呢？請看下列範例。

### 範例二、字串連接的 C 程式

```
char s[1000];  
for (i=0; i<100; i++)  
    strcat(s, token[i]);
```

您可能會懷疑，長度 1000 到底夠不夠呢？假如 token 陣列中的字串長度平均超過 10 個字，那麼 s 的長度 1000 就會不夠了。這樣看來，Java 的字串函式庫設計似乎比 C 語言要好得多了，不是嗎？

如果您不夠理解 C 語言，就很可能會寫出像範例二這樣的程式，但是這對 C 語言來說其實是一種誤用，誤用的原因是：「想要用靜態的字串處理動態的問題」。

假如您真的必需撰寫像範例二這樣的程式，那麼應該自行設計一個動態的字串函式庫，或者採用某個現成的動態字串函式庫，而不是直接用 C 語言內建的標準函數。但是 C 語言的初學者往往沒有這樣一個函式庫，因而寫出像範例二這樣的程式。

程度稍好的資訊系學生，可能理解到這個問題不能採用靜態的解決方式，因此使用 `malloc()` 函數進行記憶體分配，以解決這個令人困擾的問題，於是就寫出了下列的程式碼。

### 範例三、字串連接的 C 程式 (malloc 版)

```
char *s = malloc(1);
s[0] = '\\0';
for (i=0; i<100; i++) {
    char *t = malloc(strlen(s)+strlen(token[i])+1);
    sprintf(t, "%s%s", s, token[i]);
    free(s);
    s = t;
}
```

但是這樣撰寫 C 語言，仍然是初學者會犯的錯誤之一，這種用法完全誤用了 C 語言的能力，造成了很多次的 `malloc()` 分配，卻又很沒效率的處理字串長度的成長問題。

### 正確的用法

這種情況應該改用動態字串，請進一步下列文章

- 動態字串物件 -- (Dynamic String) 可以動態增長的字串物件，讓您不用再為字串長度傷腦筋。

## 動態字串物件 — (Dynamic String)

可以動態增長的字串物件，讓您不用再為字串長度傷腦筋。

歸根究底，字串誤用的問題，通常是由於 C 語言沒有提供一個標準的動態字串而造成的，如果您真的需要一個這樣的程式，那麼就應該採用一個支援動態字串的函式庫，然後將程式改寫如下。

### 範例、字串連接的 C 程式 (動態字串版)

```
Str *s = StrNew();
for (i=0; i<100; i++) {
    StrAppend(s, token[i]);
}
```

這樣的 C 語言程式，其實就與下列範例中的 Java 程式，看來相差不大了，最大的差別是 C 語言沒有支援物件的概念而已。

範例、字串連接的 Java 程式

```
String s = "";
for (i=0; i<100; i++)
    s = s + token[i];
```

要能撰寫上述這樣的一個程式，動態字串函式庫至少要能支援 StrNew() 與 StrAppend() 這兩個函數，那麼我們應該怎麼做呢？其實，要自己打造這樣一個程式相當容易，筆者可以馬上撰寫一個，如以下範例所示。

### 範例：實作動態字串函式庫

```
typedef struct Str {
    int len, size;
    char *s;
};

Str *StrNew();
void StrAppend(Str *str, char *s);

Str *StrNew() {
    Str *str = malloc(sizeof(Str));
    str->s = malloc(1);
    str->s[0] = '\0';
    str->len = 0;
    str->size = 1;
}

void StrAppend(Str *str, char *s) {
    int newLen = str->len + strlen(s);
    if (newLen+1 > str->size) {
        int newSize = max(str->size*2, newLen+1);
        char *t = malloc(newSize);
        sprintf(t, "%s%s", str->s, s);
        free(str->s);
        str->s = t;
        str->len = newLen;
        str->size = newSize;
    } else {
        strcat(&str->s[str->len], s);
        str->len += strlen(s);
    }
}
```

只要有了這樣一個函式庫，那麼我們就不需要為了 C 語言缺乏動態字串而困擾了，也就不需要每次都寫出難看且沒有效率的程式了，而是直接寫出乾淨，簡潔的函式庫了。

## C 語言中的寬字串 -- 包含 Unicode

要在 C 語言中使用 Unicode 字串，假如您用的是 gcc 編譯器或 Linux，您可以使用寬字元 `wchar_t` 這個形態，以取代 `char`，然後用對應的函數取代原本的字串函數，以下是常見字串函數的寬字元版對應表。

窄字元	寬字元	說明
<code>strlen()</code>	<code>wcslen()</code>	字串長度
<code>strcat()</code>	<code>wcscat</code>	字串連接
<code>strcmp()</code>	<code>wcscmp()</code>	字串比較
<code>strcoll()</code>	<code>wscoll()</code>	字串比較 (不分大小寫)
<code>strcpy()</code>	<code>wscpy()</code>	字串複製
<code>strchr()</code>	<code>wcschr()</code>	尋找字元
<code>strstr()</code>	<code>wcswcs()</code>	尋找字串
<code>strtok()</code>	<code>wcstok()</code>	字串分割
<code>strcspn()</code>	<code>wcscspn()</code>	傳回字串中第一個符合字元集的位置
<code>strpbrk()</code>	<code>wcspbrk()</code>	傳回字串中第一個符合字元集的指標
<code>strxfrm()</code>	<code>wcsxfrm()</code>	根據區域設定 <code>locale()</code> 轉換字元集

根據區域設定 `locale()` 轉換字元集 簡而言之，就是將原本 `strXXX()` 函數，轉換成 `wcsXXX()` 函數，然後照著原本的方法使用，只是對象從 `char` 改為 `wchar_t` 即可，請看下列範例。

### 程式範例：Unicode 寬字串處理函數

檔案：`unicode.c`

```
#include <stdio.h>
#include <locale.h>

int main(void)
{
    if (!setlocale(LC_CTYPE, "")) {
        fprintf(stderr, "Error:Please check LANG, LC_CTYPE, LC_ALL\n");
        return 1;
    }
    wchar_t *str1=L"Hi!你好"; // 輸出結果 (範例)
    printf("str1=%ls\n", str1); // str1=Hi!你好
    printf("wcslen(str1)=%d\n", wcslen(str1)); // wcslen(str1)=5
    printf("wcschr(str1,%lc)=%d\n", L'好', wcschr(str1, L'好')); //
    printf("wcsstr(str1,%ls)=%d\n", L"你好", wcsstr(str1, L"你好")); //
    printf("wcsspncpy(str1,aeiou)=%d\n", wcsspncpy(str1, L"aeiou")); // v
    printf("wcsspncpy(str1,EFGH)=%d\n", wcsspncpy(str1, L"EFGH")); // wcs
    printf("address(str1)=%p\n", str1); // address(str1)=00403030
    printf("wcspbrk(str1,aeiou)=%p\n", wcspbrk(str1, L"aeiou")); //
    wchar_t str2[20];
    wcscpy(str2, str1);
    printf("str2=%ls\n", str2); // str2=Hi!你好
    printf("wcscmp(str1,str2)=%d\n", wcscmp(str1, str2)); // wcscmp
    wcscat(str2, L",我是John");
    printf("str2=%ls\n", str2); // str2=Hi!你好,我是John
    return 0;
}
```

## 執行結果



```
D:\cp>gcc unicode.c -o unicode
```

```
D:\cp>unicode
```

```
str1=Hi!你好
```

```
wcslen(str1)=5
```

```
wcschr(str1,好)=4206648
```

```
wcswcs(str1,你好)=4206646
```

```
wcsspn(str1,aeiou)=0
```

```
wcsspn(str1,EFGH)=1
```

```
address(str1)=00403030
```

```
wcssbrk(str1,aeiou)=00403032
```

```
str2=Hi!你好
```

```
wcscmp(str1,str2)=0
```

```
str2=Hi!你好,我是John
```

## 後記

寬字串的處理函數有很多，並不限於上列的函數，幾乎所有具有字串的標準 C 函數都有寬版，關於更多的寬版函數請參考下列網頁。

[http://www.java2s.com/Tutorial/C/0300\\_\\_Wide-Character-String/WideCharacterFunctions.htm](http://www.java2s.com/Tutorial/C/0300__Wide-Character-String/WideCharacterFunctions.htm)

## 來自 **jserv** 的建議

```
"""
```

寬字串函數 — 寬字串的處理，在 C 語言中，通常寬字串指的是 Unicode（但不限定於

```
"""
```

wide-character 翻譯為「寬字串」，我覺得有本質的問題。

以下摘錄 CLDP：

<http://linux.org.tw/CLDP/OLD/doc/i18n-introduction.html>

"wcs" 是 "wide-chararater string" 的縮寫，而 "mbs" 是 "multi-byte str" 的縮寫，二者分別代表字串的表現方式。所謂的 multi-byte 是指數個 char 組成（一個 char 組成一個 byte），而 wide-char 是指一個 wchar\_t type 就是一個字，而 sizeof(wchar\_t) 的大小與系統有關，一般而言是 4 bytes。一般我們可以直接看、輸出輸入等都是 mul

```
char *str = "這是一個句子：abcd";
```

但我們會建議在程式內部，用 `mbstowcs()` 將它轉成 `wchar_t` 來統一處理，這個轉換 `LC_CTYPE` 的機制，它定義了 multi-byte 與 wide-char 值二者間的對應關係。做這樣轉換的好處是，您不用擔心全形、半形的問題，因為一個 `w`

`wchar_t` 有一組與 `string.h` 中相對應的字串處理函式，就定義在 `wchar.h` 中，讓我們來處理 (`wchar_t *`)，其部分的對應關係如下，其他的可以直接看 `wchar.h` 的內容：

<code>wcscpy()</code>	<====>	<code>strcpy()</code>
<code>wcsncpy()</code>	<====>	<code>strncpy()</code>
<code>wcslen()</code>	<====>	<code>strlen()</code>
<code>wcsdup()</code>	<====>	<code>strdup()</code>
<code>wscmp()</code>	<====>	<code>strcmp()</code>
<code>wcsncmp()</code>	<====>	<code>strncmp()</code>
.....		

由於 `mbs` 碼與 `wcs` 碼的對應關係是由該 `locale` 的 `LC_CTYPE` 來決定的，也就是不寫法其對應關係可能會不一樣。就我們的 `glibc2`, `zh_TW.Big5` `locale` 而言，由 `mb` `unicode` (有關 `unicode` 的資訊可以在 <http://www.unicode.org/> 中找到)，但不能保證在其他的系統或環境下也是如此。故最保險的做法，是將字串儲存成 `mbstowcs()` 轉成 `wide-char` 來運作。

==> 可以看出重點不在於字串 (C 語言的 `string` 只是一個寫法，本質上仍是連續記憶的 `character set` 的「範圍寬度」)

==> 建議保留原文 "wide character"，真要翻譯的話，可寫「擴充字元」

## 參考文獻

- 簡明手冊:使你的C/C++代碼支持Unicode -- <http://www.i18nguy.com/unicode/c-unicode.zh-CN.html>
- Programming with wide characters By Leslie P. Polzer on February 11, 2006 (8:00:00 AM) -- <http://www.linux.com/archive/feature/51836>
- [<http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf> The current Standard (C99 with Technical corrigenda TC1, TC2, and TC3 included) (PDF)], (3.61 MB). Pages 397, 398 and 400.

## 寬窄字串間的轉換

### 程式: **mbstowcs.c**

```
#include <stdlib.h>
#include <stdio.h>

int main(void){
    char *str = "Hi!您好";
    wchar_t wstr[10];
    char str2[10];
    mbstowcs(wstr, str, 10);
    printf("%s\n", str);
    printf("%ls\n", wstr);
    wcstombs(str2, wstr, 10);
    printf("%s", str2);
}
```

### 執行結果

```
D:\cp>gcc mbstowcs.c -o mbstowcs
```

```
D:\cp>mbstowcs
```

```
Hi!您好
```

```
Hi!您好
```

```
Hi!您好
```

## 字串的誤用 — (String Misuse)

無法決定長度的字串，請不要用標準函式庫，要改用動態字串。

在 Java 這樣的語言當中，字串的長度是可以改變的，您可能會使用下列程式，很自由的讓字串大小增大，Java 會自動的幫您分配字串所需要的記憶空間，而不會產生太大的問題。

### 範例一、字串連接的 Java 程式

```
String s = "";  
for (i=0; i<100; i++)  
    s = s + token[i];
```

但是在 C 語言當中，您就會遇到一個困擾，假如我要撰寫一個類似的程式，那麼字串 s 的長度應該要設定為多長呢？請看下列範例。

### 範例二、字串連接的 C 程式

```
char s[1000];  
for (i=0; i<100; i++)  
    strcat(s, token[i]);
```

您可能會懷疑，長度 1000 到底夠不夠呢？假如 token 陣列中的字串長度平均超過 10 個字，那麼 s 的長度 1000 就會不夠了。這樣看來，Java 的字串函式庫設計似乎比 C 語言要好得多了，不是嗎？

如果您不夠理解 C 語言，就很可能會寫出像範例二這樣的程式，但是這對 C 語言來說其實是一種誤用，誤用的原因是：「想要用靜態的字串處理動態的問題」。

假如您真的必需撰寫像範例二這樣的程式，那麼應該自行設計一個動態的字串函式庫，或者採用某個現成的動態字串函式庫，而不是直接用 C 語言內建的標準函數。但是 C 語言的初學者往往沒有這樣一個函式庫，因而寫出像範例二這樣的程式。

程度稍好的資訊系學生，可能理解到這個問題不能採用靜態的解決方式，因此使用 `malloc()` 函數進行記憶體分配，以解決這個令人困擾的問題，於是就寫出了下列的程式碼。

範例三、字串連接的 C 程式 (malloc 版)

```
char *s = malloc(1);
s[0] = '\\0';
for (i=0; i<100; i++) {
    char *t = malloc(strlen(s)+strlen(token[i])+1);
    sprintf(t, "%s%s", s, token[i]);
    free(s);
    s = t;
}
```

但是這樣撰寫 C 語言，仍然是初學者會犯的錯誤之一，這種用法完全誤用了 C 語言的能力，造成了很多次的 `malloc()` 分配，卻又很沒效率的處理字串長度的成長問題。

## 正確的使用法

這種情況應該改用動態字串，請進一步下列文章

動態字串物件 -- (Dynamic String) 可以動態增長的字串物件，讓您不用再為字串長度傷腦筋。

## 指標

主題	說明
指標算術— (Pointer Arithmetics)	指標的加減法，很容易會造成錯誤。
動態陣列物件 — (Dynamic Array)	利用動態陣列，您就不用再為陣列大小傷腦筋了。
陣列大小的問題— (Array Size)	如何決定陣列的大小呢？這是一個惱人的問題。
函數指標— (function pointer)	函數指標是 C 語言當中威力強大的工具，專業人士必定會善用之
函數指標型態 — (function pointer type)	用 typedef 將函數指標宣告成一種型態
變動參數(va_arg)	變動個數參數，也就是使用 ... 宣告的參數，要如何使用呢？

## 指標算術 -- 指標的加減法，很容易會造成錯誤

在 C 語言當中，指標型態的變數，像是下列範例中的 `char cp`; `int ip`; 等，其加減法的表現，會根據型態而有所不同。

舉例而言，假如 `cp = 0x0022FF77`，那麼 `cp+1` 就是 `0x0022FF78`，因為 `cp` 是一種字元指標，這種結果是理所當然的。

但是，假如 `ip = 0x0022FF6C`，那麼 `ip+1` 卻是 `0x0022FF70`，這是因為 `ip` 是整數指標，因此當我們將 `ip` 加上 1 單位距離時，這個一單位距離的大小就相當於整數 `int` 的大小，也就是 `sizeof(int) = 4`。

### 範例程式

```
#include <stdio.h>

int main() {
    char c = 'a';
    char *cp = &c;
    printf("&c=%p\n", &c);
    printf("cp=%p\n", cp);
    printf("cp+1=%p\n", cp+1);
    printf("cp+3=%p\n", cp+3);

    int i = 1;
    int *ip = &i;
    printf("&i=%p\n", &i);
    printf("ip=%p\n", ip);
    printf("ip+1=%p\n", ip+1);
    printf("ip+3=%p\n", ip+3);
}
```

執行結果

```
D:\cp>gcc ptradd.c -o ptradd
```

```
D:\cp>ptradd
&c=0022FF77
cp=0022FF77
cp+1=0022FF78
cp+3=0022FF7A
&i=0022FF6C
ip=0022FF6C
ip+1=0022FF70
ip+3=0022FF78
```

## 習題

這個設計的邏輯，是為了讓您寫程式時，可以都用 `ptr++` 來將指標前進一格，而不需要用 `ptr += sizeof(*ptr)` 這樣複雜的寫法，但是這也造成了一些問題。

假如您的認知錯誤，很可能會多此一舉，萬一您自己計算大小以便調整指標時，就會產生錯誤的結果，像是以下程式一樣。

```
#include <stdio.h>

int main() {
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *ptr = a;
    int i;
    for (i=0; i<3; i++) {
        printf("%d ", *ptr);
        ptr += sizeof(int);
    }
}
```

習題的輸出



```
D:\cp>gcc ptrerror.c -o ptrerror
```

```
D:\cp>ptrerror
```

```
1 5 9
```

## 補充

根據 jserv 的來信補充，提到下列這幾點，是本文原本所沒提到的：

- \* pointer type 的變數可以對一個純量作 '+' 和 '-' 操作
- \* 兩個 pointer type 的變數可以施加 '-' 操作，以得知位移量 (offset)，但不能

意思是說，假設：

```
int a, b;  
void *ptrA = &a, *ptrB = &b;
```

那麼：

```
ptrB - ptrA; // 合法  
ptrA + 1; 合法  
ptrA - 1; 合法  
ptrA + ptrB; // 不合法  
ptrA / 1;  // 不合法
```

## 以 C 語言實作動態陣列

在 C 語言當中，經常會碰到無法事先決定陣列大小的情況，像是實作某些符號表格時，就很難事先決定陣列大小，此時最好使用動態陣列來取代靜態陣列，這些動態陣列會實作自動長大的功能，如此就解決了無法事先決定陣列大小的問題，以下是筆者對動態陣列的一個實作。

在以下程式中，我們學習了 Linux 當中以巨集巡迴鏈結串列的方法，模仿後實作出以巨集巡迴動態陣列的方法，也就是 `ArrayEach()`，這樣的函數可以讓您再陣列元素巡迴時省一點力氣。

程式範例：動態陣列

檔案: **darray.c**

```
#include <stdlib.h>
#include <string.h>

typedef struct {
    int size;      // 陣列目前的上限
    int count;     // 陣列目前的元素個數
    void **item;   // 每個陣列元素的指標
} Array;          // 動態陣列的資料結構

void ArrayNew(Array *array, int size) {
    array->count = 0;
    array->size = size;
    array->item = malloc(array->size*sizeof(void*));
}

void ArrayAdd(Array *array, void *item) {
    if (array->count == array->size) {
        int newSize = array->size*2;
        void **newItems = malloc(newSize*sizeof(void*));
        memcpy(newItems, array->item, array->size*sizeof(void*));
        free(array->item);
        array->item = newItems;
        array->size = newSize;
    }
    array->item[array->count] = item;
    array->count++;
}
```

```
        printf("Array growing : size = %d\n", array->size);
    }
    array->item[array->count++] = item;
}

#define ArrayEach(a, i, o) for (i=0, o=(a)->item[i]; i<(a)->count;

int main() {
    char *names[] = { "John", "Mary", "George", "Bob", "Tony" };
    int i;
    Array array;
    ArrayNew(&array, 1);
    for (i=0; i<5; i++)
        ArrayAdd(&array, names[i]);
    for (i=0; i<array.count; i++)
        printf("name[%d]=%s\n", i, (char*) array.item[i]);
    void *obj;
    ArrayEach(&array, i, obj) {
        printf("name[%d]=%s\n", i, (char*) obj);
    }
}
```

### 執行結果

D:\cp>gcc darray.c -o darray

D:\cp>darray Array growing : size = 2 Array growing : size = 4 Array growing : size = 8 name[0]=John name[1]=Mary name[2]=George name[3]=Bob name[4]=Tony  
name[0]=John name[1]=John name[2]=Mary name[3]=George name[4]=Bob

## 陣列大小的問題

在 C 語言當中，陣列大小是很難決定的問題，因為一般的 C 語言並沒有提供垃圾蒐集機制。

解決這個問題的方法有兩種，第一種是採用如前述的動態陣列，如此陣列大小會隨著您加入的元素多少而自動成長，因而能夠解決這個問題。

第二種方法是不要採用 malloc 的動態分配機制，而是改用靜態的陣列進行宣告，如此您就必須能夠明確的知道陣列所能容納的元素個數。舉例而言，假如您要用一個陣列儲存一年中每一天所花費的錢，那麼就可以用 `int money[366]`; 這樣的宣告，因為您早已知道最大的陣列大小為 366，因此可以很容易的決定。

如果不是這種很明確的情形，而是要用來儲存像符號表這樣的表格，那筆者建議您不要使用靜態陣列，改採前述的動態陣列來實作，您會發現這種作法比硬去宣告一個很大的陣列好多了。

當您還需要位陣列大小傷腦筋時，應該就是想錯方向了，請改用動態陣列。

## C 語言中的函數指標

```
#include <stdio.h>

int add(int a, int b) {
    return a+b;
}

int mult(int a, int b) {
    return a*b;
}

int main() {
    int (*op)(int a, int b);
    op = add;
    printf("op(3,5)=%d\n", op(3,5));
    op = mult;
    printf("op(3,5)=%d\n", op(3,5));
}
```

執行結果：

```
D:\cp\code>gcc fpointer.c -o fpointer

D:\cp\code>fpointer
op(3,5)=8
op(3,5)=15
```

## 函數指標型態 -- (function pointer type) 用 **typedef** 將函數指標宣告成一種型態

我們希望定義一個具有一個參數的函數指標形態，則可以採用下列方式。

```
typedef void(F1)(void);
```

### 程式範例：函數指標的型態

```
#include <stdio.h>

typedef int(*OP)(int,int);

int add(int a, int b) {
    return a+b;
}

int mult(int a, int b) {
    return a*b;
}

int main() {
    OP op = add;
    printf("op(3,5)=%d\n", op(3,5));
    op = mult;
    printf("op(3,5)=%d\n", op(3,5));
}
```

執行結果

```
D:\cp>gcc fpointertype.c -o fpointertype
```

```
D:\cp>fpointertype
```

```
op(3,5)=8
```

```
op(3,5)=15
```

## 變動參數

### 範例：印出整數串列

```
#include <stdio.h>
#include <stdarg.h>

void printList(int head, ... ) {
    va_list va;
    va_start(va, head);
    int i;
    for(i=head ; i != -1; i=va_arg(va,int)) {
        printf("%d ", i);
    }
    va_end(va);
}

int main( void ) {
    printList(3, 7, 2, 5, 4, -1);
}
```

### 執行結果

```
D:\cp>gcc vaarg.c -o vaarg
```

```
D:\cp>vaarg
```

```
3 7 2 5 4
```

### 範例



```
#include <stdio.h>

int debug(const char *fmt, ...)
{
    va_list args;
    va_start(args, fmt);
    return vprintf(fmt, args);
}

int main() {
    debug("pi=%6.2f\n", 3.14159);
}
```

執行結果：

```
D:\cp>gcc debug.c -o debug
```

```
D:\cp>debug
```

```
pi=  3.14
```

## 結構

主題	說明
<a href="#">結構的初始化 — (Initialization)</a>	C 語言可以直接設定整個結構的欄位初始值
<a href="#">結構中的位元欄</a>	位元欄位，讓您位每個位元取名字
<a href="#">結構的指標算術</a>	利用結構存取欄位，或從欄位計算結構起始點
鏈結串列	
-- <a href="#">鏈結串列-基礎版</a>	最簡單的單向鏈結串列實作
-- <a href="#">鏈結串列：內含物件版</a>	通用性鏈結串列，一般性的寫法
-- <a href="#">鏈結串列：外包物件版</a>	模仿 Linux 核心的作法的鏈結串列。

## 結構的初始化 — (Initialization) 直接設定欄位初始值

這個範例程式為 C99 的《指定器初始化》( Designated Initializers), 使用 gcc 時建議加上 -std=C99 的參數。

程式範例

```
#include <stdio.h>

typedef struct {
    char *name;
    int age;
} person;

int main() {
    person p = {
        .name = "John",
        .age = 40
    };

    printf("%s is %d years old", p.name, p.age);
}
```

執行結果

```
D:\cp\code>gcc structInit.c -o structInit

D:\cp\code>structInit
John is 40 years old
```

## 結構中的位元欄 — (Bits Field) 讓您為每個位元取名字

為了展示 C 語言中位元欄位的用途，筆者將拙著《系統程式》中 CPU0 處理器的狀態暫存器 (Status Word) 拿來做為範例，該狀態暫存器的位元配置如下圖所示。

圖一、CPU0 的狀態暫存器 (SW) 之位元配置

程式範例：結構中的位元欄位

```
#include <stdio.h>
```

// 請注意，寫在前面的欄位會被編在低位元部份，因此通常我們必須倒著寫，也就是將圖  
typedef struct

```
{
    unsigned lo:4; // 像是 unsigned hi, lo:4;
    unsigned hi:4; // 請注意，不能把兩個欄位寫在同一行
} byte;
```

```
typedef struct
```

```
{
    unsigned m:1;
    unsigned b2:5;
    unsigned t:1;
    unsigned i:1;
    unsigned b1:20;
    unsigned v:1;
    unsigned c:1;
    unsigned z:1;
    unsigned n:1;
} StatusWord;
```

```
int main() {
```

```
    byte b = { .hi=0x0F, .lo=0x0C };
    unsigned char *bp = (unsigned char*) &b;
    printf("byte=%02x\n", *bp);
```

```
    StatusWord sw = { .n=1, .z=0, .c=1, .v=0, .i=1, .t=1, .m=0, .b1=0, .b2=0
    unsigned int *psw = (unsigned int*) &sw;
    printf("SW=%08x\n", *psw);
```

```
}
```

執行結果

```
D:\cp>gcc structBits.c -o structBits
```

```
D:\cp>structBits
```

```
byte=fc
```

```
SW=a00000c0
```

## C 語言結構中的指標算術

在 C 語言當中，指標的內容就是記憶體位址，於是我們可以利用指標的算術，計算出某些具有特殊價值的數字，然後進行位址操作，以便定址到我們想要的內容上。

舉例而言，在 Linux 的鏈結串列中，就定義了下列的 `offsetof()` 巨集函數，可以讓我們計算出一個結構與其欄位間的距離，

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

事實上，`offsetof()` 這個巨集已經被納入到 ANSI C 的表頭中，您也可以引用該表頭，而不需要自己實作。

如果我們要從欄位指標反求其母結構位址，則可參考下列 Linux 核心中的原始碼，使用 `container_of()` 函數

```
#define container_of(ptr, type, member) ({      \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *) ( (char *)__mptr - offsetof(type,member) );})
```

這樣的巨集函數要如何使用呢？請參考下列範例。

### 程式範例：結構中的指標運算

檔案：**structptr.c**

```
#include <stdio.h>

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)

#define container_of(ptr, type, member) ({      \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *) ( (char *)__mptr - offsetof(type,member) );})

typedef struct {
    char name[20];
```

```

    int age;
} Person;

// 以 Person 中的 age 欄位為例，說明 container_of() 函數的作法
// container_of(&p.age, Person, age)
//     typeof( ((Person *)0)->age ) is int
//     int *__mptr = (&p.age);
//     (Person *) ((char *) __mptr - offsetof(Person, age))

int main() {
    Person p = {
        .name="John",
        .age=40
    };

    size_t offset = offsetof(Person, age);
    printf("offset=0x%x\n", offset);
    printf("offsetof(Person,age)=0x%x\n", offsetof(Person, age));
    printf("&p=%p\n", &p);
    printf("&p.age=%p\n", &p.age);
    printf("p.age=%d\n", *(&p.age));
    printf("&p+1=%p\n", &p+1);
    printf("&((Person*)&p)[1]=%p\n", &((Person*)&p)[1]);
    char *cptr = (char*) &p;
    printf("cptr+1=%p\n", cptr+1);
    printf("cptr+20=%p\n", cptr+20);
    printf("*(int*)(cptr+20)=%d\n", *(int*)(cptr+20));
    int *mptr = (&p.age);
    Person *pptr = (Person *) ((char *) mptr-20);
    printf("*pptr=%p\n", pptr);
    // int **agePtr;
    // agePtr = (int**) (&p+offset);
    // printf("agePtr=%p\n", agePtr);
    // printf("*(&p+offset)=%d\n", *agePtr);
    // printf("&p+offsetof(Person,age)=%p\n", &p+offset);
    // printf("container_of(&p.age,Person,age)=%p\n", container_of(&p.age, Person, age))
}

```

## 執行結果



```
D:\cp>gcc structptr.c -o structptr
```

```
D:\cp>structptr
```

```
offset=0x14
```

```
offsetof(Person, age)=0x14
```

```
&p=0022FF50
```

```
&p.age=0022FF64
```

```
p.age=40
```

```
&p+1=0022FF68
```

```
&((Person*)&p)[1]=0022FF68
```

```
cptr+1=0022FF51
```

```
cptr+20=0022FF64
```

```
*(int*)(cptr+20)=40
```

```
*pptr=0022FF50
```

## 範例程式：鏈結串列

### 檔案：LinkedList.c

```
#include <stdio.h>

typedef struct lnode {
    struct lnode *next;
} ListNode;

void ListNodePrint(ListNode *node) {
    ListNode *p;
    for (p = node; p != NULL; p=p->next)
        printf("%p-->", p);
}

int main() {
    ListNode node1, node2, node3;
    node1.next = &node2;
    node2.next = &node3;
    node3.next = NULL;
    ListNodePrint(&node1);
}
```

### 執行結果

```
D:\cp>gcc LinkedList.c -o LinkedList

D:\cp>LinkedList
0022FF74-->0022FF70-->0022FF6C-->
```

### 參考文獻

- 深入分析 Linux 内核链表 --  
<http://www.ibm.com/developerworks/cn/linux/kernel/l-chain/index.html>



## 鏈結串列：內含物件版 — 通用性鏈結串列，一般性的寫法。

### 程式範例：鏈結串列 -- 內含物件版

```
#include <stdio.h>

typedef struct lnode {
    struct lnode *next;
    void *obj;
} ListNode;

typedef struct {
    ListNode *head;
} List;

void ListNew(List *list) {
    list->head = NULL;
}

void ListAdd(List *list, ListNode *node) {
    if (node == NULL) return;
    node->next = list->head;
    list->head = node;
}

typedef void(*F1)(void*);

void ListPrint(List *list, F1 f) {
    ListNode *p;
    for (p = list->head; p != NULL; p=p->next)
        f(p->obj);
}

typedef struct {
    char *name;
    int age;
```

```
} Person;

void PersonPrint(Person *p) {
    printf("%s is %d years old\n", p->name, p->age);
}

int main() {
    Person john = {
        .name = "John",
        .age = 40
    };
    Person mary = {
        .name = "Mary",
        .age = 32
    };
    Person george = {
        .name = "George",
        .age = 26
    };
    ListNode jnode = { .obj=&john, .next=NULL };
    ListNode mnode = { .obj=&mary, .next=NULL };
    ListNode gnode = { .obj=&george, .next=NULL };
    List list;
    ListNew(&list);
    ListAdd(&list, &jnode);
    ListAdd(&list, &mnode);
    ListAdd(&list, &gnode);
    ListPrint(&list, (F1) PersonPrint);
    return 0;
}
```

## 執行結果

```
D:\cp>gcc LinkedListObj.c -o LinkedListObj
```

```
D:\cp>LinkedListObj
George is 26 years old
Mary is 32 years old
John is 40 years old
```



## 鏈結串列：外包物件版 — 嵌入式的鏈結串列，是模仿 **Linux** 核心的作法。

本範例的實作靈感來自 Linux 核心中的雙向鏈結串列，為了簡單起見，我們改為單向鏈結串列，以便讓讀者容易理解。

### 程式範例：

檔案：**LinkedListEmbed.c**

```
#include <stdio.h>

#define offsetof(type, member) ((size_t) &((type *)0)->member)
#define ListEntry(ptr, type, member) ((type *)((char *)(ptr)-(unsigned
#define ListNew(head) ((head)->next=NULL)
#define ListAdd(head, node) { (node)->next=(head)->next; (head)->ne
#define ListEach(head, pos) for (pos = (head)->next; pos != NULL; p

typedef struct listnode {
    struct listnode *next;
} ListNode;

typedef struct {
    char *name;
    int age;
    ListNode node;
} Person;

void PersonListPrint(ListNode *head) {
    ListNode *ptr;
    ListEach(head, ptr) {
        Person *person = ListEntry(ptr, Person, node);
        printf("%s is %d years old\n", person->name, person->age);
    }
}
```

// 請注意，在本程式中，ListEach 會忽略表頭節點，因此 head 不應該包在 Person

```
int main() {
    ListNode head;
    Person john = {
        .name = "John",
        .age = 40,
    };
    Person mary = {
        .name = "Mary",
        .age = 32,
    };
    Person george = {
        .name = "George",
        .age = 26,
    };
    ListNew(&head);
    ListAdd(&head, &john.node);
    ListAdd(&head, &mary.node);
    ListAdd(&head, &george.node);
    PersonListPrint(&head);
    return 0;
}
```

## 執行結果

```
D:\cp>gcc LinkedListEmbed.c -o LinkedListEmbed
```

```
D:\cp>LinkedListEmbed
George is 26 years old
Mary is 32 years old
John is 40 years old
```

- 來自 **jserv** 的建議

=> 內文沒提到將資料搬出 list 結構的優勢，建議提供 for\_each 的使用案例：



<http://stackoverflow.com/questions/15754236/how-do-i-use-the-list-for-each-macro-in-list-h-from-the-linux-kernel-properly> 對於《將資料搬出 list 結構的優勢》這個問題，除了巨集展開不需要進函數，速度可能會比較快之外，我也想不出其他的優勢了，所以就請大家自己想想，或者問 jserv。

## 物件導向

主題	說明
物件導向的基本概念 -- 封裝，繼承，多型	(Object-Oriented) 物件導向的基本概念
封裝 — (Encapsulation)	使用 C 的結構實作封裝，將資料與函數封裝成物件
繼承 — (Inheritance)	實作繼承，讓子類別具有父類別的欄位
多型 — (Polymorphism)	實作多型，讓同一個指令可以執行不同物件中的函數
類別結構 — (Define Class)	將類別結構獨立出來，以便節省記憶體

## 物件導向的基本概念 -- 封裝，繼承，多型

C 語言雖然不是一種物件導向的語言，但是由於具有函數指標 (function pointer) 與結構 (struct)，因此可以讓我們模擬出類似物件導向的語法。在本章中，我們將說明如何用 C 語言設計物件導向的程式。

物件導向語言大致上具有三個主要的特徵 -- 「封裝、繼承與多型」，以下是這三種特徵的基本描述與範例。

封裝：將資料與函數放在一種稱為物件的結構中。

繼承：子物件可以繼承父物件的所有欄位與屬性，並且可以新增欄位或修改函數。

多型：多種不同的子物件繼承同一種上層物件時，我們可以用上層物件容納之，在呼叫時仍然會根據真實物件型態呼叫對應的子物件函數。

物件導向的三種基本特徵

封裝：

```
class Shape {  
    double area() { return 0.0; }  
}
```

繼承：

```
class Circle extends Shape {  
    public double r;  
    Circle(double pr) { r = pr; }  
    double area() { return 3.14*r*r; }  
}
```

多型：

```
Shape s[] = { new Shape(), new Circle(3.0) };  
for (int i=0; i<s.length; i++)  
    System.out.println("area()="+s[i].area());
```

## 完整程式範例

```
class Shape {
    double area() { return 0.0; }

    public static void main(String[] argv) {
        Shape s[] = { new Shape(), new Circle(3.0) };
        for (int i=0; i<s.length; i++)
            System.out.println("area()="+s[i].area());
    }
}

class Circle extends Shape {
    public double r;
    Circle(double pr) { r = pr; }
    double area() { return 3.14*r*r; }
}
```

### 執行結果

```
D:\cp>javac Shape.java

D:\cp>java Shape
area()=0.0
area()=28.259999999999998
```

在本文中，我們介紹了如何實作封裝、繼承、多型等三種物件導向的基本特性，在本章的後續小節中，我們將同樣以 Shape 這個範例，說明如何用 C 語言實作出這些物件導向功能。

## 來自 jserv 的建議

""" C 語言雖然不是一種物件導向的語言，但是由於具有函數指標 (function pointer) 與結構 (struct)，因此可以讓我們模擬出類似物件導向的語法。在本章中，我們將說明如何用 C 語言設計物件導向的程式。 """

=> 說法不精確，object-oriented programming (OOP) [1] 是種 programming paradigm，用淺顯的話語，就是「物件導向是一種程式開發的態度」，請不要把 OOP 和 OOPL 混淆了，後者是程式語言層面提供 OO 思維。我建議改為以下: """ C 語言一開始並非針對物件導向程式開發而設計的程式語言，但我們可藉由函式指標和結構體，將物件導向落實在 C 程式中。 """ 注意: C 語言規格書出現 "object" 字樣近八百次！

另外，內文還提到: """ 物件導向語言大致上具有三個主要的特徵 — 「封裝、繼承與多型」，以下是這三種特徵的基本描述與範例。 """ => 這個說法不正確，不該把「落實物件導向的機制」當作物件導向，這樣因果錯位實在不好。OOP 的落實有兩種方向：(a) object-based: 如 Java, C++ (b) prototype-based: 如 JavaScript

後者在 C 語言的落實機制可見拙作:

<http://blog.linux.org.tw/~jserv/archives/002057.html>

[1] OOP: [https://en.wikipedia.org/wiki/Object-oriented\\_programming](https://en.wikipedia.org/wiki/Object-oriented_programming)

## 以 C 語言實作物件封裝 (Encapsulation)

物件導向中的封裝，是指將資料與函數封裝在一個稱為物件的結構當中，我們可以使用 C 語言的結構，將資料與函數指標一起放入結構中，就形成了一個類似物件的結構。

### 程式範例：實作 **Circle** 物件

```
#include <stdio.h>

struct Circle {
    void (*new)(struct Circle*, float);
    float (*area)(struct Circle*);
    float r;
};

float CircleArea(struct Circle *obj) { return 3.14 * obj->r * obj->r; }

void CircleNew(struct Circle *obj, float r) {
    obj->new = CircleNew;
    obj->area = CircleArea;
    obj->r = r;
}

int main() {
    struct Circle c;
    CircleNew(&c, 3.0);
    printf("area() = %G\n", c.area(&c));
}
```

執行結果

```
D:\cp>gcc Circle.c -o Circle
```

```
D:\cp>Circle
```

```
area() = 28.26
```

## 以 C 語言實作繼承

在物件導向當中，子類別繼承父類別時，會同時繼承所有父類別的內容，我們可以在程式中加入

程式範例：以 C 語言實作繼承 (**Circle** 繼承 **Shape**)



```
#include <stdio.h>

#define ShapeMembers(TYPE) void (*new)(struct TYPE*);float (*area)(TYPE)

typedef struct _Shape { // Shape 物件，沒有欄位
    ShapeMembers(_Shape);
} Shape;

float ShapeArea(Shape *obj) { return 0; }

void ShapeNew(Shape *obj) {
    obj->new = ShapeNew;
    obj->area = ShapeArea;
}

typedef struct _Circle {
    ShapeMembers(_Circle);
    float r;
} Circle;

float CircleArea(Circle *obj) { return 3.14 * obj->r * obj->r; }

void CircleNew(Circle *obj) {
    obj->new = CircleNew;
    obj->area = CircleArea;
}

int main() {
    Shape s;
    ShapeNew(&s);
    printf("s.area()=%G\n", s.area(&s));

    Circle c;
    CircleNew(&c);
    c.r = 3.0;
    printf("c.area()=%G\n", c.area(&c));
}
```

## 執行結果

```
D:\cp>gcc inherit.c -o inherit
```

```
D:\cp>inherit
```

```
s.area()=0
```

```
c.area()=28.26
```

# 以 C 實作物件導向的多型範例

## 程式範例

```
#include <stdio.h>

#define ShapeText(TYPE) void (*new)(struct TYPE*);float (*area)(struct TYPE*)

typedef struct _Shape { // Shape 物件，沒有欄位
    ShapeText(_Shape);
} Shape;

float ShapeArea(Shape *obj) { return 0; }

void ShapeNew(Shape *obj) {
    obj->new = ShapeNew;
    obj->area = ShapeArea;
}

typedef struct _Circle {
    ShapeText(_Circle);
    float r;
} Circle;

float CircleArea(Circle *obj) { return 3.14 * obj->r * obj->r; }

void CircleNew(Circle *obj) {
    obj->new = CircleNew;
    obj->area = CircleArea;
}

int main() {
    Shape s;
    ShapeNew(&s);
    Circle c;
    CircleNew(&c);
}
```

```
c.r = 3.0;
Shape *list[] = { &s, (Shape*) &c };
int i;
for (i=0; i<2; i++) {
    Shape *o = list[i];
    printf("s.area()=%G\n", o->area(o));
}
}
```

### 執行結果

```
D:\cp>gcc poly.c -o poly
```

```
D:\cp>poly
```

```
s.area()=0
```

```
s.area()=28.26
```

## C 語言實作獨立的類別

在前述的範例當中，我們直接將資料與函數封裝在結構當中，以形成物件，這種實作方式並沒有為類別定義獨立的結構，於是每個物件當中都會有一份所有成員函數的指標，當物件的數量很多時，這可能會浪費不少記憶體。

前述的這種實作方式，比較像是一種變形後的物件導向實作法，這種方法稱為原型 (Prototype) 導向的實作法，像是 JavaScript 就採用了類似的實作方式。如果我們要實作出像 Java 或 C# 一樣的物件導向作法，應該將類別的結構獨立出來，這樣會比較能夠規模化，而且通常可以節省記憶體。

在以下的程式中，我們將再度用 C 語言實作出這種方式，將物件與類別獨立成兩個不同結構。如此，不管我們建立幾份物件，類別物件永遠都只會有一個，請看下列程式碼。

程式實作：將類別獨立出來

檔案：**polyClass.c**

```
#include <stdio.h>

#define ShapeClassMembers(OBJ) float (*area)(struct OBJ*)
#define ShapeMembers(OBJ)

struct _Shape;

typedef struct _ShapeClass { // Shape 物件，沒有欄位
    ShapeClassMembers(_Shape);
} ShapeClass;

typedef struct _Shape {
    ShapeClass *class;
    ShapeMembers(_Shape);
} Shape;

float ShapeArea(Shape *obj) { return 0; }

ShapeClass shapeClass = { .area = ShapeArea };
```

```

struct _Circle;

typedef struct _CircleClass {
    ShapeClassMembers(_Circle);
    float r;
} CircleClass;

typedef struct _Circle {
    CircleClass *class;
    ShapeMembers(_Circle);
    float r;
} Circle;

float CircleArea(Circle *obj) { return 3.14 * obj->r * obj->r; }

CircleClass circleClass = { .area = CircleArea };

int main() {
    Shape s = { .class = &shapeClass };
    Circle c = { .class = &circleClass };
    c.r = 3.0;
    Shape *list[] = { &s, (Shape*) &c };
    int i;
    for (i=0; i<2; i++) {
        Shape *o = list[i];
        printf("s.area()=%G\n", o->class->area(o));
    }
}

```

## 執行結果

```

D:\cp>gcc polyClass.c -o polyClass

D:\cp>polyClass
s.area()=0
s.area()=28.26

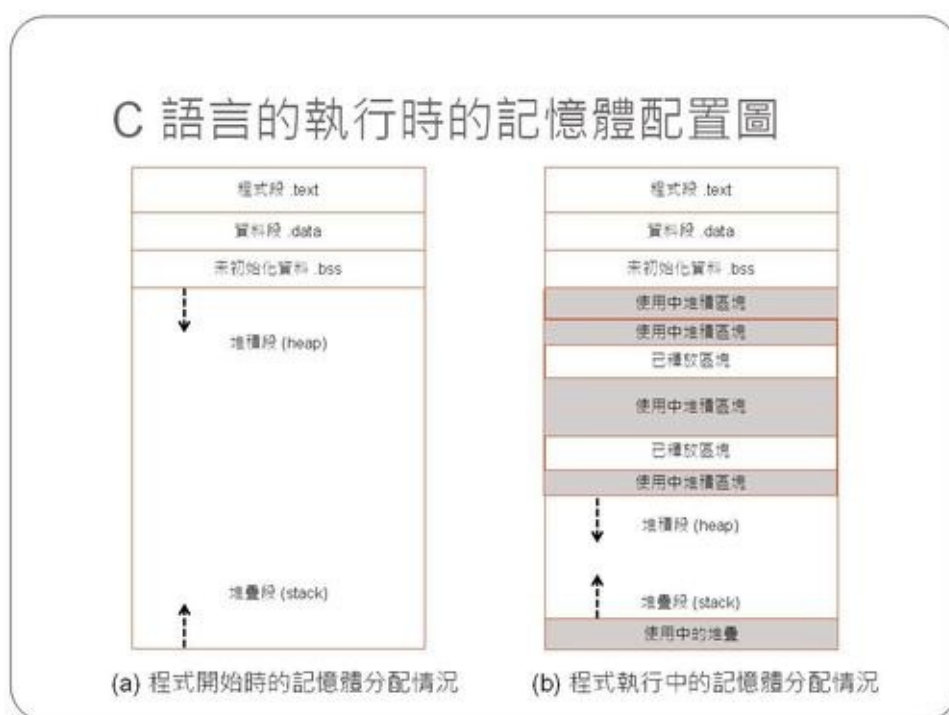
```

## 記憶體管理

主題	說明
C 的執行環境	text, data, bss, stack, heap
記憶體漏洞	(memory check)

## C 語言的執行環境

程式段 (.text) 主要存放程式的機器碼，資料段 (.data) 則是存放全域變數的資料，BSS 段 (.bss) 存放的是未初始化的全域變數，堆積段 (.heap) 則是在程式使用 malloc 進行記憶體分配時，可以分配的動態記憶體空間，而堆疊段 (.stack) 則存放「參數、函數返回點、區域變數、框架指標」等資料。圖一顯示了 C 語言的執行環境，左半部的 (a) 是這五個區段的初始狀況，而右半部的 (b) 則是在程式執行中，堆疊與堆積已經進行某些分配後的狀況。



圖一、C 語言的執行環境

請讀者將焦點先放在堆疊與堆積這兩段上，C 語言中的「參數、函數返回點、區域變數、框架指標」等資料，被儲存在堆疊段中，這個區段會隨著函數呼叫的層次數目而增長或縮短。如果這個區段成長過頭，導致堆疊段覆蓋到堆積段的空間時，就被稱為堆疊溢位 (Stack Overflow)，這種情況通常是因為程式進行遞回呼叫，卻又沒有正確判斷終止條件，導致遞迴層數過多所產生的錯誤情況。

讓我們將焦點轉到堆積段中，假如我們用 malloc() 函數分配記憶體空間，則 malloc() 函數會從堆積段 (heap) 中找到一個夠大的區塊，分配給 malloc() 函數傳回。然後，當我們使用 free() 函數釋放記憶體空間時，則原先分配的區塊會歸還給堆積系統，此時通常會在堆積的記憶空間中留下一個空洞。在程式的執行過程當



中，`malloc()` 與 `free()` 會交錯執行，因而導致整個堆積區塊開始產生許多空洞，這將會造成記憶體管理的負擔，假如堆積系統無法找到足夠大的堆積區塊時，就會造成記憶體分配失敗的情況，因而導致程式無法繼續執行。

對於現今的電腦而言，由於記憶體的容量龐大，而且通常有分頁機制可以幫助作業系統進行記憶體管理，因此堆積分配失敗的情況較為少見，但是對於早期的電腦，或者是嵌入式系統而言，堆積的分配就是一個相當難以處理的問題。

在 C 語言的函式庫設計上，通常會盡量避免使用 `malloc()` 等函數分配堆積空間，因為這会造成記憶體管理的困擾，也會讓程式的執行效率難以預料。因此，您可以看到 C 語言的字串函數，通常會盡可能避免使用 `malloc()` 函數分配空間。

### 一個誤用的 C 語言字串範例

在 Java 這樣的語言當中，字串的長度是可以改變的，您可能會使用下列程式，很自由的讓字串大小增大，Java 會自動的幫您分配字串所需要的記憶空間，而不會產生太大的問題。

### 範例一、字串連接的 Java 程式

```
String s = "";  
for (i=0; i<100; i++)  
    s = s + token[i]
```

但是在 C 語言當中，您就會遇到一個困擾，假如我要撰寫一個類似的程式，那麼字串 `s` 的長度應該要設定為多長呢？請看下列範例。

### 範例二、字串連接的 C 程式

```
char s[1000];  
for (i=0; i<100; i++)  
    strcat(s, token[i]);
```

您可能會懷疑，長度 1000 到底夠不夠呢？假如 `token` 陣列中的字串長度平均超過 10 個字，那麼 `s` 的長度 1000 就會不夠了。這樣看來，Java 的字串函式庫設計似乎比 C 語言要好得多了，不是嗎？

如果您不夠理解 C 語言，就很可能會寫出像範例二這樣的程式，但是這對 C 語言來說其實是一種誤用，誤用的原因是：「想要用靜態的字串處理動態的問題」。

假如您真的必需撰寫像範例二這樣的程式，那麼應該自行設計一個動態的字串函式庫，或者採用某個現成的動態字串函式庫，而不是直接用 C 語言內建的標準函數。但是 C 語言的初學者往往沒有這樣一個函式庫，因而寫出像範例二這樣的程式。

程度稍好的資訊系學生，可能理解到這個問題不能採用靜態的解決方式，因此使用 `malloc()` 函數進行記憶體分配，以解決這個令人困擾的問題，於是就寫出了下列的程式碼。

#### 範例三、字串連接的 C 程式 (malloc 版)

```
char *s = malloc(1);
s[0] = '\0';
for (i=0; i<100; i++) {
    char *t = malloc(strlen(s)+strlen(token[i])+1);
    sprintf(t, "%s%s", s, token[i]);
    free(s);
    s = t;
}
```

但是這樣撰寫 C 語言，仍然是初學者會犯的錯誤之一，這種用法完全誤用了 C 語言的能力，造成了很多次的 `malloc()` 分配，卻又很沒效率的處理字串長度的成長問題。

歸根究底，這個問題是由於 C 語言沒有提供一個標準的動態字串而造成的，如果您真的需要一個這樣的程式，那麼就應該採用一個支援動態字串的函式庫，然後將程式改寫如下。

#### 範例四、字串連接的 C 程式 (動態字串版)

```
Str *s = StrNew();
for (i=0; i<100; i++) {
    StrAppend(s, token[i]);
}
```

這樣範例四的 C 語言程式，其實就與範例一的 Java 程式，看來相差不大了，最大的差別是 C 語言沒有支援物件的概念而已。

動態字串

要能撰寫像範例四這樣的一個程式，動態字串函式庫至少要能支援 StrNew() 與 StrAppend() 這兩個函數，那麼我們應該怎麼做呢？其實，要自己打造這樣一個程式相當容易，筆者可以馬上撰寫一個，如範例五所示。

範例五：實作動態字串函式庫

```
typedef struct Str {
    int len, size;
    char *s;
};

Str *StrNew();
void StrAppend(Str *str, char *s);

Str *StrNew() {
    Str *str = malloc(sizeof(Str));
    str->s = malloc(1);
    str->s[0] = '\0';
    str->len = 0;
    str->size = 1;
}

void StrAppend(Str *str, char *s) {
    int newLen = str->len + strlen(s);
    if (newLen+1 > str->size) {
        int newSize = max(str->size*2, newLen+1);
        char *t = malloc(newSize);
        sprintf(t, "%s%s", str->s, s);
        free(str->s);
        str->s = t;
        str->len = newLen;
        str->size = newSize;
    } else {
        strcat(&str->s[str->len], s);
        str->len += strlen(s);
    }
}
```

只要有了這樣一個函式庫，那麼我們就不需要為了 C 語言缺乏動態字串而困擾了，也就不需要每次都寫出像範例二或範例三這樣難看且沒有效率的程式了，而是直接寫出像範例四這樣乾淨，簡潔的函式庫了。

### C 語言標準字串函式庫的設計理念

既然如此，那麼為何 C 語言要設計出像 `strcpy()`, `strcat()`, `strtok()` 這樣的字串函式庫，而不直接使用像上述的動態字串函式庫取代就好了呢？關於這個問題，我們必須回到當初 K & R 兩人設計 C 語言的初始環境，才能看出其原因。

K & R 設計 C 語言時，還沒有物件導向的概念，因此不太可能設計出像範例五這樣具有物件導向概念的字串函式庫。當時電腦的記憶體極為有限，而且 K&R 一心只想設計出 UNIX 作業系統，因此像 `strcpy()`、`strcat()`、`strtok()` 這樣的函數，可以同時支援字串陣列與指標，因而發展出這樣一套 C 語言函式庫。

來自 **jserv** 的建議

"" ""

K & R 設計 C 語言時，還沒有物件導向的概念，因此不太可能設計出像範例五這樣具有物件導向概念的字串函式庫。當時電腦的記憶體極為有限，而且 K&R 一心只想設計出 UNIX 作業系統，因此像 `strcpy()`、`strcat()`、`strtok()` 這樣的函數，可以同時支援字串陣列與指標，因而發展出這樣一套 C 語言函式庫。

"" ""

==> 這說法不正確。事實上，我們在 UNIX 在 1974 年的經典論文，不時可見 "object-oriented" OOP 風格。合理的說法是，C 語言的設計者將記憶體管理交給程式開發者去負責，UNIX 和 C 語言一樣的原則是，充分相信程式開發者，特別在記憶體管理的議題上。

## C 語言的記憶體漏洞檢查

在 C 語言中，如果有人用 `malloc()` 等函數分配了記憶體，卻忘了用 `free()` 等函數進行釋放，那就會產生記憶體漏洞。要解決這個問題，必須遵循幾個原則，第一個是程式紀律的問題，例如一個很好的習慣是，採用物件導向的寫法，然後在物件的建構函數中分配記憶體，並在解構函數中，釋放該物件所分配的所有記憶體。

第二個原則是程式測試的問題，您可以使用記憶體檢查函數，進行記憶體漏洞檢查，像是 Linux 當中就有 `mtrace` 這樣的套件可以使用，您只要引用 這個標頭檔即可，以下是一個使用 `mtrace` 進行記憶體檢查的範例。

### `mtrace` 的使用 (Linux)

檔案：`leak.c`

```
#include <stdlib.h>
#include <mcheck.h> // mtrace 的標頭檔

int main() {
    int *a;
    mtrace(); // 啟用 mtrace
    a = malloc(sizeof(int)); 分配記憶體
    *a = 7;
    // 忘了釋放
    return EXIT_SUCCESS;
}
```

執行方法：

```

setenv MALLOC_TRACE /home/karthik/temp/trace.txt // 設定 mtrace
$ gcc -g -Wall -ansi -pedantic leak.c -o leak.o // 編譯
$ ./leak // 執行
$ mtrace leak.o /home/karthik/temp/trace.txt // 追蹤記憶體漏洞
Memory not freed:
-----
      Address      Size      Caller
0x08049910         0x4  at /home/karthik/tips/leak.c:9 // 發現在 leak.c:9

```

### 簡單的檢查方法

假如您沒有辦法使用像 mtrace 這種由系統所提供的記憶體檢查方法，也可以自己製作一個很簡單的版本。為了示範這種作法，我們設計了 new(), del() 與 mcheck() 等三個巨集，以示範這種簡單的漏洞檢查法。

程式範例：mcheck.c

```

#include <stdio.h>
#include <stdlib.h>

int newSize = 0;
int delSize = 0;
#define new(TYPE, n) malloc(n*sizeof(TYPE)); newSize+=n*sizeof(TYPE)
#define del(ptr, TYPE, n) free(ptr); delSize+=n*sizeof(TYPE)
#define mcheck() printf("Memory:newSize=%d delSize=%d leakSize=%d\n",
                        newSize, delSize, newSize-delSize);

int main() {
    int *ip = new(int, 10);
    char *cp = new(char, 5);
    del(ip, int, 10);
    mcheck();
}

```

執行結果：

```
D:\cp>gcc mcheck.c -o mcheck
```

```
D:\cp>mcheck
```

```
Memory:newSize=45 delSize=40 leakSize=5
```

有了這樣的函數，您就可以知道是否有記憶體漏洞的存在了，雖然不像 mtrace 那樣可以直接告訴您產生漏洞的程式位置，但至少可以讓您檢查是否存在記憶體漏洞。

來自 **jserv** 的建議

""

K & R 設計 C 語言時，還沒有物件導向的概念，因此不太可能設計出像範例五這樣具有物件導向的 C 語言。K&R 一心只想設計出 UNIX 作業系統，因此像 strcpy()、strcat()、strtok() 這樣的函數，可以同時支援字串陣列與指標，因而發展出這樣一套 C 語言函式庫。

""

==> 這說法不正確。事實上，我們在 UNIX 在 1974 年的經典論文，不時可見 "object-oriented" 風格。合理的說法是，C 語言的設計者將記憶體管理交給程式開發者去負責，UNIX 和 C 語言一樣的原則是，充分相信程式開發者，特別在記憶體管理的議題上。

參考文獻

Identifying Memory Leaks in Linux for C++ Programs -- <http://www.dedecms.com>

## 檔案系統

主題	說明
檔案緩衝區	如何指定緩衝區大小與位址
臨時暫存檔	如何建立暫存檔案
檔案錯誤	如何處理檔案的錯誤狀況
目錄管理	建立目錄、切換路徑、取得目前路徑



## 檔案緩衝區

檔案緩衝區 -- 如何指定緩衝區大小與位址

在 C 語言的標準輸出入函式庫中，您可以使用 `setbuf(file, buffer)` 的方法，設定檔案的緩衝區，如果您用 `setbuf(file, NULL)` 這個函數將緩衝區設為 `NULL`，就會取消檔案緩衝機制，每次都直接輸出到檔案中。

另外，您也可以使用 `setvbuf(file, buffer, mode, size)` 這樣的方式，設定檔案緩衝區與緩衝模式，其中的 `mode` 有三種可能，第一種 `_IOFBF` 代表完全緩衝，該模式會等緩衝區滿了之後再輸出，第二種 `_IOLBF` 代表行緩衝 (line buffered)，一但碰到換行時就會輸出，而第三種 `_IONBF` 代表不緩衝，該方法會完全不進行緩衝而直接輸出。

程式範例

檔案：setbuf.c

```
#include <stdio.h>

int main () {
    char buffer[BUFSIZ]; // BUFSIZ 定義在 stdio.h 當中。
    FILE *file;
    file=fopen ("test.txt","w");
    setbuf (file, buffer ); // setbuf(file, NULL) 會取消緩衝區，每次
    fclose (file);
    return 0;
}
```

執行結果

```
D:\cp>gcc setbuf.c -o setbuf
```

```
D:\cp>setbuf
```

來自 **jserv** 的建議

這整章節沒有把 C 語言的溫床 – UNIX 背後的思維闡述好，希望能多談 standard I/

## 臨時暫存檔 -- 如何建立暫存檔案

### 程式範例

```
#include <stdio.h>

int main(void){
    FILE *file;
    if((file=tmpfile())==NULL) {
        printf("Cannot open temporary work file.\n");
        exit(1);
    }
    fprintf(file, "Hello! How are you.");
    fflush(file);

    char msg[10];
    fseek(file, 0, SEEK_SET);
    fscanf(file, "%s", msg);
    printf("msg=%s", msg);
    fclose(file);
}
```

### 執行結果

```
D:\cp>gcc tmpfile.c -o tmpfile
```

```
D:\cp>tmpfile
msg=Hello!
```

## 檔案錯誤 -- 如何處理檔案的錯誤狀況

### 程式範例

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *in, *out;
    char ch;

    if((in=fopen(argv[1], "rb")) == NULL) {
        perror("Error");
        exit(1);
    }

    while(!feof(in)) {
        ch = getc(in);
        if(ferror(in)) {
            perror("Error");
            clearerr(in);
            break;
        }
        putchar(ch);
    }
    fclose(in);
    return 0;
}
```

### 執行結果

```
D:\cp>gcc ferror.c -o ferror
```

```
D:\cp>ferror
```

```
Error: No such file or directory
```

```
D:\cp>ferror exist.not
```

```
Error: No such file or directory
```

```
D:\cp>ferror ferror.c
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(int argc, char *argv[]) {
```

```
    FILE *in, *out;
```

```
    char ch;
```

```
    if((in=fopen(argv[1], "rb")) == NULL) {
```

```
        perror("Error");
```

```
        exit(1);
```

```
    }
```

```
    while(!feof(in)) {
```

```
        ch = getc(in);
```

```
        if(ferror(in)) {
```

```
            perror("Error");
```

```
            clearerr(in);
```

```
            break;
```

```
        }
```

```
        putchar(ch);
```

```
    }
```

```
    fclose(in);
```

```
    return 0;
```

```
}
```

## C語言 -- 目錄管理函式庫

### 程式範例

```
#include <stdio.h>
#include <sys/stat.h>

int main() {
    char path[100];
    size_t size;
    getcwd(path, sizeof(path));
    printf("path=%s\n", path);
    mkdir("/temp1", S_IWRITE);
    chdir("/temp1");
    getcwd(path);
    printf("path=%s\n", path);
}
```

### 執行結果

```
D:\ccc\ca>gcc dirapi.c -o dirapi
```

```
D:\ccc\ca>dirapi
path=D:\ccc\ca
path=D:\temp1
```

## 錯誤處理

主題	說明
錯誤代號與 訊息	(errno, strerror, perror) 與錯誤代號相關的訊息輸出方法
錯誤訊息列表	用 strerror 列出所有內建的錯誤訊息
檔案錯誤	(ferror,clearerr) 檔案讀取寫入錯誤，清除錯誤，繼續執行下去
短程跳躍	(goto) 函數內的跳躍，不可跨越函數
長程跳躍	(setjump 與 longjump) 在錯誤發生時，儲存行程狀態，執行特定程式的方法
訊號機制	(signal) 攔截中斷訊號的處理機制
模擬 try ... catch	使用跳躍機制 (setjump, longjump) 模擬 try ... catch 的錯誤捕捉機制

## 錯誤代號與訊息 -- (errno, strerror, perror) 與 錯誤代號相關的訊息輸出方法

### 程式範例

```
#include <stdio.h>
#include <errno.h>

int main ()
{
    FILE * file;
    file=fopen ("exist.not","rb");
    if (file==NULL) {
        perror("perror");
        printf("strerror(errno)=%s\n", strerror(errno));
    }
    else
        fclose (file);
    return 0;
}
```

### 執行結果

```
D:\cp>gcc perror.c -o perror

D:\cp>perror
perror: No such file or directory
strerror(errno)=No such file or directory
```



## 錯誤訊息列表 -- 用 **strerror** 列出所有內建的錯誤訊息

### 程式範例

```
#include <stdio.h>
#include <errno.h>

int main ()
{
    FILE * file;
    file=fopen ("exist.not","rb");
    if (file==NULL) {
        perror("perror");
        printf("strerror(errno)=%s\n", strerror(errno));
    }
    else
        fclose (file);
    return 0;
}
```

### 執行結果

```
D:\cp>gcc perror.c -o perror

D:\cp>perror
perror: No such file or directory
strerror(errno)=No such file or directory
```

## C 語言 -- 檔案錯誤 **ferror()**

### 程式範例

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *in, *out;
    char ch;

    if((in=fopen(argv[1], "rb")) == NULL) {
        perror("Error");
        exit(1);
    }

    while(!feof(in)) {
        ch = getc(in);
        if(ferror(in)) {
            perror("Error");
            clearerr(in);
            break;
        }
        putchar(ch);
    }
    fclose(in);
    return 0;
}
```

### 執行結果

```
D:\cp>gcc ferror.c -o ferror
```

```
D:\cp>ferror
```

```
Error: No such file or directory
```

```
D:\cp>ferror exist.not
```

```
Error: No such file or directory
```

```
D:\cp>ferror ferror.c
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(int argc, char *argv[]) {
    FILE *in, *out;
    char ch;

    if((in=fopen(argv[1], "rb")) == NULL) {
        perror("Error");
        exit(1);
    }

    while(!feof(in)) {
        ch = getc(in);
        if(ferror(in)) {
            perror("Error");
            clearerr(in);
            break;
        }
        putchar(ch);
    }
    fclose(in);
    return 0;
}
```

## C 語言 -- 短程跳躍 (goto)

### 程式範例

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    char *fileName = argv[3];
    if (b == 0) // can't divide by 0
        goto DivideByZero;
    int result = a/b;
    FILE *file;
    if ((file=fopen(fileName, "w")) == NULL)
        goto FileError;
    else {
        fprintf(file, "%d/%d=%d\n", a, b, result);
        printf("save to file %s : %d/%d=%d\n", fileName, a, b, result);
    }
    fclose(file);
    goto Exit;
DivideByZero:
    printf("Error : Divide by zero\n");
    goto Exit;
FileError:
    printf("Error : File error\n");
    goto Exit;
Exit:
    return 0;
}
```

### 執行結果

```
D:\cp>gcc trygoto.c -o trygoto
```

```
D:\cp>trygoto 7 2 div.txt  
save to file div.txt : 7/2=3
```

```
D:\cp>trygoto 7 0 div.txt  
Error : Divide by zero
```

```
D:\cp>trygoto 7 2 trygoto.exe  
Error : File error
```

### 注意事項

goto 指令之所以被認為是短程跳躍，是因為 goto 不可以跨越函數，舉例而言，以下的跳躍方式就會出錯。

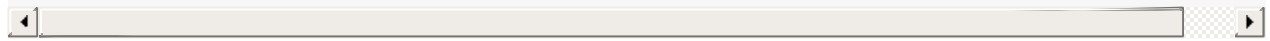
```
#include <stdio.h>

void div() {
    DivideByZero:
    printf("Error : Divide by zero\n");
}

int main(int argc, char *argv[]) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    char *fileName = argv[3];
    if (b == 0) // can't divide by 0
        goto DivideByZero;
    int result = a/b;
    return 0;
}
```

### 編譯錯誤

```
D:\cp>gcc trygotoerror.c -o trygotoerror
trygotoerror.c: In function `main':
trygotoerror.c:13: error: label `DivideByZero' used but not defined
```



## 長程跳躍 -- (**setjump** 與 **longjump**) 在錯誤發生時，儲存行程狀態，執行特定程式的方法

程式範例：**setjump** 與 **longjump**

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf jumper;

int div(int a, int b) {
    if (b == 0) { // can't divide by 0
        longjmp(jumper, -3);
    }
    return a / b;
}

int main(int argc, char *argv[]) {
    int jstatus = setjmp(jumper);
    if (jstatus == 0) {
        int a = atoi(argv[1]);
        int b = atoi(argv[2]);
        printf("%d/%d", a, b);
        int result = div(a, b);
        printf("=%d\n", result);
    }
    else if (jstatus == -3)
        printf(" --> Error:divide by zero\n");
    else
        printf("Unhandled Error Case");
}
```

執行結果

```
D:\cp>gcc jump.c -o jump

D:\cp>jump 7 2
7/2=3

D:\cp>jump 7 0
7/0 --> Error:divide by zero
```

### 來自 jserv 的建議

=> 請提及 C 語言作例外處理的重要性，以及如何用 setjump/longjump 實做 user-  
可參見拙作：<http://blog.linux.org.tw/~jserv/archives/001848.html>

### 參考文獻

Java2s (C / ANSI-C) » setjmp.h » longjmp,

<http://www.java2s.com/Code/C/setjmp.h/longjmplongjump.htm>

Exception Handling in C without C++ -- Clean error handling without overhead, by Tom Schotland and Peter Petersen <http://www.on-time.com/ddj0011.htm>

第16集 C语言中一种更优雅的异常处理机制, 作者：王胜祥 来源：希赛网 <http://www.csai.cn> 2005年5月19日

第17集 全面了解setjmp与longjmp的使用, 作者：王胜祥 来源：希赛网 <http://www.csai.cn> 2005年5月21日



## 訊號機制 -- (signal) 攔截中斷訊號的處理機制

### 程式範例

```
#include <signal.h>
#include <stdio.h>
#include <windows.h>

void sigdump(int sig) {
    printf("catch a signal:%d\n", sig);
}

int main(int argc, void *argv[]) {
    signal(SIGABRT, &sigdump);    // Process abort signal.
    signal(SIGFPE, &sigdump);    // Erroneous arithmetic operat
    signal(SIGILL, &sigdump);    // Illegal instruction.
    signal(SIGINT, &sigdump);    // Terminal interrupt signal.
    signal(SIGSEGV, &sigdump);    // Invalid memory reference.
    signal(SIGTERM, &sigdump);    // Termination signal.
    int a=7, b=0, result;
    if (strcmp(argv[1], "FPE")==0)
        result = a/b;
    else if (strcmp(argv[1], "SEGV")==0) {
        * (int*) (10000) = 1;
    }
    Sleep(1000*10);
    return 0;
}
```

### 執行結果

```
D:\cp>gcc signal.c -o signal
```

```
D:\cp>signal FPE
catch a signal:8
```

```
D:\cp>signal SEGV
catch a signal:11
```

```
D:\cp>signal
catch a signal:11
```

```
D:\cp>signal X      // 執行後請在 10 秒鐘內按下 Ctrl-C, 就會出現 catch a
catch a signal:2
```

來自 **jserv** 的建議

=> 這個案例不好，第一個因為 <windows.h>，另外沒有闡述 UNIX signal 和 Wind

參考文獻

C语言编程技巧-signal(信号), 2008-12-08 来源：网络 --

<http://www.uml.org.cn/c++/200812083.asp>

## 模擬 try ... catch -- 使用跳躍機制 (setjump, longjump) 模擬 try ... catch 的錯誤捕捉機制

程式範例

檔案 : trycatch.c

```
#include <stdio.h>
#include <setjmp.h>

enum Error { NoError=0, DivByZero=1, FileError=2 };

jmp_buf jumper;

void run(char *astr, char *bstr, char *fileName) { // try 的主
    int a = atoi(astr);
    int b = atoi(bstr);
    if (b == 0) // can't divide by 0
        longjmp(jumper, DivByZero);
    int result = a/b;
    FILE *file;
    if ((file=fopen(fileName, "w")) == NULL)
        longjmp(jumper, FileError);
    else {
        fprintf(file, "%d/%d=%d\n", a, b, result);
        printf("save to file %s : %d/%d=%d\n", fileName, a, b, result);
    }
    fclose(file);
}

int main(int argc, char *argv[]) {
    int error = setjmp(jumper); // try
    switch (error) { //
        case NoError: //
            run(argv[1], argv[2], argv[3]); // run();
            break; //
        case DivByZero: // catch DivByZero:
```

```

        printf("Error %d : Divide by zero\n", error);//    ...
        break;    //
    case FileError:    // catch FileError:
        printf("Error %d : File error\n", error);    //    ...
        break;    //
    default:    // default:
        printf("Error %d:Unhandled error\n", error);//    ...
    }
}

```

### 輸出結果

```
D:\cp>gcc trycatch.c -o trycatch
```

```
D:\cp>trycatch 7 2 div.txt
save to file div.txt : 7/2=3
```

```
D:\cp>trycatch 7 0 div.txt
Error 1 : Divide by zero
```

```
D:\cp>trycatch 7 0 trycatch.exe
Error 1 : Divide by zero
```

```
D:\cp>trycatch 7 1 trycatch.exe
Error 2 : File error
```

## 巨集處理

主題	說明
將函數巨集化	使用 <code>inline</code> 讓編譯器得以視情況選擇要用巨集或函數呼叫
引用防護	避免重複引用某一個引用檔，或者重複定義某結構
條件編譯	<code>#if</code> , <code>#else</code> , <code>#endif</code> , <code>#ifdef</code> , <code>#ifndef</code> , ....
編譯時期變數	<code>FILE</code> , <code>LINE</code> , <code>DATE</code> , <code>TIME</code> , <code>STDC</code> .
編譯時期函數	<code>#define</code> , <code>#undef</code> , <code>defined()</code> , <code>#error</code> , <code>#line</code> , ...
編譯指示	<code>#pragma</code> , <code>_Pragma()</code> , GCC dependency, GCC poison, ...
字串化	Stringification, 使用 <code>#symbol</code> 可以將某符號字串化

## C 語言 -- 使用 **Inline** 函數

程式範例

檔案：inline.c

```
inline int max(a,b) {  
    return (a>b?a:b);  
}  
  
inline int min(a,b) {  
    return (a<b?a:b);  
}  
  
int main() {  
    int x = max(3,5);  
    int y = min(3,5);  
    printf("max(3,5)=%d\n", x);  
    printf("min(3,5)=%d\n", y);  
}
```

巨集展開結果

執行 `gcc -E inline.c -o inline.i` 指令之後，就會得到 inline.i

檔案：inline.i

```
inline int max(a,b) {
    return (a>b?a:b);
}

inline int min(a,b) {
    return (a<b?a:b);
}

int main() {
    int x = max(3,5);
    int y = min(3,5);
    printf("max(3,5)=%d\n", x);
    printf("min(3,5)=%d\n", y);
}
```

來自 **jserv** 的建議

原本我寫了這句：「將函數巨集化 — (inline) 使用 inline 可以增快速度，但也會讓程式碼增大」，但似乎有很大問題，所以我就拿掉了，但是我搞不清楚問題在哪？所以忠實呈現 jserv 的來信建議。

"將函數巨集化 - (inline) 使用 inline 可以增快速度，但也會讓程式碼增大。"

=> 這嚴重誤導讀者！

C99 的 inline 行為和 macro 不同，請見：

<http://www.greenend.org.uk/rjk/tech/inline.html>

後來經過 討論 之後，我開始理解 jserv 所說的誤導是指甚麼了，請容我總結一下錯誤的原因。

inline 並非強制要編譯器以巨集方式展開，而是提示編譯器可以用巨集或函數呼叫的方式處理，也就是授權編譯器決定要不要用巨集展開的意思。以下是我和 jserv 的討論的過程

Jim Huang：在 C99，inline function 本質上還是 function，是程式開發者對 C 編譯器的最佳化「提示」，不總是會像 macro 一般透過 C preprocessor 一樣「展開」。可以做個簡單的實驗，gcc -O0 (關閉最佳化) 對照看輸出的組合語言 ('-S' 選項)，inline 沒有實際作用 陳鍾誠：原來如此，也就是《提示編譯器你可以自行決

定要用巨集還是函數呼叫的意思》是嗎？

Jim Huang：改說「提示編譯器在適當的時機，可將函式當作巨集一般展開到函式呼叫之處」，這樣會更清楚

盧鈺辰 我的理解是：inline 只是"建議"編譯器你在這裡可以把整個function貼過來，好減少呼叫函式所需的長程跳躍但是否要這麼做，會由編譯器自行決定所以inline有可能會被編譯器使用，也有可能不會。請問2位大師，我理解的對嗎？如果有錯誤指正，我會很開心

Jim Huang：盧鈺辰: inline function 無法在每個場合都將 function body 予以 inlining，比方說 inline void foo(int n) { char str[n]; ... } 因為用到 variable sized data type，編譯器看到 inline 關鍵字卻無法達到預期效果 (請想想為什麼)



## 引用防護 -- 避免重複引用某一個引用檔，或者重複定義某結構

在開發 C 語言專案時，我們通常在每一個標頭檔的開始與結尾，使用 `#ifndef` `#define` `#endif` 的方式防止重複引用，其語法通常如下所示。

```
#ifndef XXX_H
#define XXX_H
...

#endif
```

舉例而言，以下是我們所撰寫的一個標頭檔 `Str.h`，我們可以使用 `#ifndef STRH` `#define STRH ...#endif` 來避免重複引用 `Str.h` 所造成的編譯錯誤，其程式碼如下所示。

```
#ifndef STR_H
#define STR_H

typedef struct {
    char *s;
} Str;

extern void StrAppend(String*, char *);

#endif
```

這種方式是許多 C 語言初學者所不知道的，由於沒有與其他人一同開發過專案，而且寫程式時也都是自己一個人寫，就很難知道要使用這樣的技巧，這種技巧稱為「引用防護」(include guard)。

在某些 C 語言編譯器中，提供了 `#pragma once` 這樣的編譯指引，可以避免冗長的引用防護撰寫語法，但是為了可攜性的緣故，通常我們還是會加上引用防護，而不是只撰寫 `#pragma once`，因為畢竟寫一行的 `#pragma` 與寫三行的引用防護並沒有差太多。

在 Objective C 這個語言中，由於內建就有引用防護機制，因此就不需要撰寫這樣的語法了。

## 條件編譯 -- #if, #else, #endif, #ifdef, #ifndef, ....

### 條件編譯的語法

```
#if constant_expression
#elif constant_expression
...
#else
#endif
```

### 程式範例

```
#include <stdio.h>

int main() {
#ifdef Linux
    printf("OS=LINUX\n");
#elif defined(Windows)
    printf("OS=Microsoft Windows\n");
#elif defined(OS)
    printf("OS=%s", OS);
#else
    printf("OS=Unknown\n");
#endif
}
```

### 執行結果

```
D:\cp>gcc macroIf.c -o macroIf

D:\cp>macroIf
OS=Unknown

D:\cp>gcc -DWindows macroIf.c -o macroIf

D:\cp>macroIf
OS=Microsoft Windows

D:\cp>gcc -DLinux macroIf.c -o macroIf

D:\cp>macroIf
OS=LINUX

D:\cp>gcc -DOS=\"Sun Solaris\" macroIf.c -o macroIf
gcc: Solaris": Invalid argument
macroIf.c: In function `main':
macroIf.c:9: error: missing terminating " character
macroIf.c:9: error: syntax error before ')' token

D:\cp>gcc -DOS=\"Solaris\" macroIf.c -o macroIf

D:\cp>macroIf
OS=Solaris
D:\cp>
```

## 參考文獻

Wikipedia:C preprocessor --[http://en.wikipedia.org/wiki/C\\_preprocessor](http://en.wikipedia.org/wiki/C_preprocessor)

## 編譯時期變數 -- FILE, LINE, DATE, TIME, STDC.

程式範例

檔案：macroVar.c

```
#include <stdio.h>

int main(void) {
    // #line 100 "renameFromMacroVar.c"
    printf("Compiling %s, line: %d, on %s, at %s, STDC=%d",
           __FILE__, __LINE__, __DATE__, __TIME__, __STDC__);

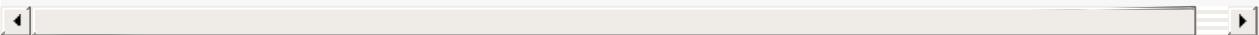
    return 0;
}
```

執行結果

```
D:\cp>gcc macroVar.c -o macroVar
```

```
D:\cp>macroVar
```

```
Compiling macroVar.c, line: 6, on Sep  2 2010, at 14:55:30, STDC=1
```

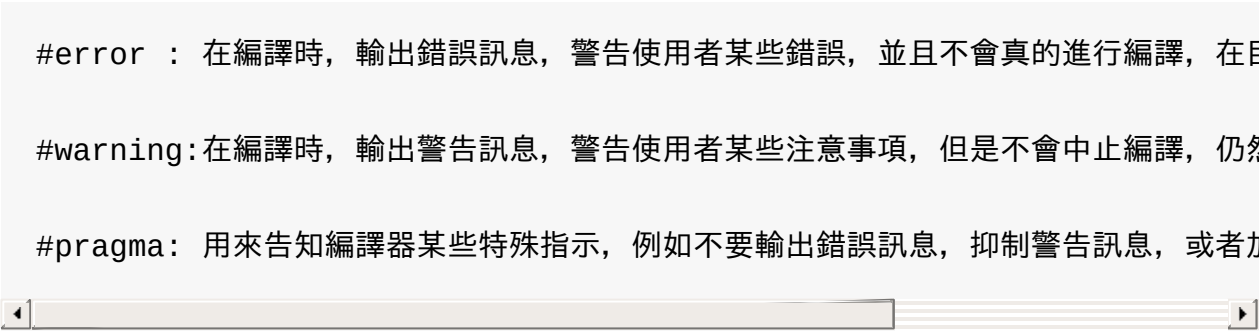


## 編譯時期函數 -- #define, #undef, defined(), #error, #line, ...

在 C 語言的巨集處理器當中，常用的編譯時期函數有 `defined()`, `#error`, `#warning`, `#pragma`，以下我們將介紹這些編譯時期函數的用法。

`defined()`：檢查某的巨集定義是否被定義了，通常與 `#if` 合用，像是 `#if defined(OS)`

...



`#error`：在編譯時，輸出錯誤訊息，警告使用者某些錯誤，並且不會真的進行編譯，在目標檔案中，會輸出錯誤訊息。

`#warning`：在編譯時，輸出警告訊息，警告使用者某些注意事項，但是不會中止編譯，仍然會繼續編譯。

`#pragma`：用來告知編譯器某些特殊指示，例如不要輸出錯誤訊息，抑制警告訊息，或者力

雖然 `#error` 與 `#warning` 後的訊息並不需要加上字串式的引號，但是建議最好還是加上，以避免特殊符號所產生的一些問題，導致巨集處理器的誤解。以下是這兩種狀況。

不好：`#warning Do not use ABC, which is deprecated. Use XYZ instead.`

較好：`#warning "Do not use ABC, which is deprecated. Use XYZ instead."`

程式範例

檔案：`macroFunc.c`

```
#include <stdio.h>

#if !defined( HELLO_MESSAGE )
    # error "You have forgotten to define the header file name."
#endif

char *format = "%s",
      *hello = HELLO_MESSAGE;

int main() {

    printf ( format, hello );

}
```

## 執行結果

```
D:\cp>gcc macroFunc.c -o macroFunc
macroFunc.c:4:6: #error "You have forgotten to define the header f:
macroFunc.c:8: error: `HELLO_MESSAGE' undeclared here (not in a fur

D:\cp>gcc -DHELLO_MESSAGE=\"Hello!\" macroFunc.c -o macroFunc

D:\cp>macroFunc
Hello!
```

## 編譯指示 -- #pragma, \_Pragma(), GCC dependency, GCC poison, ...

編譯指示 #pragma 是用來告知編譯器某些特殊指示，例如不要輸出錯誤訊息，抑制警告訊息，或者加上記憶體漏洞檢查機制等。這些指示通常不是標準的 C 語言所具備的，而是各家編譯器廠商或開發者所制定的，以便讓編譯器可以具有某些特殊的選項。

舉例而言，#pragma STDC 就可以用來要求編譯器採用標準 C 的語法進行編譯，只要看到有任何不符合標準 C 的語法，編譯器就會輸出錯誤。

### #Pragma message

```
#ifdef _X86
#pragma message("_X86 defined") // 在編譯時輸出 _X86 defined
#endif
```

### #Pragma warning

```
#pragma warning( once:37 43; disable:32; error:17) // 37,43 只警告一

#pragma warning( push ) // 保存目前警告狀態
#pragma warning( once:37 43)
#pragma warning( disable:32 )
#pragma warning( error:17 )
.....
#pragma warning( pop ) // 恢復先前的警告狀態
#pragma warn -100 // Turn off the warning message for warning #100
int insert_record(REC *r) {
    ...
}
#pragma warn +100 // Turn the warning message for warning #100 back
```

### #Pragma once



```
#pragma once // 保證引用檔 (*.h) 只會被引用一次，如此就不需要用「引入防護」
```

## #Pragma code\_seg

```
#Pragma code_seg(["section-name"][, "section-class"])
#pragma code_seg("INIT") // 設定存放於 INIT 區段，開發驅動程式時會用到
extern"C"
void DriverEntry(...) { ... }
```

## #pragma hdrstop

```
#pragma hdrstop // 表示引用檔編譯到此為止，以加快編譯速度。
```

## #pragma startup

```
#pragma startup <func> <priority>
#pragma exit <func> <priority>
```

```
void india(); void usa();
```

**pragma startup india 105**

**pragma startup usa**

**pragma exit usa**

**pragma exit india 105**

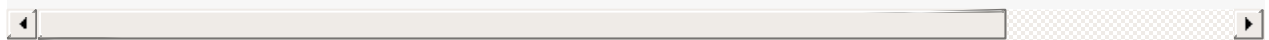
```
void main() { printf("\nI am in main"); getch(); } void india() { printf("\nI am in india"); getch(); } void usa() { printf("\nI am in usa"); getch(); }
```

 執行結果

```
I am in usa
I am in India
I am in main
I am in India
I am in usa
```

### #pragma package(...)

```
#pragma package(smart_init) // 使用某套編譯指引 (在 BCB 中, 根據優先級的
```



### #pragma resource "..."

```
#pragma resource "*.dfm" // 把*.dfm 資源檔加入專案。
```

### #pragma loop\_opt(...)

每個編譯程式可以用#pragma指令激活或終止該編譯程式支援的一些編譯功能。例如，對迴圈優化功能：

```
#pragma loop_opt(on) // 啟動迴圈最佳化
#pragma loop_opt(off) // 停止迴圈最佳化
```

### #pragma asm

```
#pragma asm // 代表後面寫的是組合語言 (Microsoft)
```

### #pragma small

```
#pragma small // 使用小記憶體模式 (Microsoft X86)
```

### #pragma registerbank(..)

```
#pragma registerbank(0) // 使用 8031 處理器中的 bank0 (Keil C)
#pragma code

#pragma code // 表示唯讀資料應儘可能放在 ROM 裡以節省 RAM (Keil C)
```

## 參考文獻

pragma 預處理指令 -- <http://topalan.pixnet.net/blog/post/22334686>

## 字串化 -- Stringification, 使用 #symbol 可以將某符號字串化

使用 #symbol 可以讓巨集處理器將 symbol 符號轉為字串，這個過程稱為 (Stringification)，以下是程式範例。

範例一：將運算式字串化

檔案：stringfication.c

```
#include <stdio.h>

#define WARN_IF(EXP) \
    do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
    while (0)

int main() {
    int x = 0;
    WARN_IF(x == 0);
}
```

執行結果：

```
D:\cp>gcc stringfication.c -o stringfication
stringfication.c:11:2: warning: no newline at end of file

D:\cp>gcc stringfication.c -o stringfication

D:\cp>stringfication
Warning: x == 0
```

範例二：利用字串化取得變數名稱。

檔案：stringfication2.c

```
#include <stdio.h>
// 本程式節錄修改自 TinyCC
typedef struct TCCSyms {
    char *str;
    void *ptr;
} TCCSyms;

#define TCCSYM(a) { #a, &a, },
/* add the symbol you want here if no dynamic linking is done */
static TCCSyms tcc_syms[] = {
    TCCSYM(sprintf)
    TCCSYM(fprintf)
    TCCSYM(fopen)
    TCCSYM(fclose)
    { NULL, NULL },
};

int main() {
    int i;
    for (i=0; tcc_syms[i].str != NULL; i++)
        printf("symbol:%-10s address:%d\n", tcc_syms[i].str, tcc_syms[i].ptr);
}
```

執行結果：

```
D:\cp>gcc stringfication2.c -o stringfication2
```

```
D:\cp>stringfication2
```

```
symbol:sprintf      address:4200528
symbol:fprintf      address:4200512
symbol:fopen        address:4200544
symbol:fclose       address:4200496
```

參考文獻

GCC online document (Stringification) --

<http://gcc.gnu.org/onlinedocs/cpp/Stringification.html#Stringification>