C

Programming Language
An Introduction
C 程式設計語言基礎

Preface

這份文件(或書)是給一個想學 C programming language 的初學者學習用.

C 程式語言是一個多年來廣爲被業界所使用的程式語言, 各種大型系統都可見其蹤影,
學會 C 語言只是一個開始, 代表你可以與其它的 C programmer 溝通,
並看懂成千上萬用 C 開發的系統的程式碼.

劉錦憲. **JSLIU**
E-Mail: jsliu@mail2000.com.tw
Website: http://jsliu.com

劉錦憲

目錄

C 的歷史

C 前身爲 B 語言, 在 Bell Lab (貝爾實驗室) 由當時的研究員 Dennis Ritchie 發展出來, 目的是爲了開發UNIX 作業系統, 替代部份組合語言的工作, 並可在不同的環境上開發系統與執行.

最早的標準是 K&R , 後來 ANSI 在 1982 年制定標準(X3J11)後, 至今的程式設計風格與style 就沒多大改變. X3J11 標準中, 制定了 C 的標準程式庫, 也就是現在大家常用的 printf, scanf, fopen, fclose....這些函數, 最新一次的標準制定是在1999年完成.

ANSI C 與其它 C compiler 的關聯:

ANSI C 可說是各廠商開發 C compiler 時所參考的標準, 出售的產品的包裝上也會註明 ANSI C compatible, 表示遵循 ANSI C 的規範.

只使用 ANSI C 定義的標準將可保證你寫的 C 程式在任一廠牌的 C compiler 上編譯與執行, 只要她的產品上寫明 ANSI C compatible, 而且你的程式也符合 ANSI C 的規範.

ANSI 是 American National Standards Institute 的縮寫, URL 是 http://www.ansi.org/·

C 的優點

1.
大多數的系統都提供 C compiler. 許多電腦硬體的供應商開發出新硬體時, 除了組譯器之外, 會優先考慮做出的系統開發工具就是 C compiler. C programmer 有較多的發展空間.

2.
C 的程式可以容易地轉到其它電腦或作業系統上. 寫程式時符合 ANSI 標準, 並且用的函數都是 ANSI 標準程式, 可以輕易將程式轉移到其它的環境上, 再把與硬體或環境相關的程式碼改寫就可以了.

3.
C 產生出來的執行檔小且快. C 程式相當小, 部份對 C 一知半解的人會說: 用 C 寫一個印"Hello", 編譯出來的執行檔很大. 因爲這些人不瞭解爲什麼, 對 C 的函數庫觀念也不清楚.所以會有這種錯誤的批評.

4.
C 程式很接近硬體層次, 可直接控制大部份的週邊硬體. C 有pointer (指標), 可以存取任意記憶體位址, 許多CPU採用memory mapping I/O, 所以可以用C來控制I/O, 寫硬體控制程式.

開始寫一個 C program.

開始第一個程式:

```
/* ====================
 什麼也不做的程式.
==================== */
int       main()
{
          return 0;          /*  傳回  0 */
}
```

這個程式什麼也不做. C 標準規定 main() 函數 是程式的開始, 主程式的意思,
一個 C 寫的軟體或程式只有一個 main() 函數, 不管你寫多少程式,
一開始執行一定是由 main() 開始.

數學的函數規定一定要傳回一個值, 但 C 的函數(function)可以不用傳回任何東西,
爲了告訴 compiler : main() 這個 function 傳回一個整數, 所以我們在 main()
前面加上 int . 意思就是 main() 這個 function 傳回(return)一個整數的數值,
執行這個程式在結束時, 會將這個整數的數值傳回給作業系統,
在這個程式中, 一進入 main 什麼也不做就傳回 0 給作業系統結束了程式.

C 的一個 block 由大括號組成, main() 這個函數的 body (本體)就是:

```
{
          return 0;/*  傳回  0 */
}
```

大括號的左括號是這個 main() 函數的開始, 右括號是結束, 你也可以寫成:

```
/* ====================
 什麼也不做的程式.
==================== */
int       main()
{         return 0;          /*  傳回  0 */    }
```

只要括號有對齊, 一對一, 幾個左括號就對應幾個右括號就可以了,
但爲了視覺美觀與程式閱讀的便利性, 我們不建議上面的寫法.
另外, C 是(case-sensitive)的程式語言, 也就是大小寫是完全不同,
例如 main() 與 MAIN() 與 Main() 是完全不同的, 若你學過 BASIC 或 PASCAL,
這一點請牢記.

C 的 comment (註解).

```
/* 傳回 0 */
```

上面這行是註解(comment), 這一行不會被 compiler 處理, /* 跟 */ 包起來的
文字就稱註解, 所以程式的最前面也是一段註解:

```
/* ====================
  什麼也不做的程式.
==================== */
```

通常我們會在程式中加一些註解, 方便我們在日後維護程式時理解, 當程式共寫了
幾千行到幾萬行時, 適當的加一些註解可以便於我們瞭解整個程式是在做什麼事,
在正式的程式開發中, 加入適當或詳細的註解已經是一個程式設計員的基本要求.
底下列出一些常見的註解風格:

```
/**************************
  * 什麼也不做的程式.
**************************/
```

或

```
/* ------------------------------
-   什麼也不做的程式. -
------------------------------ */
```

若是一份正式的程式, 可能還會加上一些程式的檔案名稱, 開發日期, 作者, 最後修改日....,
例如:

```
/* ====================
  file name: c1.c

  功能: 什麼也不做的程式.
  環境: DOS, Windows, UNIX...etc.
  作者: J.S Liu.

  開始日期: 2000/1/1.
  最後日期: 2000/12/31.
==================== */
```

要注意的是, C 不能用巢狀註解, 譬如:

```
/*        這是外層註解      /* 內層註解 */              */
```

使用 C 的標準輸出函數

先看以下的程式:

```
/* ====================
   Say Hello World!.
==================== */
#include <stdio.h>
void          main()
{
            /*  印出  Hello */
          printf("Hello World!");
}
```

執行結果:

```
Hello World!
```

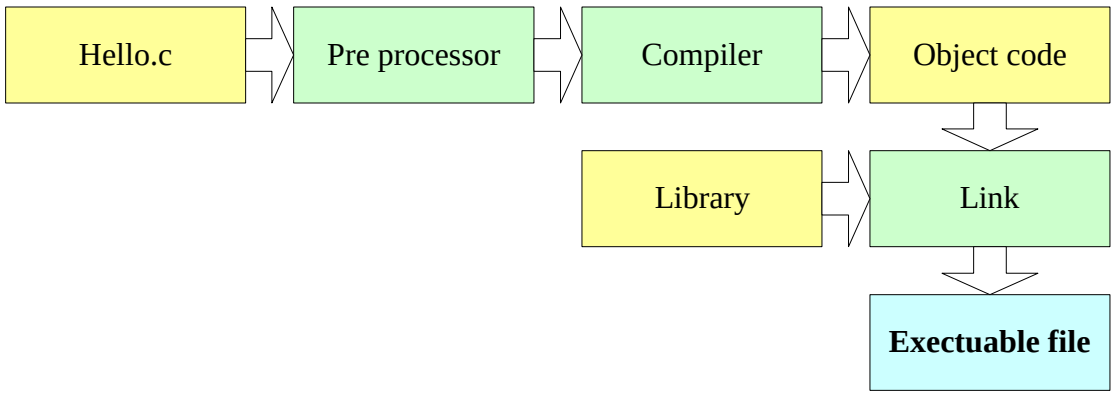
程式中:

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

這一行是把 stdio.h 包進來一起 compile, 這個檔案是 ANSI 標準, 裡面放了許多 C 標準 I/O (輸出輸入) 函數 (functions) 的定義(definition), 寫這一行的目的簡單講是因爲在程式中使用了printf 這個函數輸出字串到螢幕上, 這個函數的定義放在 stdio.h 中. ANSI C 規定你用到的函數或變數必需要有定義.

如果還是不太懂, 那麼就先記住你要用到 printf 就先加上 #include <stdio.h> 這一行, 在後面將會更詳細解釋爲什麼.

另外, #include 這個指令並不是 C 語言的一部份, 而是 C 的 pre-processor 處理的, 處理之後才把 code 丟給 compiler, 同樣的, 在後面會提到這一點,
以下是整個程式編譯的過程:



C 的每一段敘述結束要用 ; (分號) . 記住是一段敘述, 不是一行, 舉例來說:
你可以在一行裡面寫好幾個 printf("Hello World"), 都用分號隔開, 就像底下這段程式:

```
/* ====================
   Say Hello World! Bye Bye.
==================== */
#include <stdio.h>
int       main()
{
          /* 印出 Hello World! Bye Bye */
          printf("Hello World! "); printf("Bye "); printf("Bye");
          return 0;
}
```

爲了程式閱讀方便並不建議你寫成上面這樣, 最好是一個敘述一行, 如下.

```
/* ====================
   Say Hello World! Bye Bye.
==================== */
#include <stdio.h>
int       main()
{
          /* 印出 Hello World! Bye Bye */
          printf("Hello World! ");
          printf("Bye ");
          printf("Bye");
          return 0;
}
```

程式執行結果:

```
Hello World! Bye Bye
```

上面的程式看起來就容易理解多了, 軟體或程式的開發注重容易理解與方便溝通,
良好的程式風格與適當的註解也才是一個好的程式.

C 的基本型態 (basic types)

以下是 C 中常見的標準型別(型態).

型 態 名 稱	意 義
char	字元
int	整數
float	浮點數
double	倍精準浮點數
void	void

以下是修飾詞, 可搭配標準型態的宣告.

修 飾 詞 名 稱	意 義
long	長
short	短

與 BASIC 不同的是, C (與 PASCAL) 在使用一個東西(函數或變數)時必需要先宣告或是定義.

以下是一個整數型別的程式範例:

```
/* ====================
  變數宣告的例子 1.
==================== */
int       main()
{
          /*  變數宣告  */
          int         a;
          int         A;
          int         b, c;

          return 0;
}
```

在上面程式中, 變數宣告區存放一些我們所需要的變數, 並且宣告這些變數的型態, 我們在上面共宣告了 a, A, b, c 這4個變數, 型態都是 int (整數). 最後什麼也不做, 就離開並傳 0 給 O.S .

再看來看第二個變數宣告的例子.

```
/* ====================
  變數宣告的例子  2.
==================== */
int       main()
{
          /*  變數宣告  */
          int          a;
          int          A;
          int          b, c;

          a = 1;
          A = 8;
          b = 2;

          c = A - a + b;              /*  先計算  A - a + b,  將結果傳會給  c */
          printf( "%d", c );                  /*  以  printf  印出  c  這個整數型態的變數  */
          return 0;
}
```

先設 a = 1, A = 8, b = 2, 把 A - a + b 算出來之後存到 c 這個變數, 再以 printf 函數
輸出 c 這個整數型態的變數至螢幕上.
程式執行結果:

```
9
```

C 允許 programmer 在宣告一個變數時, 就給定常數(constant)值, 所以你也可以寫成:

```
/* ====================
  變數宣告的例子  3.
==================== */
#include <stdio.h>
int       main()
{
          int          a = 1;
          int          A = 8;
          int          b = 2, c;

          c = A - a + b;
          /*  輸出  a, A, b, c  到螢幕  */
          printf( "a = %d, A = %d, b = %d, c = %d ", a, A, b, c );

          return 0;
}
```

程式執行結果:

a - 1, A = 8, b - 2, c = 9

配合修飾詞的情況.

型 態 名 稱	意 義
long int	長整數
short int	短整數
long double	長倍精準浮點數

若宣告時只寫 long, 而未加 int 或 double, 則一律視爲 long int.
若只寫 short 而未加 int, 則一律視爲 short int.

```
/* ====================
  修飾字 long and short 的宣告範例.
==================== */
int       main()
{
          long              a;
          long     int      b;

          short             m;
          short    int      n;

          return 0;
}
```

程式中宣告了4個變數, 分別是 a, b, m, n.
其中 a 與 b 的型態是一樣的, 都是 long int, 而 m 與 n 也是一樣的, 都是 short int.

C 的浮點數

以下是浮點運算的程式範例:

```
#include <stdio.h>
void            main()
{
        float     a = 0.5;
        double  b = 1.2;
        int       c = 3;

        b = b + a + c;
        /*  輸出  a, b, c  到螢幕  */
        printf( " a = %3.1f, b = %3.1f, c = %d ", a ,b, c );
}
```

程式執行結果:

```
a = 0.5, b = 4.7, c = 3
```

我們宣告了 a, b, c, 3 個變數, 並分別給予的型態是 float, double, int. 也分別給了初值.
把 b + a + c 計算之後的結果再存到 b.
你在一些編譯器上編譯這個程式時, 會出現一些警告(warring)的訊息, 內容大概是:
“type mismatched” 之類的訊息.
因爲在這一行裡面, 出現了不同型態(type)的變數進行運算,
雖然 C compiler 會幫你轉型(casting)進行運算, 但建議你最好自己轉型,
轉型的寫法是:

```
b = b + (double)a + (double)c ;
```

你把運算結果存放的變數是哪一種型態 (type), 就在後面的變數或運算子前加上
(type). 這種轉型只是暫時性的, 爲了在運算過程中讓所有的運算子變數型態都
保持一致.

C 的字元

以下是字元型態的範例:

```
/* ====================
  字元範例 1
==================== */
#include <stdio.h>
int       main()
{

          char            x, y;
          x = 'a';
          y = (char)97;

          /* 輸出 x, y, x, 最後一個是以 ASCII 值顯示 y */
          printf( " x = %c, y = %c, ASCII of y = %d", x, y, y );
          return 0;
}
```

在 PC 上任一種 O.S 的執行結果:

```
x = a, y = a, ASCII of y = 97
```

字元型態的變數以 char 宣告, 在 C 中, 字元是用單引號 ' 括起來, 如下幾個例子:

```
‘b’       /* 字元 b */
‘C’       /* 字元 C */
‘=’       /* 字元 = */
‘&’       /* 字元 & */
```

你也可以直接設定一個整數值給字元型態的變數, 在上面的程式中, 我們寫了:

```
y = (char)97;
```

你可以查一下 ASCII 中, 字元 97 是什麼字, y 就會是那個字.
97 的前面加上 (char) 是爲了轉型所需, 你應該知道吧.
這初學者最大的疑惑就是爲何查 ASCII 的字元集的 97 , 且爲何會輸出一個 'a',
這與機器及環境有關, 有些機器上, 97 可能是 'X' 或是其它字元,
做爲一個C programmer, 在你開發程式之前, 你應當要熟知所在機器上的字元集.
例如一些 IBM 大型主機(mainframe)上是用 EBDIC 字元集而不是常見的 ASCII 字元集,
環境的不同將造成程式在不同平台開發及被執行時有所不同.

常數數字表示法

底下的程式範例:

```
/* ====================
  digital - 1
==================== */
#include <stdio.h>
int       main()
{
          int       a = 64;
          int       b = 0x40;
          long      c = 64L;

          printf("%d,%d,%d", a, b, c );
          return 0;
}
```

執行結果:

```
64, 64, 64
```

程式中 0x40 代表的是一個 16 進位的數字, 也就是每 16 進一, 與一般常用的 10 進位不同. 0x40 與 10 進位的 64 值相同.

64L 代表的是 64 這個數值的 type (型態)是 long. 不加 L 則表示是 integer (整數型態).

Basic types (基本型態) 的大小 (storage size)

常常有許多人會把 int 當16-bit, 可以表示 32767到-32768.
這是錯誤的說法, 因爲在 32-bit 的 C compiler 上, int 通常是 32-bit.
再提到 long (長整數), 在 WATCOM C/C++ 386 版的 long 跟 int 一樣是32 bit,
但在 Symantec C/C++ 6.X 與 Borland C/C++ 4.X 的 386 compiler 中 long 是 64 bit.
K&R 及 ANSI 沒規定上面這些基本型態的長度, 因爲規定這些型態的大小之後
會產生另一些問題, 譬如某些系統是16-bit,但規定了32-bit 長的 int, 在運算時會因轉換
而變慢一些.

某些系統是 32-bit, 但 int 只有 16-bit, 程式又沒辦法有效利用32-bit 的運算效能,
等於還是拿 32-bit電腦在跑 16-bit 的程式.
一些 8-bit 時代開發的 C 程式甚至有部份 code 還繼續用在 16-bit 甚至 32-bit 的平台上,
若是定義了基本型態的大小, 當初 8-bit 時的 int 是 8-bit 長, 到了 16-bit 平台上
勢必會變慢, 因爲每次整數運算都要轉換成 8-bit 之後再運算.
這些例子說明是讓初學者知道當初 C 的設計者爲什麼不定義這些基本型別的大小.

不規定這些基本型態的長度並不會影響 C 的可攜性,
因爲可以利用 C 的 macro (巨集) 來解決這些問題, 在以後我們會談到如何使用 C
的巨集及前置處理器 (C Pre-Processor, 簡稱 CPP) 來解決可攜性的問題.

C 的輸出與輸入的函數簡介

本節中介紹兩個 C 的函數, 一個是 printf, 另一個是 scanf.

printf() 是一個功能強大的函數, printf 是 ANSI C 制定在標準的 C 程式庫
中的一個函數. C 是一個很小的程式語言, 最早的 C 只有28個關鍵字(keywords),
早期的 C 沒有這些函數可以用, 每個程式設計人員都要自己寫自己的輸出與輸入
基本函數給自己用, 後來 K&R 與 ANSI 制定了標準, 在標準中規範了這些函數,
所以你不需要寫自己的列印函數, 因爲這些東西已經有標準的函數了.
任何一家公司推出的 C compiler 如果宣稱符合 ANSI C 標準, 那麼她一定會提供
這些程式庫與函數, 只要你寫的程式只用這些標準函數, 沒用到特殊的函數,
那麼你的程式將具有良好的可攜性, 也就是你的程式可以在不同的 C compiler 上
編譯(compile)通過, 不會受限在某一種品牌的 C compiler 上.

舉個例子, 早期使用 Turbo C 的 programmer, 用了 gotoxy, putpixel, putimage...,
這些函數, 結果程式拿到 MS C 或 Visual C 上就出現一堆錯誤訊息, 因爲這些函數
不是標準函數, 是當時 Borland 爲了吸引 programmer 購買, 所以附了這些函數在
產品中, 你用了這些不是 ANSI 標準的函數, 就代表你寫的程式將會無法在其它
的 C compiler 上編譯使用, 因爲其它的 C compiler 並不一定提供這些函數.

現在要介紹 printf 這個函數的一些其它功能, 更詳細的介紹請自行查閱
C 標準程式庫的書籍文件.

列印整數, 以十進位方式印出:

```
printf( " c is %d", c );          /* 用 %d */
```

列印整數, 以十六進位方式印出:

```
printf( " c is %x", c )           /* 用 %x */
```

列印浮點數:

```
printf( " a is %f", a );          /* 用 %f */
```

列印浮點數, 正整數印 5 位數, 小數點後印 2 位數.

```
printf( " b is %5.2f", b );           /* 用 %5.2, 印 5 位正整數, 2 位小數 */
```

列印一個字元.

```
printf( " x is %c", x );              /* 用 %c 印出一個字元 */
```

列印一個字串.

```
printf( " s is %s", s );                /*  用  %s  印出一個字串  */
```

換行. 換行可用 \n.

```
printf( "換行\n");
```

輸出一個 tab. 用 \t

```
printf( "\t" );
```

輸出一個雙引號 ". 用 \"

```
printf( "\"");
```

輸出倒斜線. 用 \\.

```
printf( "\\" ) ;
```

printf 可接受不定數目的參數, 意思是這個函數沒有規定只能放多少參數, 例如:

```
printf( " a is %d, b is %f, c is %f", a, b, c );
```

若要從鍵盤輸入資料則可以使用 scanf 這個函數, 先看範例再解說,
以下是個簡單的例子:

```
/* ==========================
 輸入一個字元
========================== */
#include <stdio.h>
int       main()
{
          char      ch;

          printf("Input a char:");
          scanf( "%c", &ch );                /* ch  前面加個  &(位址運算元) */
          return 0;
}
```

scanf 與 printf 類似, 差別是 printf 是輸出, scanf 是輸入, 但 scanf 的參數是給位址,
不是直接傳值, 因此我們在變數 ch 前加上位址運算爲 &ch,
關於位址運算元將在後面指標的章節深入提到, 這裡你只要先記得使用時加上 & 就好.

再看另一個輸入整數的範例:

```
/* ========================
 輸入一個整數
======================== */
#include <stdio.h>
int     main()
{
        int     i;

        printf("Input an integer:");
        scanf( "%d", &i );              /* ch  前面加個  &(位址運算元) */

        printf( "the number is %d",    i );
        return 0;
}
```

程式執行結果:

```
Input an integer:16
the number is 16
```

16 是輸入的整數值, 輸入之後將會存放到變數 i , 接著再用 printf 輸出 i.

指定運算元 (Assignment Operator) - 1

C 的指定運算元有很多種, 先介紹最常用的 = 運算元, 其它的指定運算元在後面介紹.

運 算 符 號	意 義
=	右值指定給左值

程式範例:

```
/* ====================
  = 運算元的範例 1
==================== */
void       main()
{
          int        a, b;
          float        c;

          a = b = 5 + 1;
          c = (float)a + 0.1;
}
```

程式中:

```
a = b = 5 + 1;
```

這一行程式應該很容易理解, 但底下這一行需要再說明一下:

```
c = (float)a + 0.1;
```

首先我們知道指定運算元是把右邊的運算式指定給左邊,
但兩者型態不同時, 建議最好是自己轉型(casting),
方法很簡單, 就是在不同型別的變數前面加上(type),
type 與最右邊變數的型別一樣, 盡量保持整個運算式的型別全部一致.

其它基本型別的指定則依此類推, 在此不再詳述.

算術運算元 (Arithmetic Operator).

基本的算數運算, 包含四則運算及餘數運算.

運 算 符 號	意 義
+	加法
-	減法
*	乘法
/	除法
%	求餘數

程式範例:

```
/* ====================
  基本運算範例.
==================== */
#include<stdio.h>
int       main()
{
          int              a,b;
          a = 10;       b = 3;

          printf( "%d \n", a * b );
          printf( "%d \n", a / b );
          printf( "%d \n", a + b );
          printf( "%d \n", a    - b );
          printf( "%d \n", a % b );
          return 0;
}
```

執行結果:

```
30
3
13
7
1
```

關係運算元 (Relational Operator)

運 算 符 號	意 義
<	小於
<=	小於等於
>	大於
>=	大於等於
==	測試兩邊是否相等
!=	測試兩邊是否不相等

這些關聯運算的運算結果只有 0 與 1 兩種結果, 關聯運算通常用在流程控制上,
像是 if - else, while. do-while, for, 將在以後的流程控制中詳細介紹其應用,
現在先看基本的關係運算元及其結果.

程式範例:

```
/* ====================
  關係運算元的範例.
==================== */
#include <stdio.h>
int       main()
{
          int         a = 10, b = 5;

          printf( " a == b is %d \n", a == b );
          printf( " a > b is %d \n", a > b );
          printf( " a < b is %d \n", a < b );
          printf( " a >= b is %d \n", a >= b );
          printf( " a <= b is %d \n", a <= b );
          printf( " a != b is %d \n", a != b );

          printf( "\n" );

          b = 10;

          printf( " a == b is %d \n", a == b );
          printf( " a > b is %d \n", a > b );
          printf( " a < b is %d \n", a < b );
          printf( " a >= b is %d \n", a >= b );
          printf( " a <= b is %d \n", a <= b );
          printf( " a != b is %d \n", a != b );

          return 0;
}
```

執行結果:

```
a == b is 0
a > b is 1
a < b is 0
a >= b is 1
a <= b is 0
a != b is 1

a == b is 1
a > b is 0
a < b is 0
a >= b is 1
a <= b is 1
a != b is 0
```

學過其它程式語言的人有時也會被此迷惑, 事實上這也是 C 語言簡單及強大的地方, 接近硬體與機器語言, 就像上面的程式中, a > b 計算的結果會傳回一個 0 或 1.

位元運算元 (Bitwise Operator)

在早期, C 語言優於其它程式語言之處, 除了 pointer, 位元運算元也是其中之一, 這些運算元都是 C 內建(built-in)的功能.

運 算 符 號	意 義
~	補數運算
<<	位元往左移位
>>	位元往右移位
&	AND
^	XOR
|	OR

先來看位元運算的一個範例:

```
/* ====================
  位元運算元的範例.
==================== */
#include<stdio.h>
void       main()
{
          int              a,b;
          a = 15;
          b = 1;

          printf("%d \n", a | b );                    /* a OR b */
          printf("%d \n", a & b );                    /* a AND b */
          printf("%d \n", a ^ b );                     /* a XOR b */

          printf("%d \n", a << 1 );                  /* a  位元左移  1  位  */
          printf("%d \n", a >> 1 );                  /* a  位元右移一位  */
          printf("%d \n", ~a );                        /* A  的補數運算  */
}
```

執行結果:

```
15
1
14
30
7
-16
```

在某些 compiler 上出現的數值不同, 但這是因爲 C 未定義基本型別的大小的原因, 查一下基本型別大小再對照執行結果, 將更瞭解這些運算元的功能.

關於 OR, AND, XOR…, 這些運算元的意思不加以詳述.

指定運算元 (Assignment Operator) - 2

以下的指定運算元只是結合基本運算再加上 = 這個指定運算元.
有些人並不建議你用 = 與 ++ 與 -- 以外的指定運算元, 原因是程式的可讀性差,
但無論如何你在學習時必需要知道一下這些運算元.

運 算 符 號	範 例	等 價 運 算 式
+=	a += b	a = a + b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
-=	a -= b	a = a - b
%=	a %= b	a = a % b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b
++	a++	a = a + 1
--	a--	a = a - 1
&=	a&= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b

邏輯運算元 (Logical Operator)

C 有3個邏輯運算元, 基本上常與條件判斷運算合用, 例如 if, while, for.

運 算 符 號	意 義
!	Logical NOT
&&	Logical AND
||	Logical OR

程式範例:

```
/* ====================
 Logical NOT.
==================== */
#include <stdio.h>
void    main()
{
        int         a;

        a = 3;
        printf("%d\n", !a );

        a = 0;
        printf("%d\n", !a );
}
```

執行結果:

```
0
1
```

(!)這個運算元的傳回值只有兩個: 0 或 1. 當你把一個非 0 值給 ! 運算元運算之後, 會傳回 1, 若是 0 則傳回 1. 很簡單吧.

Logical AND 的範例:

```
/* ====================
 Logical AND.
==================== */
#include <stdio.h>
void    main()
{
        printf("%d\n", 1 && 3 );
        printf("%d\n", 0 && 0 );
        printf("%d\n", 2 && 2 );
}
```

執行結果:

```
1
0
1
```

(&&)運算元傳回值與 (!) 一樣, 當 (&&) 運算元兩邊的運算子之值其中一個爲 0, 則會傳回 0, 否則傳回 1.

Logical OR 的範例:

```
/* ====================
 Logical OR.
==================== */
#include <stdio.h>
void    main()
{
        printf("%d\n", 1 || 0 );
        printf("%d\n", 0 || 0 );
        printf("%d\n", 2 || 2 );
}
```

執行結果:

```
1
0
1
```

(||)運算元傳回值與 (&&) 一樣, 但不同的是, 當 (||) 運算元兩邊的運算子之值都爲 0 時, 則會傳回 0, 否則傳回 1.

其它的運算元

sizeof(type) 取得特定型別的 size

```
/* ====================
  sizeof 的範例.
==================== */
 #include <stdio.h>
void     main()
{
         char     a;

         printf( " The size of     int is %d \n", sizeof(int) );
         printf( " The size of     char a    is %d \n", sizeof(a) );
}
```

執行結果:

```
The size of int is 4
The size of char a is 1
```

注意的是, 執行結果會因爲你所用的編譯器不同, 可能不同, 譬如在 DOS 下的 Turbo C, Microsoft C, 其 int 長度是 2, char 是 1. 在新一代 64 bit CPU 的電腦上, 有些 C compiler 的 int 長度是 8. 用 sizeof 可以得知你目前使用的 compiler 對某些型別的長度. 其它尚未提到的位址運算元, 將在指標的地方介紹.

藉著 sizeof 運算, 我們可以取得目前機器作業平台上的 C compiler 的基本型態的存放大小.

函數 (Function)

之前介紹的 printf 是 C 的標準函數, 也就是拿別人寫好的工具在用, 在開發應用系統時,
許多時候都需要自行寫一些函數給自己或同僚使用. 現在要教你如何自己寫一個函數,
並使用自己寫的函數.

```
/* ====================
  Function (1)
==================== */
#include <stdio.h>
/* ====================
  計算長方型面積, 需傳入長與寬.
==================== */
int          rect( int x, int y )
{

          int          result;
          result = x*y;
          return reslut;                  /*  傳回  result */
}
/* ====================
  main function
==================== */
void      main()
{
          int        x = 8, y = 4;
          int        a;

          a = rect( x, y );
          printf( "8*4  的面積是  %d", a );
}
```

我們寫了一個 area 的函數, 這個函數需要給 2 個整數參數,
使用這個函數計算 2 個整數相乘後的結果, 並把這個結果列印輸出
至螢幕上, 先看第 1 行:

```
int        rect_area( int x, int y )
```

area 前面放 int 是代表用了 area 這個函數後, 會傳回一個整數回去.
再看:

```
return result;              /*  傳回  result */
```

這一行是把 result 傳回去.

也可以用下面這種寫法:

```
return( result );          /*  傳回  result */
```

這樣看起來比較能清楚顯示出要傳回的變數是 result, 不過請記得, return 不是 function.
以下是另一種範例及寫法:

```
/* ====================
  Function (2)
==================== */
#include <stdio.h>
float          circle( int r );              /*  宣告  circle  的  prototype */
void           main()
{
          float      answer;
          answer = circle(8);
          printf( "  圓周長度是    %f", answer );
}
/* ====================
  circle  函數, 計算  circle  的圓周長
==================== */
float          circle( int r )
{
          float      result;
          result = 3.14159 * (double)2 * r;
          return ( result );
}
```

我們在 main() 裡面用了 circle() 這個函數, 但是因爲 circle() 函數是寫在後面,
compiler 在編譯到:

```
answer = circle(8);
```

這一行時, 必需要知道 circle() 的定義, 也就是 compiler 必需要清楚地
知道cirlce 會傳回什麼型態的資料, 需要傳入幾個參數, 這些參數又是
什麼型態...., 等等的資訊. 所以我們在第 2 行寫了:

```
float          circle( int r );              /*  宣告  circle  的  prototype */
```

C compiler 處理時, 會把這一行讀進去, 等你用到 circle 時,
compiler 先檢查你呼叫 circle 用法有沒有錯, 譬如參數型態
跟數目有沒有不正確? 沒有錯的話就繼續編譯, 最後連結(link)時,
才跟 circle 的實體機器碼連結一起成一個可執行檔.

這也是爲什麼我們用了 printf 這個函數就需要寫這一行:

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

因爲 printf 的 prototype 就是放在 stdio.h 中.
還不太懂嗎? 試著把

```
float       circle( int r );          /* 宣告 circle 的 prototype */
```

這一行刪除然後再編譯一次, 看看出現什麼樣的警告或是錯誤訊息.
簡單說明一下 C compiler 如何處理這個程式:

1. 含入 #include <stdio.h> , 把裡面的定義與宣告讀入記憶體存放.

2. float circle(int r); /* 宣告 circle 的 prototype */
   把這一行對於 circle 函數原型(protype)宣告讀入存放.

3. 用到了 circle 這個函數, 檢查在記憶體裡面有沒有 circle ?
   有的話把兩者比對檢查, 看看型別與參數有沒有對應?

4. 接著
   printf(" 圓周長度是 %f", answer);
   這一行, 處理方式跟 circle 一樣, 比對型別有沒有錯誤.

其它細節與後面的 object file or library 的 link 部份不在此解釋,
你用到的任何一個物體, 必需要有明確的原型(prototype)宣告,
函數原型的宣告跟變數原型的宣告很類似, 目的是告訴 compiler (編譯器),
方便 compiler 做型別檢查與編譯程式.
另一個問題是, 爲什麼第一個程式把 area 放在前面就不用做原型宣告呢?
因爲函數的 head 跟 prototype (原型)是一致的, compiler 編譯到 area 函數時
找不到area 的 prototype, 就把 area 的 head 當原型放入記憶體中,
你也可以試試把原型改成跟函數完全不同之後編譯看看會有什麼錯誤訊息,
看看這些錯誤訊息可以更瞭解C 及C compiler 的運作.

基本上由此你就可以大略瞭解爲什麼有人說 C 很小, C 是函數構成, 你用了更多的
函數, 你的程式就會變大, 你用的函數如果功能很強, 像 printf 這種強大功能的函數
時, 你會發現編譯出來的可執行檔變得很大, 你也可以自己寫個簡單一點的 print
函數而不使用標準程式庫的 printf, 這些都是 C programmer 應該知道的.

C 的流程控制一 (**if-else**).

```
if(     expression    )
          statement;
```

運算式的結果不是 0 (not zero), 則執行 statement, 如果要執行的 statement 有 2 個或以上時, 可用下列的格式:

```
if(     expression    )
{
          statement;
}
```

程式範例:

```
/* ====================
  if 的範例1.
==================== */
#include <stdio.h>
void      main()
{
          char      ch;

          printf( "input a char:" );
          scanf( "%c", &ch );
          if( ch == 'a' )
                    printf( " You pressed 'a' " );
}
```

程式執行時, 請輸入一個字元後按 ENTER 鍵, 若輸入的是 a, 則會印出一段 " You Pressed 'a'" 這一段文字.

程式也可寫成:

```
/* ====================
  if  的範例1.
==================== */
#include <stdio.h>
void      main()
{
          char      ch;

          printf( "input a char:" );
          scanf( "%c", &ch );
          if(    i == 'a' )
          {
                    printf( " You pressed 'a' " );
          }
}
```

執行結果一樣, 通常爲了可讀性, 建議是一律加上大括號, 當然有些人不如此認爲,
不過我的習慣是一律加上去.

```
/* ====================
  if  的範例2.
==================== */
#include <stdio.h>
#include <conio.h>
void      main()
{
          if( 1 )
          {
                    printf( "TRUE! " );
          }
}
```

執行本程式一定會印出 "TRUE!".
其餘可參考前面的邏輯運算元及關係運算元, 傳給 if 的值不爲 0, 則會執行 printf("TRUE!");

再看另一個範例:

```
/* ====================
 if 與 else 的範例 1.
==================== */
#include <stdio.h>
void    main()
{
        char    ch;

        printf( "input a char:" );
        scanf( "%c", &ch );
        if(   i == 'a' )
        {
                printf( " You pressed 'a' " );
        }
        else
        {
                printf( " You didn't press 'a'" );
        }
}
```

程式執行之後, 請輸入一個字元, 若是 'a' 則會印出 " You pressed 'a'",
否則會印出 " You didn't press 'a'".

if-else 可以巢狀使用, 以下是一個巢狀的 if-else 範例.
判斷輸入的數值是小於 100 或 大於等於 100 但小於 200 或是大於等 200.

if – else 範例 2:

```
/* ====================
 if 與 else 的範例 2.
==================== */
#include <stdio.h>
void    main()
{
        int     i;

        printf( "input an integer:" );
        scanf( "%d", &i );
        if(   i < 100 )
        {
                printf( "i < 100" );
        }
        else
        {
                if( (i >= 100) && (i < 200) )
                {
                        printf( "i >= 100 and i < 200"' );
                }
                else
                {
                        printf("i >= 200");
                }
        }
}
```

C 的流程控制二 (**switch case**).

switch 與 case 是一起使用的, switch 的語法是:

```
switch( status )
{
        ...
}
```

status 只能爲整數長整數或字元.

case 是依 status 是否相同, 相同則執行後面的敘述句.

```
case        status:
```

default 是當所有的 case status 都不發生時, 就會執行 default 後的程式.
每一 case 的結束可用 break, 若不放 break, 則會執行下一個 case,
這地方將以程式範例解說, 以下是程式範例:

```
/* ====================
  switch - case 的範例 1.
==================== */
#include <stdio.h>
void          main()
{

        char     c;

        printf( "Input a char:" );
        scanf( "%c", &c );

        switch( c )
        {
        case        'a':
                printf(" you pressed a ");
                break;
        case        'b':
                printf(" you pressed b ");
                break;
        case        'c':
                printf(" you pressed c ");
                break;
        default:
                printf(" not a, b, c ");
                break;
        }
}
```

執行結果:

```
你按了 'a' 會出現 "you pressed a".
按了 'b' 會出現 "you pressed b".
按了 'c' 會出現 "you pressed c".
按其它鍵會出現 "not a, b, c "
```

以下是程式範例 2 (不加 break 的結果):

```
/* ====================
  switch - case 的範例 2.
==================== */
#include <stdio.h>
void          main()
{

          char     c;

        printf( "Input a char:" );
        scanf( "%c", &c );

        switch( c )
        {
                case        'a':     printf(" you pressed a \n");
                case        'b':     printf(" you pressed b \n");
                case        'c':     printf(" you pressed c \n");
                default:            printf(" not a, b, c ");
        }
}
```

以下是程式範例 3 (沒有 default 的情況):

```
/* ====================
  switch - case 的範例 3.
==================== */
#include <stdio.h>
void          main()
{

        char     c;

        printf( "Input a char:" );
        scanf( "%c", &c );

        switch( c )
        {
        case        'a':
                printf(" you pressed a ");
                break;
        case        'b':
                printf(" you pressed b ");
                break;
        case        'c':
                printf(" you pressed c ");
                break;
        }
}
```

執行結果:

```
你按了 'a' 會出現 "you pressed a"
                                    "you pressed b"
                                    "you pressed c"
                                    "not a, b, c "

按了 'b' 會出現 "you pressed b"
                                "you pressed c"
                                "not a, b, c "

按了 'c' 會出現 "you pressed c".
                                "not a, b, c "
```

以下範例是整數的情況:

```
/* ====================
  switch - case 的範例 4.
  印出 odd, 或 even.
==================== */
#include <stdio.h>
void            main()
{
            int         i;

            printf( "Input a number:" );
            scanf( "%d", &i );

            i = i % 2;

            switch( i )
            {
            case          0:
                        printf("Even!");
                        break;
            case          1:
                        printf("Odd!");
                        break;
            }
}
```

C 的流程控制三 (**for**).

C 的迴圈計有: for, while, do-while幾種.
先來看一個for迴圈的範例:

```
/* ====================
  Program 1 - for
    ==================== */
#include<stdio.h>
void                main()
{
        int         i;

        for( i =0; i < 6; i++ )             /*  迴圈開始  */
                printf("%d\n", i );        /*  印出  i */
}
```

執行結果:

```
0
1
2
3
4
5
```

for迴圈共有三個 expression 如下:

for(*expression1* ; *expression2* ; *expression3*)

expression1是初始設定, 迴圈開始前執行一次, 以後不再執行.
expression2是迴圈條件判斷, 當 expression2 執行結果爲 0 時則不再執行迴圈.
所以在上例中的 (i < 6) 在 i = 0, i = 1, i = 2, i = 3, i = 4, i = 5 時, 都回傳回 1,
對此若有不明白之處請參見前面運算元的章節.
expression3除了第一次之外, 每次迴圈執行時都會被執行一次.

原程式的for寫法:

```
for( i =0; i < 6; i++ )
```

expression1的設定是 i = 0,
expression2 繼續執行迴圈的條件是 i < 6.
expression3 每執行一次迴圈就會被執行到, 內容是 i++.

以下是 for 迴圈的第 2 個程式範例:

```
/* ====================
  Program 2 - for
    ==================== */
#include <stdio.h>
void        main()
{
        int     i, j;

        for ( i = 0, j = 10;    i < 6; i++, j++ )
        {
                printf( "i = %d, ", i    );
                printf( "j = %d \n", j );
        }
}
```

執行結果:

```
i = 0, j = 10
i = 1, j = 11
i = 2, j = 12
i = 3, j = 13
i = 4, j = 14
i = 5, j = 15
```

這個範例說明 for 迴圈中的初值運算式 與 變更運算式可以放一個以上,
以 , 逗號隔開.

```
for( i = 0, j = 10; i < 6; i++, j++ )
```

上面這行程式中, 一開始我們設了 2 個初值給 i, j, 分別是 0 與 10,
中間的條件判斷是 i < 6, 如果中間不成立, 就不執行迴圈,
最後的變更運算式有兩個:

i++ 跟 j++.

也就是每執行完一次迴圈就把 i 跟 j 各累加 1.
以下幾個程式跟上面的程式是等價的, 請多比較與瞭解.

範例 3:

```
/* ====================
  Program 3 - for
    ==================== */
#include <stdio.h>
void        main()
{
        int     i, j;
        j = 10;                     /*  迴圈外先設定初值  */
        for ( i = 0;    i < 6; i++ )
        {
                printf( "i = %d, ", i    );
                printf( "j = %d \n", j );
                j++;                /*  迴圈後加上變更運算式  */
        }
}
```

範例 4:

```
/* ====================
  Program 4 - for
    ==================== */
#include <stdio.h>
void        main()
{
        int     i, j;
        j = 10;             /*  迴圈外先設定初值  */

        for ( i = 0;    i < 6;    )
        {
                printf( "i = %d, ", i    );
                printf( "j = %d \n", j );
                i++;
                j++;
        }
}
```

範例 5:

```
/* ====================
  Program 5 - for
    ==================== */
#include <stdio.h>
void        main()
{
        int     i, j;
        i = 0;          j = 10;                 /*  迴圈外先設定初值  */
        for ( ;    i < 6;    )
        {
                printf( "i = %d, ", i    );
                printf( "j = %d \n", j );

                i++;
                j++;
        }
}
```

範例 6:

```
/* ====================
  Program 6 - for
    ==================== */
#include <stdio.h>
void        main()
{
        int     i, j;

        i = 0;          j = 10;                 /*  迴圈外先設定初值  */
        for ( ; ; )
        {

                if( i >= 6 )        /*  用  if  加上  break */
                        break;   /* break  會立即離開現在的迴圈  */

                printf( "i = %d, ", i    );
                printf( "j = %d \n", j );
                i++;
                j++;
        }
}
```

我們在說明 switch - case 時提到了 break 這個 keyword,
同樣的, break 也適用於所有的迴圈, 包含 for , while, do - while.
等一下會再以幾個範例說明之, 現在先看另一個使用在迴圈控制的指令 continue.

以下範例程式將印出 568:

```
/* ====================
  Program "continue"
    ==================== */
#include <stdio.h>
void        main()
{
        int     i;

        for( i = 0; i < 10; i++ )
        {
                if(    (i !=5) && (i !=6) && (i !=8) )
                {
                        continue;                       /*  忽略以後的  program,  回到  for. */
                }
                printf( "i = %d\n ", i    );
        }
}
```

執行結果:

```
i = 5
i = 6
i = 8
```

在迴圈中, 一碰到 continue 則立即略過底下的程式.

底下程式是九九乘法表的程式, 用了兩層迴圈:

```
/* ====================
 99  乘法.
==================== */
#include <stdio.h>
int        main()
{
           int         x, y;

           for( x = 1; x <= 9; x++ )
           {
                      for( y = 1; y <= 9; y++ )
                      {
                                 printf("%d", x*y );
                      }
                      printf(“\n”);
           }
           return 0;
}
```

執行結果:

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

C 的流程控制四 (**while, do-while**).

讓我們看一下 while 的語法:

```
while( expression )
{
        ...
}
```

如果 expression 的結果是 0, 將結束 while 迴圈.
while 的無窮迴圈寫法如下:

```
while(1)
{
        /* 這是無窮迴圈 */
}
```

以下是一個 while 的範例:

```
/* ====================
  while 的範例 1.
==================== */
#include <stdio.h>
void        main()
{
        int     i, j;
        i = 0;          j = 10;                 /* 迴圈外先設定初值 */
        while( i < 6 )
        {
                printf( "i = %d, ", i    );
                printf( "j = %d \n", j );
                i++;
                j++;
        }
}
```

執行結果:

```
i = 0, j = 10
i = 1, j = 11
i = 2, j = 12
i = 3, j = 13
i = 4, j = 14
i = 5, j = 15
```

同樣的 while 迴圈中也可配合 break 與 continue 使用.

do - while 與 while 非常類似, 只有一個地方不同.
那就是 do - while 迴圈一定會被執行一次以上. 另一種說法是:
while 迴圈的條件檢查是在迴圈最上方, 而 do - while 迴圈是在迴圈最底下檢查.

```
do
{
        ...
}
while( expression )
```

以下是一個 do-while 的範例:

```
/* ====================
  do - while.
==================== */
#include <stdio.h>
void        main()
{
        int     i, j;
        i = 0;          j = 10;                 /*  迴圈外先設定初值  */

        do
        {
                printf( "i = %d, ", i    );
                printf( "j = %d \n", j );
                i++;
                j++;
        }
        while( i < 6 );    /*  檢查條件的地方  */
}
```

執行結果:

```
i = 0, j = 10
i = 1, j = 11
i = 2, j = 12
i = 3, j = 13
i = 4, j = 14
i = 5, j = 15
i = 6, j = 16
```

請觀察以上執行結果跟之前的程式範例有什麼差別.

C 的流程控制五 (**goto**).

goto 會直接跳到所指定的位置去, 但這位置限制在同一函數內.

```
/* ====================
  goto 範例.
==================== */
#include <stdio.h>
int       main()
{
          char       ch;
          ch = getch();
          scanf( "%c", &ch );
          switch( ch )
          {
                    /* 按了 Y/y 會跳到 Yes */
                    case          'Y' :
                    case          'y' :goto YES;
                                       break;
                    /* 按了 N/n 會跳到 NO */
                    case          'n' :
                    case          'N' :goto NO;
                                       break;
                    /* 不是 Yy/Nn 則到 END */
                    default:           goto END;
                                       break;
          }

          YES:;
          printf( "YES");
          return 0;

          NO:;
          printf( "NO");
          return 0;

          END:;
          reutrn 0;
}
```

是否使用 goto 曾引發 computer science 學術界的筆戰, 我的建議是儘量不使用 goto, 如果使用了, 請記得加入大量的程式註解說明.

C 的 Array (陣列).

這一章將說明 C 的陣列宣告及操作, C 的陣列宣告法:

type *variable*[*size*];

在 C 語言中以括號做爲陣列的表示, 先來看第一個例子:

```
/* ====================
  array - 1.
==================== */
#include <stdio.h>
int     main()
{
        int     grade[5];               /* size = 5 的 array */
        int     i;

        grade[0] = 75;          /* 1st element */
        grade[1] = 80;          /* 2nd element */
        grade[2] = 85;          /* 3rd element */
        grade[3] = 70;          /* 4th element */
        grade[4] = 90;          /* 5th element */

        for( i = 0; i < 5; i ++ )
        {
                printf("Number %d = %d\n", i, grade[ i ] );
        }
        return 0;
}
```

程式執行結果:

```
Number 0 = 75
Number 1 = 80
Number 2 = 85
Number 3 = 70
Number 4 = 90
```

C 的陣列是由 0 開始, 因此上例中, 第一個element是 grade[0], 第二個是 grade[1], 依此類推, 共 5 個型別是 int 的資料.

下面兩個程式範例與上面的程式執行結果一樣:

```
/* ====================
  array - 2
==================== */
#include <stdio.h>
int       main()
{
          int       grade[5] = { 75, 80, 85, 70, 90 };

          for( i = 0; i < 5; i ++ )
          {
                    printf("Number %d = %d\n", i, grade[ i ] );
          }
          return 0;
}
```

範例程式是在宣告 grade 時就設定起始值(初值), 設定方式是以大括號括住所有的起始值, 每個起始值以 ”,” 隔開.

再看下面的範例:

```
/* ====================
  array - 3
==================== */
#include <stdio.h>
int       main()
{
          int       grade[] = { 75, 80, 85, 70, 90 };

          for( i = 0; i < 5; i ++ )
          {
                    printf("Number %d = %d\n", i, grade[ i ] );
          }
          return 0;
}
```

上例中, 宣告 grade 時並未給定陣列有多少個element, 但在後面我們給了 5 個初始值, 因此編譯器將會予 grade 爲 5 個elements的陣列大小.

n 維陣列(n-ary Array)的表示法如下:

type *variable*[*size*][*size*][*size*]…[*size*];

共 n 個.

以下是一個 2 維陣列的例子:

```
/* ====================
  2d array.
==================== */
#include <stdio.h>
int       main()
{
          int       array[3][3];
          int       x,y;

          array[0][0] = 1;
          array[0][1] = 2;
          array[0][2] = 3;
          array[1][0] = 4;
          array[1][1] = 5;
          array[1][2] = 6;
          array[2][0] = 7;
          array[2][1] = 8;
          array[2][2] = 9;

          for( x = 0; x < 3; x++ )
                    for( y = 0; y < 3; y++ )
                              printf("%d,", array[x][y] );

          return 0;
}
```

執行結果:

1,2,3,4,5,6,7,8,9,

C 的 String (字串).

C 的基本型態(basic types)中並沒有字串這個基本型態，許多程式語言都提供字串的基本型態宣告,譬如 BASIC, PASCAL.

字串其實就是一堆字元的集合，存放在連續的記憶體空間中，在 C 語言中，字串的結尾是以 0 (NULL) 爲結束.

還記得底下的例子吧:

```
/* ====================
   Say Hello World!.
==================== */
#include <stdio.h>
void          main()
{
           /* 印出 Hello */
          printf("Hello World!");
}
```

“Hello World!” 就是一個字串，字串以雙引號括住，現在我們要更深入討論字串，底下的例子:

```
/* ====================
  String 1
==================== */
#include <stdio.h>
int       main()
{
          char     s[6] = “Hello”;  /* 包含一個字串結束字元，所以是6不是5 */

          printf( “%s”, s );
          return 0;
}
```

執行結果:

```
Hello
```

我們在宣告時, 由於初始值是 “Hello”, 5 個字元, 但要記得加上結束字元之後是 6 個字元, 因此我們宣告 s[6], 而不是 s[5], 編譯器會自動在最後加入 0. 如下:

s 的6個 element.

0	1	2	3	4	5
‘H’	’e’	‘l’	‘l’	‘o’	0

以下兩個範例執行結果同上例:

```
/* ====================
  String 2
==================== */
#include <stdio.h>
int       main()
{
          char      s[] = “Hello”;

          printf( “%s”, s );
          return 0;
}
```

```
/* ====================
  String 3
==================== */
#include <stdio.h>
int       main()
{
          char      s[] = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, 0 };

          printf( “%s”, s );
          return 0;
}
```

還記得在上一章節 C 的 Array 中提到若是不給予 array 大小, 則會依起始值的大小, 編譯器自動配置適當的大小, 所以 s 是一個包含 6 個 char 的 array.
String-2 的範例中, 編譯器會在字串起始值的後面補上 0(NULL),
所以不用像 String-3 的程式
在最後加上一個 0, 因爲 String-3 是以單獨 char element 設定方式, 必需自己補上結束字元, 建議結束字元儘量不要寫 0, 而是寫 NULL, 關於 NULL 在 stdio.h 中會定義到.

以下是改爲 NULL 的程式範例:

```
/* ====================
  String 4
==================== */
#include <stdio.h>
int       main()
{
          char      s[] = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, NULL };

          printf( “%s”, s );
          return 0;
}
```

接著是一個 C 語言的初學者常常出現的問題:

```
/* ====================
  error – 1
==================== */
int       main()
{
          char      s[6];

          s = “Hello”;       /* ERROR!  錯誤  */
          return 0;
}
```

要理解這個錯誤其實很簡單, 再看下面一個錯誤的程式之後, 試著想一想:

```
/* ====================
  error – 2
==================== */
int       main()
{
          char      s[6];

          s = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, 0 };      /* ERROR!  錯誤  */
          return 0;
}
```

以下是正確的解答之一:

```
/* ====================
  string copy - 1
==================== */
int         main()
{
            char      s[6];

            s[0] = 'H';
            s[1] = 'e';
            s[2] = 'l';
            s[3] = 'l';
            s[4] = 'o';
            s[5] = 0;

            return 0;
}
```

原因很簡單, C 的 = (等號運算元) 只針對基本型別運作, 之前已說過 C 的
基本型別中不包含 string 這個 type, 不過若是程式中要設定某個字元陣列豈不是
很麻煩要一個一個去設定?
當然不是, 方法很多, 你可以自己寫一個 string copy 的 function , 或是找找看
ANSI C 定義的 standard C Library 中有沒有符合所需要 function.

自行寫一個 string_copy 的 function 的範例:

```
/* ========================
  自己寫一個 string copy 的 function.
======================== */
#include <stdio.h>
/* ====================
  string_copy function.
==================== */
void    string_copy( char target[], char source[] )
{
        int     i = 0;

        while( source[ i ] != NULL )
        {
                target[ i ] = source[ i ];
                i++;
        }
        target[ i ] = NULL;     /*  最後別忘了加上結束字元  */
}

/* ====================
  main function.
==================== */
int     main()
{
        char    str[6];

        string_copy( str, "Hello" );            /*  測試一下我們寫的 function */
        printf( "%s", str );

        return 0;
}
```

程式執行結果:

```
Hello.
```

上面的例子是爲了示範用，事實上 ANSI C 定第的 standard library (標準函數庫)中，
就有字串複製的函數:

```
/* ====================
  strcpy
==================== */
#include <stdio.h>
#include <string.h>
int       main()
{
          char      str[6];

          strcpy( str, “Hello” );
          printf(“%s”, str );

          return 0;
}
```

程式執行結果:

```
Hello
```

我們用了 strcpy 這個函數，當然要事先定義 function prototype (函數原型)，
而 strcpy 的 function prototype definition 就是放在 string.h 中，
因此我們把 string.h 包含 (include) 進來.

ANSI C 的 standard library 中制定了許多字串處理函數，處理字串時請先查閱相關的
函數庫手冊，避免浪費不必要的時間自己重新開發.

C 的 pointer (指標, 指位器).

C 的 pointer (指標, 另一常譯爲指位器)一般是被認爲 C 語言中最具威力的工具及最難以學習的, 事實上 pointer 並不困難.

第一個範例:

```
/* ====================
  pointer - 1
==================== */
#include <stdio.h>
int       main()
{
          char      *s_pointer = “Hello”;
          char      s[] = “World”;

          printf(“%s\n”, s_pointer );
          printf(“%s”, s );
          return 0;
}
```

程式執行結果:

```
Hello
World
```

先解釋一下這一行:

```
char      *s_pointer = “Hello”;
```

首先編譯器會預留一塊記憶空間存放 Hello, 再把這個空間的 address (也就是 ‘H’ 字元的位址) 設定給 s_pointer.

有點困惑嗎? C 的 pointer 與 array 很類似, 基本上 array 只是一塊連續的記憶空間, 一個變數存放著這個空間的 address (位址), 而 pointer 也只是一個變數存放著一個 address. 若還是不明白, 在以後我們會以更多的範例來說明 pointer 的觀念與使用以及 pointer 與 array 的關係.

底下的例子:

```
/* ====================
  pointer - 2
==================== */
#include <stdio.h>
int       main()
{
          char      *s_pointer;
          char      str[] = “Hello World!”;

          s_pointer = str;
          printf(“%s\n”, s_pointer );

          return 0;
}
```

執行結果:

```
Hello World!
```

上面程式中, 所做的事情不是 string copy, 而是將 pointer 所指的 address 設定(assign)
爲 str 所指的值, 而此時有 str 及 pointer 指向 “Hello World!”.
再看底下的例子:

```
/* ====================
  pointer - 3
==================== */
#include <stdio.h>
int       main()
{
          char      *s_pointer;
          char      str[] = “Hello World!”;

          s_pointer = str;
          s_pointer[0] = ‘h’;         /* ‘H’ 轉 ‘h’ */
          printf(“%s\n”, str );

          return 0;
}
```

執行結果:

```
hello World!
```

前面提過, pointer 與 array 其實只是一個變數存放著一個 address, 在前面的 array 章節中學到的 array 操作自然可以應用(apply)在 pointer 上, 我們把 s_pointer 所指的 address (位址) 上的第一個 element 改成 'h'. 又因爲 str 所指的 address 與 s_pointer 一樣, 所以 printf 輸出 str 爲 "hello World!".

底下的例子:

```
/* ====================
  pointer - 4
==================== */
#include <stdio.h>
int       main()
{
          char      *s_pointer;
          char      str[] = "Hello World!";

          s_pointer = str;
          s_pointer = s_pointer + 1;          /* 也可寫成 s_pointer++ */

          printf("%s\n", str );
          printf("%s\n", s_pointer );
          return 0;
}
```

執行結果:

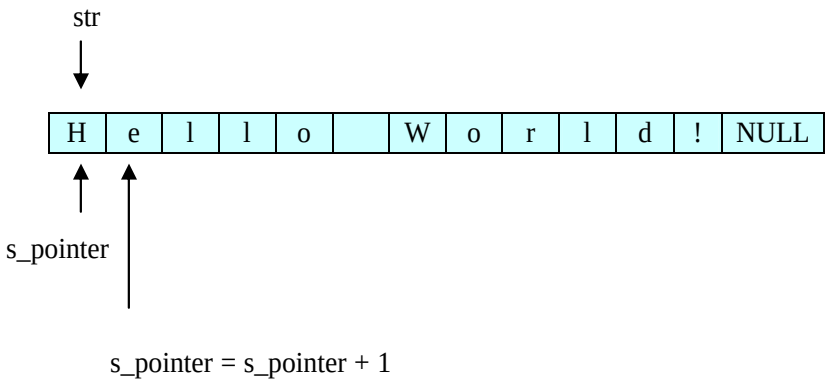
```
Hello World!
ello World!
```

既然我們可以指定 pointer 的 address (位址), 自然就可以將現在所指的 address 往前或往後減少或累加, 上例中我們是累加:

```
s_pointer = s_pointer + 1;
```

這一行程式會將 s_pointer 所指的 address 加一, 因此原本 s_pointer 是指向 'H', 加一之後變成 'e' 開始.

請參考下圖.




```
/* ====================
  pointer - 5
==================== */
#include <stdio.h>
int       main()
{
          char      *s_pointer = "Hello";
          char      ch1, ch2;

          ch1 = *s_pointer;
          ch2 = s_pointer[0];

          printf("%c, %c", ch1, ch2 );

          return 0;
}
```

執行結果:

```
H, H
```

程式示範如何以 pointer 運算取出目前 pointer 所指的 address 上的資料, 我們已知 s_pointer 所指的位址及其上面存放的資料, 以 '*' pointer operator (指標運算元) 可取出所存放的資料, 程式之中比較了等價的 array 操作, 在程式開發上, 個人是偏愛使用 array operator, 但有些 programmer 認爲使用 pointer operator 編譯器產生的 code 較 array operator 快.

以下範例是用 pointer 的間接運算子的例子，當然也可以用 array operator 做到:

```
/* ====================
    pointer – 6
==================== */
#include            <stdio.h>
void    main()
{
        char    *str = "Eric";

        printf( "%c", *(str+0) );          /*  也可寫  printf(“%c”,str[0] ); */
        printf( "%c", *(str+1) );          /*  也可寫  printf(“%c”,str[1] ); */
        printf( "%c", *(str+2) );          /*  也可寫  printf(“%c”,str[2] ); */
        printf( "%c", *(str+3) );          /*  也可寫  printf(“%c”,str[3] ); */
}
```

程式執行結果:

```
Eric
```

看完以上的例子是否更清楚 pointer 的使用，及 pointer 與 array 之間的關係?

在一個比較大的軟體開發計劃中，常常需要動態向系統要求記憶體，以下是一個範例:

```
/* ================
  memory allocation/free 1
================ */
#include <stdio.h>
#include <stdlib.h>
int        main()
{
           char      *str;
           char      s[] = “Hello World!”;
           int       length;

           length = strlen( s ) + 1;    /* 求得 s 長度，加 1 是因爲結束字元. */

           str = (char*)malloc( length );       /* 向O.S 要 memory */

           strcpy( str, s );

           printf(“%s\n”, str );

           str[0] = ‘h’;

           printf(“%s\n”, s );

           free( str );

           return 0;
}
```

程式執行結果:

```
Hello World!
Hello World!
```

strlen 是 C 的 standard library 中求得 string length 的一個 function, 取得 s 的長度之後加 1 是因爲 strlen 並不會把結束字元算進字串的長度. 另外程式中,
以 malloc 這個函數向系統索取一塊記憶空間, malloc 傳回這個空間的 address.

```
str = (char *)malloc( length );
```

其中, (char *)是我們強制轉型，將 malloc傳回的 pointer 型態轉型與 str 一致爲 (char *). 接著再以 strcpy 將 s 裡面的字元 copy 至 str.

程式後面的 str[0] = 'h' 只是爲了證明 str 與 s 所指的 address 及資料是完全不同的. 程式中以 free 這個 function 釋放(release) 之前向系統索取的空間, 所謂有借有還, 維持良好的程式開發習慣.

```
/* ================
  memory allocation/free 2
================ */
#include <stdio.h>
#include <stdlib.h>
int       main()
{
          char      *str;
          char      s[] = "Hello World!";
          int       length;

          length = strlen( s ) + 1;

          str = (char*)malloc( length );       /* 向O.S 要 memory */
          if( str == NULL )
          {
                    printf("Memory allocation failed");
                    return 0;
          }
          strcpy( str, s );
          printf("%s",str );
          free( str );

          return 0;
}
```

執行結果, 記憶空間充足的情況"

Hello World!

或記憶空間不足時:

Memory allocation failed

上面的程式中, 我們多了一個判斷, 判斷 str 是否爲 NULL, 若爲 NULL 則印出 "Memory allocation Failed", 接著離開程式返回系統, 因爲 malloc 在無法取得你所需大小的記憶空間時會傳回 NULL. 這一個判斷是必需的, 特別是在開發大型系統時要處理這些問題.

最後談到的是 & 位址運算元, 這個運算元符號雖然與 Bitwise AND 一樣,
但用法不同, Bitwise AND 的用法是:

Identifier1 & *Identifier2*

而現在說的位址運算元是:

&*Identifier*

本運算元會傳回 *Identifier* 的 Address(位址), 在 C 中, 所有的 variable (變數)
都存在一個實體的記憶體空間中, 當你需要知道這個變數存放的位址時,
就可以使用 "&" operator.

```
/* ====================
 & operator.
==================== */
#include <stdio.h>
int     main()
{
        int     *pointer_a, a;

        pointer_a = &a;
        a = 10;
        printf("%d, %d", a, *pointer_a );

        return 0;
}
```

執行結果:

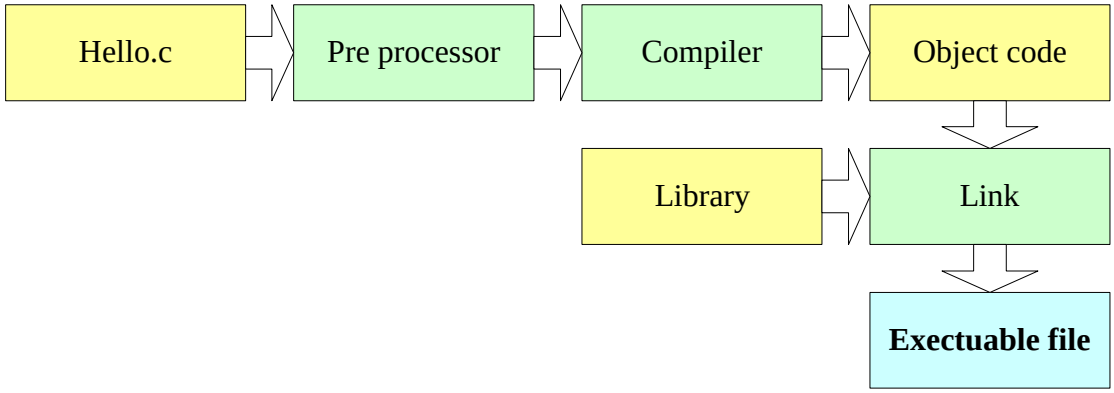
10, 10

程式中, 將 a 的 address 設給 pointer_a (另一說法是: 把 pointer_a 這個 pointer 指向 a),
因此 pointer_a 所"指"的 address 與 a 的 address 一樣, 所以 a, 與使用 *pointer_a 取出
指向的位址上的資料是一樣的.

在前面的"C的輸出與輸入"章節中, 函數 scanf 的參數皆是 pointer, 所以我們以
"&" operator 將變數的 address 傳入.

The Preprocessor (前置處理器).

這一節講 C 的 Preprocessor (或 Pre-Processor, 前置處理器), 最早的 C compiler 並沒有前置處理器, 但後來認爲需要加上一個可以處理 const 或簡易的 語法代換的 macro processor (巨集處理器). 先來看一下之前看過的一張圖:

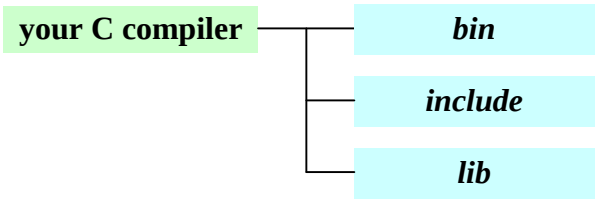


可以知道, 程式在送至 Compiler 產生 Object code (machine code)之前, 會先經過 pre processor 處理, 以前學過的 #include 就是一個 preprocessor 的指令. 在程式中任意一行的第一個非空白字元若是 '#', 表示這是一個 pre-processing directive. (前置處理器處理的指引).

底下是#include 的詳細解說:

#include < *header file* >

Preprocessor 碰到 '<' 與 '>' 括號的檔案, 會搜尋 include path, 這個 path 的設定 依各家 C compiler 不同而有所不同, 一般 C compiler 的目錄通常如下:



bin 中存放的是一些 compiler 相關的可執行檔, lib 存放 C 的 libraries, 而 stdio.h 這種屬於標準函數庫的 header file 是放在 include path 下.

另一種寫法:

#include " *header file* "

則是先搜尋目前 source file(原始檔) 所在的 path 是否存在 header file. 若沒有則再搜尋 include path.

接著介紹的是 #define:

```
/* ====================
 #define 的範例 1.
==================== */
#include<stdio.h>
#define ONE             1
#define TWO             2
#define HELLO                   "hello"
int     main()
{
        printf("%d, %d, %s", ONE, TWO, HELLO );
        return 0;
}
```

執行結果:

1, 2, hello

相信不難理解, #define 的應用場合很多, 譬如 C programming language 沒有定義 TRUE 與 FALSE, 所以很多 programmer 會自行以 #define 定義 TRUE 及 FALSE, 增加程式的易讀性.

以下的程式範例:

```
/* ========================
 #define 的範例 2.
======================== */
#define TRUE            1
#define FALSE           0
#define BOOL            int
/* ====================
 如果傳入值大於 10, 則傳回 TRUE
==================== */
BOOL  is_great_than_10( int i )
{
        if( i > 10 )
                return TRUE;
        else
                return FALSE;
}
/* ====================
 main function.
==================== */
#include<stdio.h>
int     main()
{
        int     i;
        BOOL  result;

        printf("Input a number:");
        scanf("%d", &i );

        result = is_great_than_10( i );

        if( result == TRUE )
                printf("Great than 10!");                   /* 大於 10 */
        else
                printf("Not great than 10!");      /* 不大於 10 */
        return 0;
}
```

由上例, 適當地使用用 #define 可以增加程式的可讀性, 與註解一樣可以幫助理解程式的功能及除錯與維護程式.

條件編譯, #if 與 #else 與 #endif. 關於條件編譯, 是指在編譯之前依設定的條件選擇要編譯的程式, 這項功能的用途常見於 debug, 跨平台條件編譯. 先看底下的範例:

```
/* ========================
  #if, #else, #endif. 1
======================== */
#include <stdio.h>
int     main()
{
        #if     DEBUG
                printf("Debug Version");
        #else
                printf("Release Version");
        #endif

        return 0;
}
```

執行結果:

```
Release Version
```

若在程式前加一行如下:

```
/* ========================
  #if, #else, #endif. 2
======================== */
#include <stdio.h>
#define DEBUG                   1
int     main()
{
        #if     DEBUG
                printf("Debug Version");
        #else
                printf("Release Version");
        #endif

        return 0;
}
```

執行結果爲:

```
Release Version
```

編譯第一個程式前, Preprocessor (前置處理器) 處理時不輸出底下的程式:

```
printf("Debug Version");
```

而只輸出:

```
printf("Release Version");
```

第二個程式中, 因爲:

```
#define DEBUG                   1
```

所以 Preprocessor 處理後輸出的 code 是:

```
printf("Release Version");
```

使用條件編譯時需注意的是, 條件編譯是在 compile (編譯)時決定, 而之前學過的 if – else 則是在程式的執行時期(run-time)決定程式執行的 path (路徑).

在前面的章節中提到可以使用 preprocessor 解決基本型態在不同環境上的大小不一的問題, 底下是一些各廠牌 C compiler 的特性:

Compiler	Target Platform	int	long int
Turbo C 2.0	DOS	2 bytes	4 bytes
MS C 5.1	DOS/Windows 3.1	2 bytes	4 bytes
WATCOM C/C++ 9.5	DOS 32-bit	4 bytes	4 bytes
Symantec C/C++ 7.0	DOS 32-bit	4 bytes	8 bytes

下面的程式請儲存爲 btype.h.

```
/* ========================
  Basic Types Definition.
  File name: btype.h
======================== */
#if     (!BTYPE_H)

        /* ====================
           判斷 !BTYPE_H 是爲了防止因爲重複 #include 造成的 error.
        ==================== */
        #define BTYPE_H     1

        #if     TURBOC20
                #define BYTE        char
                #define WORD        int
                #define DWORD       long
        #endif
        #if     MSC51
                #define BYTE        char
                #define WORD        int
                #define DWORD       long
        #endif
        #if     WATCOMC95
                #define BYTE        char
                #define WORD        short   int
                #define DWORD       int
        #endif
        #if     SYMANTECC70
                #define BYTE        char
                #define WORD        short   int
                #define DWORD       int
        #endif

#endif
```

以下的範例程式需要上面的 btype.h, 另外假設目前所使用的編譯器是 Turbo C 2.0:

```
/* ====================
 basic types test.
==================== */
#define TURBOC20      1
#include “btype.h”
#include <stdio.h>
int        main()
{
           int        wd, dwd;

           wd = sizeof(WORD);
           dwd = sizeof(DWORD);
           printf(“sizeof(WORD) = %d, sizeof(DWORD) is %d”, wd, dwd );
           return 0;
}
```

執行結果:

```
sizeof(WORD) = 2, sizeof(DWORD) is 4
```

若你用的不是 Turbo C 2.0, 而是使用 WATCOM C/C++ 9.5, 則把:

```
#define TURBOC20 1
```

改爲:

```
#define WATCOMC95    1
```

這個小技巧相當常見, 譬如 MS Visual C/C++ 的 Windows SDK 與 MFC 中, 可常見到 DWORD, WORD, BOOL…, 這一些定義, 運用這些定義可以讓你的程式更容易跨越不同或未來的系統.

深入的指標使用 (**Advanced pointer**).

這一章節中將更深入討論 C language 的 pointer.

多重 pointer (pointer to pointer), 一般初學者在此會感到迷惑, 我們用例子解釋.
範例一:

```
/* ====================
  pointer to pointer – 1.
==================== */
#include <stdio.h>
int       main()
{
          char      *Hello = “Hello”;
          char      *World = “World”;
          char      *GoodBye = “Good Bye”;

          char      *StrArray[3];
          char      **Str;
          int       i;

          Str = StrArray;
          StrArray[0] = Hello;
          StrArray[1] = World;
          StrArray[2] = GoodBye;

          for( i = 0; i < 3; i++ )
          {
                    printf(“%s”, StrArray[ i ] );
          }

          for( i = 0; i < 3; i++ )
          {
                    printf(“%s”, *Str );
                    Str++;
          }

          return 0;
}
```

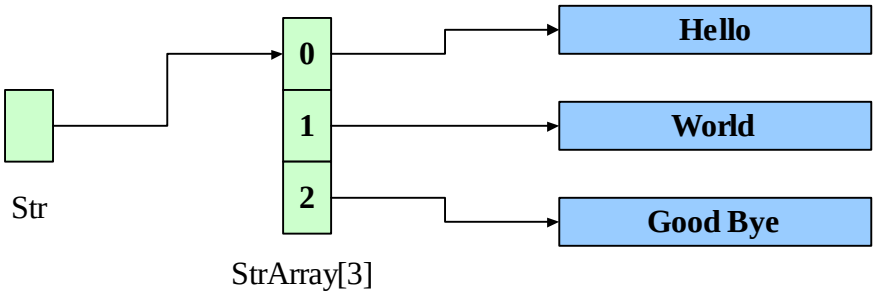
執行結果:

```
Hello
World
Good Bye
Hello
World
Good Bye
```

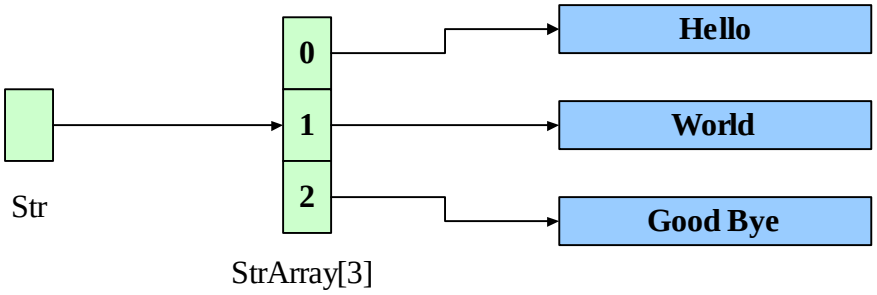
程式中:

```
Str = StrArray;
StrArray[0] = Hello;
StrArray[1] = World;
StrArray[2] = GoodBye;
```

使得:



Str++ 使得:



前面學過了基本的 pointer 使用, 要取出 pointer 目前所指的值可以用 '*' operator.
所以下面一行可以印出目前 Str 所指的值: "World".

```
printf("%s", *Str );
```

另一個常見的例子是 main function 的 argc, argv.
main 函數的參數是程式執行前, 所輸入的參數(argument).

```
/* ====================
  pointer to pointer – 2.
  請將程式存爲 arg.c, 產生的 exectuable file 爲 arg.exe 或 arg.o
==================== */
#include <stdio.h>
int       main( int argc, char **argv )
{
          int       i;
          for( i = 0; i < argc; i++ )
          {
                    printf("%s\n", argv[ i ] );
          }
          return 0;
}
```

執行結果:

若輸入 arg Hello World
後面接了兩個參數, 一個是 Hello, 另一個是 World,
參數與參數之間以空白(space)隔開,
則結果爲:

```
arg
Hello
World
```

我們常見 command line 上輸入的 ls *.conf 或是 DOS 的 xcopy a:\ b:\,
在 C 是傳入 main function. 其中 argc 是參數的數目, argv 是參數字串.
argv[0] 是程式執行檔名稱.

C 的 pointer 除了可以用來存放資料的 address (另一說法是: 用來指向資料),
也可以存放一個函數的 address (另一說法是: 用來指向函數), 怎麼說呢?
先看底下的範例:

```
/* ========================
  function pointer 1
======================== */
#include <stdio.h>
/* ====================
   say "Hello".
==================== */
void     Hello(void)
{
         printf(" Hello ");
}
/* ====================
   main function.
==================== */
int      main()
{
         void      (*func)(void);          /* 宣告一個 function pointer */

         func = Hello;                     /* 把 Hello 的 address 傳給 func */

         func();

         return 0;
}
```

執行結果:

```
Hello
```

程式並不難理解, main function 的第一行是宣告一個 function pointer,
接著把 Hello function 的 address 設給 func pointer, 接著再執行 func.
類似的技巧常被用來做一個 jumping table.

```
/* ========================
  function pointer 2
======================== */
#include <stdio.h>
/* ====================
   say “Hello”.
==================== */
void      Hello(void)
{
          printf(“ Hello\n”);
}
/* ====================
   say “World”.
==================== */
void      World(void)
{
          printf(“ World\n”);
}
/* ====================
   main function.
==================== */
int       main()
{
          void       (*func[3])(void);        /*  宣告一個  function pointer array */
          int        i = 0 ;

          func[0] = Hello;                    /*  建立  Jumping table */
          func[1] = World;

          while(1)
          {
                    printf(“Input a number between 0 and 1 : ”);
                    scanf( “%d”,&i );

                    /*  若  I  大於等於  2  或是小於  0  則離開  loop */
                    if( (i >= 2)||(i < 0) )
                              break;
                    else
                              func[ i ]( );       /*  執行! */
          }
          return 0;
}
```

執行結果:

```
Input a number between 0 and 1 :
```

若輸入 0 則出現:

```
Hello
```

若輸入 1 則出現:

```
World
```

直到輸入的值不是 0 或不是 1.

上例的 table 較小, 只有 2 個 element, 在一些應用程式中通常會用很大的 table,
特別是一些 GUI 相關的程式, 譬如 MS Windows 的 SDK (Software Develop Kit).
GNOME 的 GNU GTK. Linux 與 FreeBSD 的 kernel 也用了很多 function pointer 的技巧.

深入的型態 (Advanced types).

這裡要談的是幾種深入的資料型態, 包括了 struct, union, bit-fields.
先來談 struct. 以下範例:

```
/* ========================
  struct - 1
======================== */
#include <stdio.h>
#include <string.h>
/* ====================
  struct Mouse.
==================== */
struct    Mouse
{
          int       xPos, yPos;
          char      Name[10];
};
/* ====================
  main function.
==================== */
int       main()
{
          struct    Mouse           myMouse;

          myMouse.xPos = 10;
          myMouse.yPos = 20;
          strcpy( myMouse.Name, "Micky" );

          printf("Name:%s,X:%d,Y:%d", myMouse.Name, myMouse.xPos, myMouse.yPos );
          return 0;
}
```

執行結果:

```
Name:Micky,X:10,Y:20
```

struct 是一種 compund data type(複合資料型態), 可包含一個或以上的 basic types(基本型態) 或 compound data types.

要存取(access) struct 的member (成員), 可用 dot "." operaotr (點運算元):

identifier.member

譬如程式中, 存取 myMouse 的 xPos 及 yPos:

```
myMouse.xPos = 10;
myMouse.yPos = 20;
```

struct 也是早期以 C 模擬 Object Oriented Programming Language (OOPL) 的方法之一,
缺點是 struct 內的 member 完全是公開的(public), 沒有權限保護.
再看底下的例子:

```
/* ========================
  struct – 2 (point to a structure)
======================== */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
/* ====================
  struct Mouse.
==================== */
struct    Mouse
{
          int       xPos, yPos;
          char      Name[10];
};
/* ====================
  main function.
==================== */
int       main()
{
          struct    Mouse           *myMouse;

          myMouse = (struct Mouse*)malloc( sizeof( struct Mouse) );

          myMouse->xPos = 10;
          myMouse->yPos = 20;
          strcpy( myMouse->Name, "Micky" );

          printf("Name:%s,X:%d,Y:%d",
                        myMouse->Name, myMouse->xPos, myMouse->yPos );

          free( myMouse );
          return 0;
}
```

執行結果:

```
Name:Micky,X:10,Y:20
```

上例程式中，在執行時期(run-time)配置一塊 memory，大小是 struct Mouse.
在第一個程式中，是用以下的方式存取 member:

identifier.member

現在由於是一個 pointer，因此存取方式是:

identifier->member

例如程式中，存取 xPos 與 yPos:

```
myMouse->xPos = 10;
myMouse->yPos = 20;
```

以 pointer 存取的方式常見於呼叫函數時，傳遞 structure type 的變數，因爲在 C 中無法直接傳遞整個 structure，這很沒效率，若一個 struct 內有 10 個 member，則需要傳 10 個 member，但傳遞一個 pointer 相當於只傳遞一個 member.
譬如我們寫個 function:

```
/* ====================
 Copy Mouse 1
==================== */
void    cpyMouse1( struct Mouse *tgt, struct Mouse *src )
{
        tgt->xPos = src->xPos;
        tgt->yPos = src->yPos;
        strcpy( tgt->Name, src->Name );
}
```

在基本的 pointer 提過，當變數是一個 pointer 時，可以使用 * operator 取得 pointer 所指的 address 上的 data. 所以也可以寫成如下的方式:

```
/* ====================
 Copy Mouse 2
==================== */
void    cpyMouse2 struct Mouse *tgt, struct Mouse *src )
{
        (*tgt).xPos = (*src).xPos;
        (*tgt).yPos = (*src).yPos;
        strcpy( (*tgt).Name, (*src).Name );
}
```

完整的範例程式:

```
/* ========================
  struct – 3.
======================== */
#include <stdio.h>
#include <string.h>
/* ====================
  struct Mouse.
==================== */
struct    Mouse
{
          int       xPos, yPos;
          char      Name[10];
};
/* ====================
  Copy Mouse 1
==================== */
void      cpyMouse1( struct Mouse *tgt, struct Mouse *src )
{
          tgt->xPos = src->xPos;
          tgt->yPos = src->yPos;
          strcpy( tgt->Name, src->Name );
}
/* ====================
  main function.
==================== */
int       main()
{
          struct    Mouse             myMouse,yourMouse;

          yourMouse->xPos = 10;
          yourMouse->yPos = 20;
          strcpy( yourMouse->Name, “Micky” );

          cpyMouse1( &myMouse, &yourMouse );

          printf(“Name:%s,X:%d,Y:%d”, myMouse.Name, myMouse.xPos, myMouse.yPos );

          return 0;
}
```

執行結果:

```
Name:Micky,X:10,Y:20
```

C programming language 雖然提供了 struct, 但是在宣告變數時都要加上 struct, 可以使用 C language 的指令 ”typedef” 定義自訂型態:

typedef *type-information* *type-name*

例如在前例中, 我們可以把 struct Mouse 定義成一個新的型態(type)爲 MOUSE:

typedef struct Mouse MOUSE;

程式範例:

```
/* ========================
  typedef - 1
======================== */
#include <stdio.h>
#include <string.h>
/* ====================
  struct Mouse.
==================== */
struct    Mouse
{
          int       xPos, yPos;
          char      Name[10];
};
/* ====================
   new type: MOUSE
==================== */
typedef           struct Mouse              MOUSE;

/* ====================
  main function.
==================== */
int       main()
{
          MOUSE          myMouse;

          myMouse.xPos = 10;
          myMouse.yPos = 20;
          strcpy( myMouse.Name, “Micky” );

          printf(“Name:%s,X:%d,Y:%d”, myMouse.Name, myMouse.xPos, myMouse.yPos );
          return 0;
}
```

執行結果:

Name:Micky,X:10,Y:20

“typedef” 是一個在以 C 開發大型程式或軟體中常用的一個指令, 用 typedef 配合 struct 適度的定義一些新型態, 可以幫助記憶, 譬如自行定義: BOOL, COMPLEX….

C programming language 中, 有個與 struct 類似的叫 union.
“union” 與 “struct” 最大的不同是, 在 “union” 中, 每個 member (成員) 的起始位置是一樣的, 或是說每個 member 會佔用相同的 storage location(存放位置).

```
/* ====================
  union - 1
==================== */
union    Record1
{
         int       xPos, yPos;
         char      ch;
};
/* ====================
  main function.
==================== */
#include <stdio.h>
int      main()
{
         Record1 R1;

         R1.xPos = 66;

         printf(“xPos=%d, yPos=%d, ch=%c”, R1.xPos, R1.yPos, R1.ch );
         return 0;
}
```

執行結果:

```
xPos=66, yPos=66, ch=B
```

由上可知, 在 union Record1 的所有 member, 其實是存在於相同的一份位置.

同樣的, 也可以使用 typedef 對 union 定義的複合型態加以定義.

Scope (視野).

在程式中, variable (變數) 與 function (函數) 的 scope (視野) 是相當重要的問題,
當一個 programmer 在寫程式時, 他必需知道所用的 variable 與 function 的 scope.

以下程式(scope1.c):

```
/* ====================
 scope1.c
==================== */
int     sum( int x, int y )
{
        return x+y;
}
#include <stdio.h>
int     main()
{
        int     a, b;

        a = 10;
        b = 20;
        printf("%d", sum( a, b ) );
        return 0;
}
```

在程式中, main function 中的 a 與 b 的 scope 就是在 main function 中,
sum function 中, x 與 y 的 scope 就在 sum function 中, 這很容易理解,
再看底下的程式(scope2.c):

```
/* ====================
 scope2.c
==================== */
int     a,b;
int     sum()
{
        return a+b;
}
#include <stdio.h>
int     main()
{
        a = 10;
        b = 20;
        printf("%d", sum( a, b ) );
        return 0;
}
```

在上例中, a 與 b 的 scope 是在整個 scope2.c 中, 也就是可被 main 及 sum "看"到,
所以在 main 中不用傳入 a 與 b, sum 也可看到 a, b 兩個 variables.

在程式中, a 與 b 可說是 global variable, 開發程式時最好避免使用第二種寫法,
因爲會造成維護上的困難, 特別是有很多 function 去存取這些共同的變數時,
你要追查與這個變數相關的 bug 時, 就要一個一個去追查去存取這個變數的 function.
但有時你必需用此寫法, 譬如若你要做一個 function 去偵測目前顯示卡的型號,
並把這個型號以一個 variable 存放, 這個時候因爲顯示卡的硬體是一個整體的資料,
至少在執行這個程式時顯示卡不會被替換. 關於類似的討論與書籍請見最後的推薦資料.

一般放在 function 中的 variable 若不寫明存放方式(storage class), 則一律視爲 auto.
譬如以下程式:

```
int     main()
{
        auto    int     a,b;
        return 0;
}
```

與下面的程式是一樣的:

```
int     main()
{
        int     a,b;
        return 0;
}
```

在以前的程式中, 因爲都是 auto variable, 所以我們都不加上 auto.
但在這一章主要是講解 "scope", 必需提出說明.

當程式越寫越長, 分割成幾個 “.c” 的檔案時, 利用 “extern” 可以存取其它檔案中的 variable 與 function. 但條件是所要引用的 variable 與 function 不是被宣告成 “static”. 首先說明分割成幾個 “.c” 的檔案是如何 compile. 以 gcc 爲例, 若要編譯三個檔案:

```
gcc     file1.c file2.c file3.c –ofile1.o
```

也可以使用 make 這個常見的 utility, 關於 “make” 請參見推薦資料.
底下爲程式範例:

```
/*  請將這個程式存爲  file1.c */
int     sum( int a, int b )
{
        return (a+b);
}


/*  請將這個程式存爲  file2.c */
int     mul( int a, int b )
{
        return (a*b);
}

/*  請將這個程式存爲  file3.c */
extern  int     sum( int, int );
extern  int     mul( int, int );
#include <stdio.h>
int     main()
{
        printf(“6+2 = %d\n”, sum(6,2) );
        printf(“6*2 = %d\n”, mul(6,2) );
        return 0;
}
```

執行結果:

```
6+2 = 8
6*2 = 12
```

在 file3.c 中, 宣告了 sum 與 mul 的函數及其型態, 並以 extern 告訴 compiler 這兩個函數是存在放在別的地方(object file or library or other c source files), 而 sum 與 mul 兩個函數的本體分別存在 file1.c 與 file2.c 中.

你可以試試不把 file1.c 或 file2.c 加入一起 compile 與 link, 會出現的錯誤訊息是 “Link error” 一類的, 並說明找不到 sum 或 mul 函數.

在以前的範例中, 我們曾使用 #include <stdio.h>, 其實 stdio.h 裡面就存放了許多類似函數的宣告, 我們使用 printf 或 scanf 這些函數前, 需要先告訴 compiler, 這些函數是存放在其它的 external file 中.

我們也可以寫一個 .h 檔案來存放我們自己寫的 function, 如下:

```
/* 請將這個程式存爲 myfunc.h */
#if       (!MY_FUNC_H)
#define MY_FUNC_H  1

/* 相加 */
extern   int       sum( int, int );

/* 相乘 */
extern   int       mul( int, int );

#endif
```

再將主程式 file3.c 改寫如下(file4.c):

```
/* 請將這個程式存爲 file4.c */
#include <stdio.h>
#include "myfunc.h"
int       main()
{
          printf("6+2 = %d\n", sum(6,2) );
          printf("6*2 = %d\n", mul(6,2) );
          return 0;
}
```

將 file1.c , file2.c, file4.c 一起 compile and link (編譯與連結). 執行結果:

```
6+2 = 8
6*2 = 12
```

當工作一段時間之後, 會有些常用的函數或是設計些通用的函數,
再將這些函數編譯成 object code 再集合起來建立 library 可節省重複 compile source code 的時間. 建立的方法請參見你所用的 compiler 的使用手冊, 若是 Microsoft 的 Visual C, 可使用 lib.exe 這個 utility, Borland 系統的 compiler 則使用 tlib.exe .

“static” 宣告的 variable 與 function (變數與函數)其 scope 將只在一個 module 中,
(或說只在這個 source file 中). 例如我們在 foo.c 中宣告了一個變數 a, 是 static 的變數.
則 a 的 scope 只在 foo.c 中. 以下程式說明:

```
/* ====================
 將此程式存爲 foo1.c
==================== */
#include<stdio.h>
static    int       foo1 = 100;
static    void      Hello()
{
          printf(“Hello\n”);
}


/* ====================
 將此程式存爲 foo2.c
==================== */
#include <stdio.h>
extern    int       foo1;
int       main()
{
          Hello();
          printf(“%d”, foo1 );
          return 0;
}
```

程式編譯連結會產生錯誤訊息, 大概的意思不外是找不到 “foo1” 或沒有 “foo1” 的存在.
另外的錯誤訊息是找不到 Hello() function 或是沒有 Hello function 的存在.
若把 foo1.c 中的變數 “foo1” 與 Hello function 前的 “static” 刪除則方可編譯通過.

在傳統的結構化程式設計中也有關於資訊隱藏與封裝的觀念, 這並不是 OOP (
Object Oriented Programming) 獨有的觀念, 在 C 中, module 是可用一個 .c 的 file
爲單位, 使用 “static” 隱藏住不想被外界存取的 variable 與 function.

所謂的 module 是一堆相關(related)與互相配合(cooperate)的 variable 及 function 的組合.

最後提到的是 “volatile”, 這個 keyword 一般用來表示 variable (變數) 不經程式寫入, 其值也會改變, 這樣的說法常令人無法瞭解 “volatile” 的用途.
在 embedded system 及 OS 或 device driver 的設計時, 常會用到 “volatile”,

有許多 CPU 的 I/O 設計是採用 memory mapping I/O, 也就是某個 memory address 其實是對應到一個 I/O 上, 假設某個 CPU 的 memory address 200h (h表示16進位) 是某個 button (按紐) 的狀態, 若是 1 則是被按下, 若是 0 則爲一般狀態.

此時你可以宣告變數如下:

```
volatile        int     *Button = (int*)0x200;
```

用 pointer operator 去存取 address 200h 上的資料.

使用 “volatile” 可以防止 compiler 對程式碼進行 optimization (最佳化)後, 使得程式執行結果不是我們想要的. 在一般的應用程式開發上 “volatile” 很少被用到.

推薦書籍

以下書籍可供深入學習及研究.

1.
"**The C Programming Language**", by Brain W Bernighan and Dennis M. Ritchie, Prentice Hall; ISBN: 0131103628.

這本書可以說是 C 語言的規格書, 若在 C 的語法及使用上有疑惑的地方, 本書是最好的解答本.

2.
"**The Mythical Man-Month : Essays on Software Engineering**", by Frederick P., Jr. Brooks, Frederick P. Brooks Jr, Addison-Wesley Pub Co; ISBN: 0201835959.

這本書描述了關於軟體工程的一些觀念, 相當有名的一本書.
作者 Brooks 是 IBM 大型主機的 O.S 發展者.

3.
"**Writing Solid Code : Microsoft's Techniques for Developing Bug-Free C Programs**", by Steve Maguire, Microsoft Press; ISBN: 1556155514.

這本書說明了一些 programming 上的技巧, 包括 debug, anti-bug. 書中的 code 多數是 C 寫的, 熟讀與使用本書的技巧可以減少程式錯誤與學習建構大型程式.

Trademarks and Copyrights

GCC (GNU C Compiler) 是 http://www.gnu.org 的產品.

Symantec C/C++ 7.0 是 Symantec Corp 的產品.

WATCOM C/C++ 9.5 是 Sybase Corp (WATCOM systems 已被 Sybase 併購) 的產品.

MS Visual C/C++ 與 MS C 5.1 是 Microsoft Corp 的產品.

DOS/Windows 3.1 是 Microsoft Corp 的產品.

Turbo C 2.0 是 Inprise Corp (Borland 改名爲 Inprise) 的產品.

OWL 是 Inprise Corp 的產品.