什麼是指標？
指標其實也是一個變數，只是比較特別的是它所儲存的值是 (另一個變數的) 記憶體位址。

和一般的變數比較一下，譬如：**`char a;`** 和 **`int b;`** 分別表示 **`a`** 所儲存的是一個字元而 **`b`** 所儲存的是一個整數。而 **`char *ap;`** 和 **`int *bp;`** 則表示 **`ap`** 是一個指到字元的指標而 **`bp`** 是一個指到整數的指標。

這裡出現了星號 **`*`** operator，就是靠這個符號讓 compiler 知道你需要用到指標變數 **`ap`** 或 **`bp`** 來儲存另外一個變數的記憶體位址。

對於一般的變數我們會用 **`a = '9';`** 或是 **`b = 3;`** 的敘述來指定 **`a`** 和 **`b`** 變數所儲存的值。同樣的我們也可以指定指標變數所儲存的值。但是我們不能像對一般變數一樣隨意指定**`'9'`**或是 **`3`** 這樣的常數值給指標，因為指標變數的值是記憶體位址，所以指標的值必須是真正存在而且有意義的位址才行。

下面是程式裡面常用的兩種設定指標的值的方式，一個是直接複製，把一個已經儲存了「某個記憶體位址」的**指標**的值設給另一個指標，譬如 **`bp = cp;`** (假設 **`cp`** 已經有值)。另一個是用 **`&`** 符號來取得一般變數的位址：**`bp = &b;`** 這樣的動作可以想成是把整數指標 **`bp`** 指到整數 **`b`**，其實就是把整數 **`b`** 的記憶體位址儲存到 **`bp`** 裡頭。

這裡有一個非常基本的前提就是程式中所定義的一般變數，在程式執行的過程中其實會被存放在記憶體的某個位置，所以程式執行過程中記憶體某個位置會被用來代表前面定義的字元變數 **`a`** 和整數 **`b`**，而 **`a = '9';`** 和 **`b = 3;`** 就會把 **`'9'`** 放進代表 **`a`** 的那個位置，而把 **`3`** 放進代表 **`b`** 的那個位置。

一個 **`char`** 型別的變數佔用的記憶體空間是 1 byte，而一個 **`int`** 變數通常是 4 bytes, 那麼一個指標變數需要佔用多少 bytes？
不論是指到哪一種型別的指標 **`char *ap;`** 或 **`int *bp;`** 它們需要的記憶體空間在 32-bit 系統下都只要 4 bytes，因為指標變數所儲存的是是另一個變數的記憶體位址，而 32-bit 系統的記憶體位址只需要 4 bytes 就能表示。(變數需要用到多少記憶體來儲存，可以用 **`sizeof`** 來測試。)

[例外] 唯一可以直接設定給指標的整數值是 **`0`**，但習慣上通常不寫 **`0`** 而是使用 stdio.h 中所定義的 **`NULL`** 常數來表示。由於 **`0`** 這個位址並不會真的被使用到，所以常會用 **`NULL`** 來判斷某個指標是否指到有用的位址，譬如：一開始先設定 **`bp = NULL;`** 經過一連串運算後，你的程式可能需要用 **`if (bp != NULL) ...`** 來判斷 **`bp`** 是否已經被指定了一個有意義的位址。

[練習 WA_01]
請用圖示方式來表示 **`b`** 和 **`bp`** 的關聯。假如 **`b`** 所儲存的值為 **`3`**，而 **`b`** 這個變數的記憶體位址是 **`100000`**。

如何取得指標所指到的記憶體位址中所儲存的值？如何修改指標所指到的記憶體位址中所儲存的值？

這時候就輪到星號 **`*`** 上場了。我們可以在指標變數的前面加上星號來將指標變成一般的變數，譬如 **`*bp`**，不過有一個非常重要的前提：加了 **`*`** 的指標必須已經指向某個記憶體位址，也就是必須已經被初始化 (initialization)。
所以下面這樣的寫法是錯的：

```
int *bp;
*bp = 3;
```

要像下面這樣才行

```
int *bp;
int b;
bp = &b;
*bp = 3;
```

因為 **`*bp`** 相當於把 **`bp`** 所儲存的記憶體位址轉換成那個位址所儲存的變數。如果 **`bp`** 是一個指向整數的指標，則 **`*bp`** 就是 **`bp`** 所指到的整數。所以 **`bp`** 當然必須真的有指到某個整數才行，這樣 **`*bp`** 才有意義。

如果沒有先對指標作初始化就直接想用 **`*`** 將指標轉成它所指到的變數，由於未初始化的指標根本還不知道指到記憶體的哪個位址，所以用了 **`*`** 的下場通常就是程式當掉。

[練習 WA_02]
產生一個 **`float`** 變數 **`x`** 和一個指向 **`float`** 的指標 **`xp`**，先直接把 **`x`** 的值設為 **`3.1`**，然後再透過 **`xp`** 讓 **`x`** 的值增加 **`0.04`**。

為什麼需要指標？

在 C 程式裡面，指標的用途實在太廣了。譬如，指標可以讓我們定出 linked lists 和 trees 等等各式各樣有用的資料結構。就目前所學到的範圍來說，我們可以用指標達到 function 呼叫時 call-by-reference 的效果。

為什麼需要 call-by-reference？

又要舉出已經看過的例子：**swap()**
如果要寫一個 function 可以把傳入的兩個整數變數的值交換，該怎麼寫

```
#include <stdio.h>
void swap(double, double);
int main(void)
{
   double x = 5.5, y = 7.7;
   printf("x=%lf, y=%lf\n", x, y);
   swap(x, y); /* x and y will not be swapped */
   printf("x=%lf, y=%lf\n", x, y);

   return 0;
}
/* useless swap */
void swap(double a, double b)
{
   double tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

上面的寫法當然沒用，因為 **a** 和 **b** 都只屬於 **swap()**，一但程式執行的流程離開了 **swap()**， **a** 和 **b** 就不見了。而且你的主程式呼叫 **swap()** 時只是把 **x** 和 **y** 的值，複製到 **a** 和 **b** 之中。你可以把主程式中的呼叫 **swap(x, y);** 改成 **swap(5, 7);**，在這個例子中這兩種的意思是一樣的，所以可以看出在這種寫法下，**swap()** 所做的事一點意義也沒有。

[練習 WA_03]
試著用 debugger 來 trace 主程式呼叫 **swap()** 的過程，看看局部變數 **a** 和 **b** 什麼時候會存在。

大家應該都已經非常熟悉 **swap()**，知道要改成下面的寫法才對

```
void swap(double *a, double *b)
{
   double tmp;
   tmp = *a;
   *a = *b;
   *b = tmp;
}
```

而呼叫的地方應該改成 **swap(&x, &y);** 因為我們必須把存放 **x** 和 **y** 兩個變數的記憶體位址 **&x** 和 **&y** 告訴 **swap()**，讓 **swap()** 到那兩個位址去找，參數傳進去後相當於做了 **a = &x;** 和 **b = &y;** 的動作。然後用 ***a** 和 ***b** 來設定那兩個記憶體位址中所儲存的值。

[練習 WA_04]
寫一個 function 傳入整數值 **n**，求出 **1+2+...+n** 和 **1*1+2*2+...+n*n** 並把答案分別儲存到主程式中的兩個變數 **m** 和 **s** 中。

指標與陣列的關連？

在 C 程式裏所有可以用陣列方式表示的寫法，都可以改用指標方式來達成。

先來回顧一下陣列。陣列相對於指標來說，算是比較容易理解和使用，因為陣列只是儲存在記憶體中的一串連續元素，譬如 **`int a[10];`** 就是連續十個整數，而 **`a`** 是這一串連續整數的**代稱**，代表的是第一個元素的位址 (第一個元素的 index 是 **`0`**，C 語言中陣列的編號由 **`0`** 開始)。陣列的使用很簡單，**`a[0]`** 代表第一個元素， **`a[9]`** 代表第十個元素，所以我們可以把 **`a[0]`** 或 **`a[9]`**都當作是一般的整數來使用，只不過比較特別的是它們在記憶體中的存放位置是連續的。

既然陣列的元素在記憶體中的存放位置是連續的，那麼如果我們知道第一個元素的位址，不就可以透過指標來找到其他的元素了嗎？想通了這個道理，你的程式就可以藉由指標，更靈活地存取陣列元素。

如何將一個指標指到陣列的第一個元素？方法就和取得一般變數的位址一樣 (譬如 **`bp = &b;`**)：

```
int *pa;
pa = &a[0];
```

另一種寫法是 **`pa = a;`** 因為就像前面說的，**`a`** 這個名稱本身所代表的意義是這個陣列的起始位址。

將指標指到了陣列的開頭位址，再來就要看看如何存取其他元素。在這之前，先再回顧一下指標的位址運算原則。**`(pa+1)`** 所代表的記憶體位址和 **`pa`** 差多少呢？
這和 **`pa`** 這個指標所指到的變數型別有關，由於 **`int *pa;`** 而整數需要 4 bytes 的記憶體來儲存，所以 **`(pa+1)`** 和 **`pa`** 差 4 bytes。

[練習 WA_05]
假如是 **`char *pa;`** 則 **`(pa+1)`** 和 **`pa`** 差多少？

由此衍伸出來，我們得到四種存取陣列元素的寫法，

```
int a[10], *pa;
pa = &a[0];
*pa = 1;
pa[1] = 2;
a[2] = 3;
*(a + 3) = 4;
```

其中第四種用法要稍微注意一下，陣列名稱 **`a`** 代表陣列開頭元素的位址，但是它**並不是一個指標變數**，程式 compile 之後它所代表的相對位址就會確定，譬如在 **`1000`** 這個位址，而 **`&a[2]`** 就變成 **`1008`**，而 **`(a + 3)`** 就變成 **`1012`**。所以 **`a++`** 這樣的動作對陣列來說是不行的；雖然陣列的名稱在大多數情況下用法和指標沒兩樣，但是它終究只是那一連串陣列元素的**代稱** (代表著開頭位址)，而不是一個指標變數，所以不能去改變它的值。(實際上，compiler 看到 **`a[2]`** 這樣的寫法時，會把它翻譯成 **`*(a+2*sizeof(int))`**，然後再把 **`a`** 用位址取代。)

除此此外，在 function 的**參數宣告**中 **`void f(int ary[])`** 和 **`void f(int *ary)`** 陣列與指標這兩種不同寫法是完全相通的。(可能會讓人覺得比較奇怪的是對於 **`int ary[]`** 這種用法，在 **`f()`** 裡頭可以使用 **`ary++`**，因為對 function 來說，參數 **`ary`** 其實是個存放在堆疊記憶體中的變數，不過這已經有些離題了。)

同樣的，呼叫時傳遞位址也可以用兩種方式

```
f(a);
f(&a[0]);
```

甚至如果有需要，不一定要傳陣列的開頭位址，譬如

```
f(a + 2);
f(&a[2]);
```

[練習 WA_06]
寫個小程式測試一下，如果 **`pa = &a[2];`** 那麼 **`pa[-1]`** 或 **`*(pa-2)`** 這樣的用法可以用嗎？

[練習 WA_07]
寫一個 function **`f()`**，傳入兩個各自已經**由小到大排序好**的**正整數**陣列 **`a`** 和 **`b`**，以及一個比 **`a`** 和 **`b`** 合起來還長的空陣列 **`c`**，(假設 **`a`** 和 **`b`** 的最後一個元素都是 **`-1`** 代表陣列結束)，**`f()`** 要將 **`a`** 和 **`b`** 合併並且**從大到小**排好儲存在 **`c`** 裡頭。

```
void f(int a[], int b[], int c[]);
```

字元陣列與字串

關於字元陣列其實只要把前面回顧的整數陣列的 **int** 改成 **char**。而一個**字串**則必須在結尾的地方多加入一個 **'\0'** 字元，表示字串到此結束。其餘的相關用法其實都只是指標和陣列的用法。

[練習 WA_08]
說明並圖示下面兩種寫法的差異：

```
char str1[] = "piece of cake";
char *str2 = "piece of cake"; /* 附註：ANSI C 要求 compiler 可以支援到長度 509 的字串 */
```

[練習 WA_09]
假設在 **main()** 裡面有下列三行，請問 **str1**， **str2**， **str3** 的內容會是什麼？

```
char str1[100];
char str2[100] = {'a'};
char str3[100] = "";
```

[練習 WA_10]
寫一個 function 傳入一個字串，傳回該字串的長度 (整數值)。

[練習 WA_11]
寫一個 function **f()**，傳入一個字串，判斷是否為 palindrome (像是 "level", "wasitacatisaw")。

指標陣列 Pointer Arrays (Pointers to Pointers)

陣列的元素除了可以是整數、浮點數、字元之外，當然也可以是指標。所以我們可以產生一個指標陣列，其中每個元素儲存了某個位址。

[練習 WA_12]
用圖示表現出下面的字串

```
char *ptrary[] = {"piece", "of", "cake"};
```

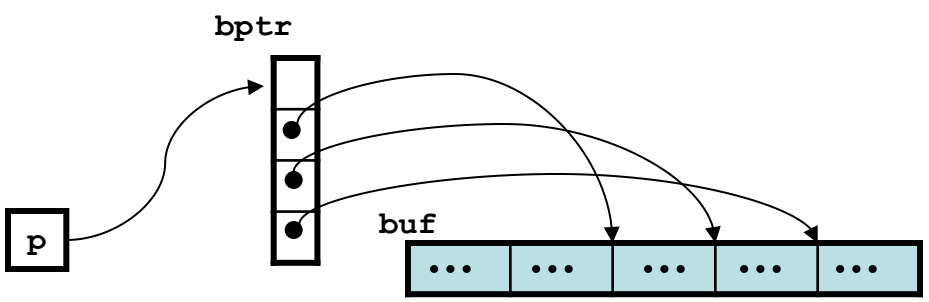
二維陣列 Two-Dimensional Arrays

[練習 WA_13]
圖示下面兩種寫法的差別

```
char *str1[] = {"piece", "of", "cake"};
char str2[][8] = {"piece", "of", "cake"};
```

```
#include <stdio.h>
int main(void)
{
   char *str1[] = {"piece", "of", "cake"};
   char str2[][8] = {"piece", "of", "cake"};
   int i, j;
   for (i=0; i<3; i++) {
      for (j=0; j<8; j++)
         printf("%c", str1[i][j]);
      printf("\n");
   }
   for (i=0; i<3; i++) {
      for (j=0; j<8; j++)
         printf("%c", str2[i][j]);
      printf("\n");
   }
   return 0;
}
```

[練習 WA_14]
二維陣列使用起來很方便，譬如用來儲存矩陣或影像。但是在 C 程式中我們必須預先給定陣列大小，譬如 **int a[10][20];**。這個練習是要藉由**指標陣列**來模擬動態產生二維陣列的機制。
假設程式最前面已經定了兩個 global 陣列：**char buf[1000000];** 是一個字元陣列。**char *bptr[1000];** 是一個字元指標的陣列。
試著寫出一個 function 叫做 **char ** mtx(int m, int n);** 傳入的參數 **m** 和 **n** 代表我們想要產生的二維陣列的大小，回傳值則是一個指向指標的指標。 **mtx()** 要替我們設定 **bptr** 的指標內容 (指到 **buf**)，然後傳回一個指標，讓我們能用例如 **p = mtx(5, 10);** 的方式取得一個二維陣列，然後就可以在程式把 **p** (type 是 **char ****) 當作二維陣列來使用，譬如 **p[2][5] = 7**。在 **mtx()** 裡面應該會包含兩個 **static** 變數，用來記住已經被使用掉的 **buf** 和 **bptr** 有多少。如果 **mtx()** 發現沒有足夠的空間產生二維陣列，就把回傳值設為 **NULL**。



什麼是 C Functions ?

程式是一長串運算的組合，這些運算可以被區分拆解成一些主要步驟或是性質類似的任務，將這些步驟獨立出來就形成 functions。

使用 functions 有不少好處：

1. 程式變得比較清晰易懂，也方便除錯，可以單獨測試每個 function 是否做出正確的運算；
2. 可重複使用自己曾經寫過或別人已經寫好的 functions，這樣能節省許多程式開發的時間。

譬如你的程式裡面需要計算 **a[0]+a[1]+...+a[9]**， **a[10]+a[11]+...a[19]**，和 **b[0]+b[1]+...+b[9]**

```
int i, sum1, sum2, sum3;
for ( i=0, sum1=0; i<10; i++ ) {
   sum1 += a[i];
}
for ( i=10, sum2=0; i<20; i++ ) {
   sum2 += a[i];
}
for ( i=0, sum3=0; i<10; i++ ) {
   sum3 += b[i];
}
```

你會發現兩個迴圈其實非常相似，所以可以寫一個 function 來處理這兩個迴圈要做的事情，然後靠傳入不同的參數來改變運算的內容：

```
int SUM(int array[], int length)
{
   int i, answer;
   for (i=0, answer=0; i<length; answer += array[i++]);
   return answer;
}
```

所以原來的程式碼就變成

```
int sum1, sum2, sum3;
sum1 = SUM(&a[0], 10);
sum2 = SUM(&a[10], 10);
sum3 = SUM(&b[0], 10);
```

每一個步驟做的事情就變得很清楚，而且只要你確定 **SUM()** 的程式是正確的，剩下的就只要注意該傳什麼參數，而不需要再擔心計算過程會出問題。

另外一種常見的情況是使用現成的 libraries 所包含 functions，譬如 C Standard Library 就包含一些常用數學運算或是字串處理所需的 functions，使用時只要在程式最前面用

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

把現成的 functions 宣告引入，就能在接下來的程式碼中使用 **sqrt(x), strcmp(s1, s2)** 等等函式。

另外，自己寫的 functions （像是前面的 **SUM()**），同樣也要在程式最前面加入 function prototype 的宣告

```
int SUM(int a[], int len);
```

讓你的 compiler 先知道有這樣的 function 存在。

在撰寫一個 function 時，除了 function 本身要做的事情之外，還有兩個重點需要考慮，分別是要傳入的參數是什麼，以及要回傳的是什麼？以前面的 **SUM()** 為例，參數有兩個：**int array[]** 和 **int length** 回傳值的型別則是 **int**。

假如 function 沒有回傳值，就要宣告成 **void**，例如 **void f(int a);**，假如連參數也沒有，就變成

```
void f(void);
```

[練習 WB_01]

配合陣列和字串來複習一下。假設用 **char a[100];** 和 **char b[100];** 儲存長度小於一百位數的正整數字串，**char c[200]** 用來儲存計算結果。請下面寫出幾個 functions 做 **a** 和 **b** 的加減乘除然後把答案存到 **c**：

```
void addLong(char c[], char a[], char b[]);
void subtractLong(char c[], char a[], char b[]);
void multiplyLong(char c[], char a[], char b[]);
void divideLong(char c[], char a[], char b[]);
```

什麼是 Recursive Functions ?

自己呼叫自己的 function 就是 recursive function。譬如

```
void f(char *a)
{
   if (*a == '\0') return;
   f(a+1);
   putchar(*a);
}
```

[練習 WB_02]
修改上面的 function， 把輸入字串 **"abc"** 輸出為 **"abccba"**。用圖示來說明程式執行的流程。
[練習 WB_03]
修改練習 WB_02，改成處理中文字串 (一個 big5 中文字要用兩個 characters)。

寫 recursive functions 最需要注意的是停止條件，不然有可能不停呼叫自己，最後把所有可用的記憶體資源都用光。(呼叫 function 時如果傳參數，這些參數需要用記憶體存放；另外，只要 function 還沒執行到 **return**， 那個 function 中的 local variables 也都會持續佔用記憶體，所謂的 stack。所以每多一層 function call，就多用了一些記憶體。)

什麼時候需要 Recursive Functions ?

所有可以用 recursive 方式來寫的程式都可以用 iterative 方式來寫 (**while** 或 **for**)； 反之亦然。既然如此，為什麼不全部用 iteration 就好,幹嘛用 recursion，不但比較耗記憶體，而且執行速度又不會比較快 (因為處理參數也需要額外的時間。)
原因是有些演算法用 recursive 方式來寫比較易懂，也比較簡潔，譬如一些搜尋和排序的演算法。除此之外，有些資料結構，譬如 trees，用 recursive 方式來定義及處理會比較方便。

[練習 WB_04]
重新寫出兩個 functions，用 recursive 方式計算字串長度和判斷字串是否為 palindrome。

在一個 Function 裡面定義的變數和 Function 外定義的變數有什麼差別 ?

依照變數的有效或可建範圍，我們可以把它們劃分成 local variables 和 global variables，定義在 function 裡面的變數只有那個 function 可以看得到，而定義在整個程式最前面，在所有 functions 外面的變數，則是全部的 functions 都看得到。在命名方面，同一個 scope 裡同一個名字當然不能用兩次，譬如

```
int a;
double a;
```

但是在 function 裡的 local variable 可以和 global variable 同名，這種情況 global variable 在那個 function 裡面會被當作不存在。

[練習 WB_05]
下面的程式會顯示什麼？

```
#include <stdio.h>
int aa = 100;
void f(void)
{
   int aa = 3;
   printf("f: %d\n", aa++);
}
int main(void)
{
   printf("main: %d\n", aa++);
   f();
   printf("main: %d\n", aa++);
   f();
   return 0 ;
}
```

上面的例子其實也顯示了變數存在於記憶體的時間長短。一般的 local variables 只有當 function 被呼叫的時候才會自動出現在記憶體 (stack) 中，而當 function 結束後就不見。而 global variables 則是在整個程式執行過程都會一直存在。

如何讓 Function 裡面的變數一直存在？

在變數的型別前面加上 **static**，

```
void f(void)
{
   static int aa = 3;
   printf( "f: %d\n", aa++ );
}
```

你可以用 **static** 變數來記錄 function 被呼叫了幾次。除此之外，如果你想持續記錄某些資料，但是又不想像 global variables 一樣可以被全部的 functions 隨意修改，就可以用 **static** variables，只透過某個特定的 function 來修改。 不過還是無法阻止使用指標方式去修改，譬如

```
#include <stdio.h>
int aa = 100;
int* f(void)
{
   static int aa = 3;

   printf("f: %d\n", aa++);
   return &aa;
}
int main(void)
{
   int *aptr;

   printf("main: %d\n", aa++);
   aptr = f();
   printf("main: %d\n", aa++);
   *aptr = 10;
   f();
   return 0 ;
}
```

[練習 WB_06]
試著解釋一下上面的程式碼的執行結果。如果把上面 **f()** 裡的 **static** 字拿掉會怎樣？

其他 Recursive Functions 範例：計算最大公因數

```
#include <stdio.h>

int gcd(int a, int b); /* gcd() 需要傳回答案 (整數) */

int main(void)
{
   int x, y;
   scanf("%d%d", &x, &y);
   printf("%d\n", gcd(x, y));
   return 0;
}

int gcd(int a, int b)
{
   int ans; /* 制式寫法 */
   if (b > 0) {
      printf("(%d, %d)\n", a, b);
      ans = gcd(b, a%b);
   }  else {
      ans = a;
   }
   /* 如果 b == 0 則 a 就是答案 */
   return ans;
}
```

其他 Recursive Functions 範例：快速計算最大公因數

```
#include <stdio.h>

int bgcd(int a, int b); /* bgcd() 需要傳回答案 (整數) */

int main(void)
{
   int x, y;

   scanf("%d%d", &x, &y); /* 輸入兩個整數 */
   printf("%d\n", bgcd(x, y));

   return 0;
}

/*
   這個遞迴公式的原理不難
   主要是依照 a 和 b 的奇偶數關係
   如果其中一個是奇數另一個是偶數
   則 2 絕對不會是它們的公因數
   因此直接把 2 這個因數去掉
   新的問題變成計算
   bgcd(a/2, b) 或 bgcd(a, b/2)
   這樣問題就變小了
   例如 bgcd(24, 15)
   其實只要算 bgcd(12, 15) 就可以 因為不會有 2 這個公因數
*/
int bgcd(int a, int b)
{
   int ans; /* 制式寫法 */
   int tmp;

   if (a<b) {  /* 如果 a 比 b 小就調換 */
      tmp = a; /* 確保 a 總是大於等於 b */
      a = b;
      b = tmp;
   }

   /* 兩種停止條件 */
   if (b==1)
      return 1;

   if (b==0)
      return a;

   if (a%2 == 0) { /* 如果 a 是偶數 */

      if (b%2 == 0) { /* 如果 b 是偶數 */
         ans = 2 * bgcd(a/2, b/2);
      } else {        /* b 是奇數 */
         ans = bgcd(a/2, b);
      }

   }  else { /* a 是奇數 */

      if (b%2 == 0) { /* 如果 b 是偶數 */
         ans = bgcd(a, b/2);
      } else {        /* b 是奇數 */
         ans = bgcd((a-b)/2, b);
      }

   }

   return ans; /* 制式寫法 */
}
```

其他 Recursive Functions 範例：遞迴方式計算平方和

```
#include <stdio.h>
int ssum(int a);

int main(void)
{
   int x;
   scanf("%d", &x);
   printf("=%d\n", ssum(x));
   return 0;
}

/* 遞迴公式: ssum(a) = a*a + ssum(a-1) */
/* 同時還會顯示 a*a+(a-1)*(a-1)+...+1*1 */
int ssum(int a)
{
   int ans; /* 制式寫法 */

   if (a>1) {
      printf("%d*%d+", a, a); /* 除了計算 ans 之外 還顯示一些計算過程 */
      ans = a*a + ssum(a-1);
   }
   else {
      printf("1*1");
      ans = 1;
   }
   return ans; /* 制式寫法 */
}
```

其他 Recursive Functions 範例：顯示整數的二進位表示法

```
#include <stdio.h>
void i2b(int n);
void i2b_wrong(int n);

int main(void)
{
   int x;
   scanf("%d", &x);
   i2b(x); /* 呼叫 i2b() 把 x 的二進位表示法顯示出來 */
   printf("\n");
   i2b_wrong(x); /* 錯誤的寫法 */
   printf("\n");
   return 0;
}

/* 關鍵在於 遞迴呼叫的時候
   所有在 遞迴式 之前的程式碼 都會以呼叫的順序來進行
   但是在 遞迴式 之後的程式碼 會以呼叫的相反順序進行 */
void i2b(int n)
{
   if (n > 0) { /* 如果 n > 0 就把 n 除以 2 的餘數顯示出來 */
      i2b(n/2); /* 然後把問題化簡成為 顯示 n/2 的二進位表示法 */
      printf("%d", n%2); /* 把最低位的 bit 顯示出來 */
   }
   /* 如果 n == 0 就不用再遞迴下去 */
   /* 什麼事都不用做了 */
}

void i2b_wrong(int n) /* 錯誤的寫法 */
{
   if (n > 0) {
      printf("%d", n%2);
      i2b_wrong(n/2);
   }
}
```

我們用一個簡短的範例程式示範如何開檔和讀檔。範例程式會從一個純文字檔案 5000_words.txt 裡讀取資料，請從課程網頁 (參考書籍與連結的補充資料)，下載 5000_words.txt，檔案裡面包含了五千個英文單字：

5000_words.txt 檔案片段
...
procedure n. A manner or method of acting.
proceed v. To renew motion or action, as after rest or interruption.
proclamation n. Any announcement made in a public manner.
procrastinate v. To put off till tomorrow or till a future time.
procrastination n. Delay.
…

程式一開始先宣告一個型別是 **FILE** 的指標 **fp**，然後用 **fopen()** 開啟要讀取的純文字檔，並把取得的檔案指標用變數 **fp** 記下來。有了這個指標，就可以用 **fscanf()** 或 **fgetc()** 循序讀取檔案內容。程式裡面還用了 **feof()** 來判斷是否已讀到檔案結尾，最後當檔案使用完畢則要用 **fclose()** 來關閉檔案。除了這個範例使用的幾個 functions 之外，還有幾個常用的與檔案有關的 functions，例如 **fprintf()**、**fread()**、**fwrite()**、**fseek()**，詳細的用法可以查詢使用手冊 (課程網頁上有 The GNU C Library Reference Manual 的 PDF 檔的連結) 。

```
#include <stdio.h>
#include <stdlib.h>    /* 為了用 srand() 和 rand() */
#include <time.h>      /* 為了用 time() 產生亂數種子 */
#define LINE_LEN 40    /* 假設單字長度不超過 40 個字元 */
#define NUM_WORDS 5000 /* 總共有 5000 個單字 */
char dictionary[NUM_WORDS][LINE_LEN]; /* 儲存單字的陣列 每個 row 都是一個字串 儲存一個單字 */

int main(void)
{
   FILE *fp; /* 開檔案 需要一個指標指到開啟的檔案 */
   int nw;
   int i;

   /* fopen() 的用法就是傳入檔名字串 以及開檔方式  "r" 表示只要讀檔  */
   /* fopen() 會傳回指到檔案的指標  失敗的或那個指標會是 NULL 表示沒有指到任何地方 */
   if ( (fp = fopen("5000_words.txt", "r")) == NULL ) {
      printf("Cannot open file.\n");
      exit(1);
   }
   /*
      用 fscanf() 把所有的單字都讀進來
      fscanf() 和 scanf() 的用法相同
      只是要多傳一個參數  把檔案指標傳給 fscanf()
      才能夠知道是要從哪個檔案讀資料
      feof(fp) 可以判斷檔案 fp 是否已經讀到了盡頭
      如果已經到了檔案結尾 feof(fp) 會傳回 true
      fgetc(fp) 則只會讀取一個字元  我們只想讀取單字
      所以用 fgetc(fp) 把剩下的單字解釋略過 一直讀到換行  再繼續讀下個單字
   */
   nw = 0;
   while (!feof(fp) && nw < NUM_WORDS) {
      fscanf(fp, "%s", dictionary[nw]);
      while ( !feof(fp) && fgetc(fp) != '\n' ) ;
      nw++;
   }
   fclose(fp);
   /*
      隨機選 10 個單字出來
      dictionary[][] 的每一個 row 都是一個字串
      例如 dictionary[1002] 儲存的是 "convertible" 這個字串
      所以 dictionary[1002][0] 就是 'c'  而 dictionary[1002][1] 就是 'o'
      以此類推 dictionary[1002][10] 是 'e' 然後 dictionary[1002][11] 是 '\0'
   */
   srand(time(NULL));
   for (i=0; i<10; i++) {
      printf("%s\n", dictionary[rand()%NUM_WORDS]);
   }
   return 0;
}
```