第三單元
C 語言基礎架構

本單元將介紹 KEIL C 編譯軟體模式下之 C++程式語言架構，由最基礎的指令宣告開始講解，循序介紹資料型態、運算子、迴圈及函式，最後進階講解結構化的 C++程式設計。

其實，在各式編譯 C++或 C#語言的軟體架構下，均依循著最典型的 C 語言的資料型態來進行演化，當使用某部份指令不明確時，只要依循著典型的 C 語言格式幾乎都可以完成編譯的動作，因為高階語言 C++與 C# 的演化，其實是為了簡化繁雜而且規定嚴謹的典型 C 語言，讓程式語言寫作得以更加彈性化與簡單化。而本單元介紹的 C/C++資料格式與語法多數通用於其它編譯系統軟體，部分為 KEIL C 編譯器所特有的保留字及宣告方式已特別註解，至於物件導向程式設計的 class 架構並不適用於 KEIL C 編譯軟體，且多數微處理器或韌體系統開發的編譯軟體，並不需要如此複雜的程式開發，所以，本單元已將物件導向 class 與以省略。

3.1 程式語言的基礎架構

3.1.1 程式語言的目的與架構

什麼是程式語言 ? 在微電腦的世界裡是一長串的 0 與 1 的位元碼，與人類語言格式的描述方式是完全不同的類型，而程式語言開發的目的，即為了將我們所需望表達的資料訊息，透過編譯軟體轉換為微電腦所能夠判讀的資料格式，最終得以依照開發者的構思與邏輯時序循序動作，執行相關的功能與運作。

本書使用的是 keil uVision3 軟體為程式語言編譯器,在 ATMAL 公司所開發的晶片裡,編號規格為 AT87、AT89、AT91、T80、T83、T87、T89 等相關列單晶片均可使用，既然稱為"語言"則必定有相關的格式與文法存在，以下使用簡單的範例來介紹 C 語言程式架構的開端。

```
#include    <AT89x51.h>                 //編譯的程式包含 reg51.h 所引述的資料
#define     count   100                 //定義 count 名稱的值為十進位 100
unsigned    char   x;                   //宣告程式存在一個字節且無正負符號的全域變數 x
void   delay   (unsigned   int);        //宣告在主函式後存在著 delay 的副函式
void   main (void)                      //主函式
{  unsigned    char   z;                //宣告函式存在一個字節且無正負符號的區域變數 z
   x=0xFF;                              //設定 x 變數初值為 0xFF
   P1=x;                                //致能 PORT1 為輸出，初值為 0xFF
   while(1)                             //無限迴圈
   {   for ( z=0 ; z<10 ; z++ )         //執行 10 次的 for 迴圈
       {   P1=~ P1;                     // P1 反向
           delay (count);               //時間延遲，將常數 100 引入延遲副程式中
}  }  }
void   delay   (unsigned   int   j)     // 延遲副函式
{  unsigned   int   i = 0;              //宣告函式存在一個整數且無正負符號的區域變數 i
   do                                   //do… while 前測迴圈
    {   i++; }                          // i = i+1
   while ( i < j );                     //當 i<j 條件成立時繼續執行迴圈
}
```

在上述的程式語言格式裡，引述了一個寫程式語言的重點，當完成所有的定義及宣告後，所有的程式開始執行必須要有一個主函式作為程式的開端，而 main()就是是所以程式執行不可或缺的主函式，幾乎所有的 C++資料格式都是引入這個主函式不變，所以必須記得，當一個程式語言開始執行命令或運算指令，都必須是從主函式開始與結束。

而在撰寫程式語言有幾點是必須注意的:

※只要是函式內必須使用的變數，均須於函式敘述的開端宣告，如上述程式的 z 與 i，否則編譯器將視為錯誤命令格式不予執行。

※C 語言的指令是有區分大小寫的，且在宣告變數或定義名稱時，都必須注意不可與保留字使用相同字彙，而且在做變數或函式宣告時記得第一個字不可以為數字。

3.1.2 二進位資料格式

既然我們所習慣的數據資料格式是不被微電腦所能接受的，所以我們需要瞭解微電腦真正所能夠判讀的數據型態及轉換的方式，而微電腦的控制其實是依循著二進位碼來進行數據的解讀與運算，當存在同一位置且為一連續的暫存器為一次的運算單位計數時，我們

便稱這個單位容量大小為該微電腦晶片的運算位元尺寸,目前市面上常見的有 8bit、16bit、32bit、64bit 等四種規格,而 AT89S51 則是屬於 8bit 位元尺寸的單晶片,以下表列為 8bit、16bit、32bit 等三種格式的最小及最大數據容量及表示方式:

bit	二進位	十進位	十六進位
8	0b00000000~0b11111111	0~255	0x00~0xFF
16	0b0000000000000000~ 0b1111111111111111	0~65535	0x0000~0xFFFF
32	0b00000000000000000000000000000000~ 0b11111111111111111111111111111111	0~4294967295	0x00000000~0xFFFFFFFF

在撰寫程式時須注意的格式問題，一般的十進位是可以直接撰寫數字，但撰寫十六進位時必須在數字前加上「0x」，編譯器才得以辨認我們所宣告的資料格式，而使用任何一種單晶片亦或是電腦的程式語言系統，均需要了解它的位元空間容量及資料格式型態，才能夠有效的運用並且避免資料運算過程溢位造成錯誤，詳細具體的運算方式與技巧請參閱「數位邏輯設計」相關應用書籍。

3.1.3 keil C 語言的保留字

何謂保留字 ？ 就是編譯器將這些字節保留作為其它用途，如命令、宣告、主函式名稱、迴圈、敘述或定義型態等等，通常編譯器會將這些保留字節在使用過程中已不同的顏色或字型保留，但使用者仍須注意絕對不可與保留字節重複使用，即使經過定義該字節已被其它字節所取代，這些保留字字節仍具有其特殊意義無法使用。

3.1.3.1「American National Standards Institute，ANSI」美國國家標準協會所制定的保留字，適用於所有 C 語言的編譯系統使用。

asm	auto	break	case	char	const
continue	default	do	double	else	entry
Enum	extern	float	for	fortran	goto
int	long	register	return	short	signed
Sizeof	static	struct	switch	typedef	union
undigned	void	volatile	while		

3.1.3.2 keil C 編譯系統自訂保留字，適用於 keil uVision 版本編譯系統使用。

_at_	_priority_	_task_	alien	bdata	bit
code	compact	data	far	idata	interrupt
large	pdata	reentrant	sbit	sfr	sfr16
small	using	xdata			

3.2 基本敘述符號與前置作業指令

3.2.1 註解、敘述、分號與無效指令

在 C 語言編譯過程中，有某些符號並非代表運算或判斷指令，但卻是每一段編一過程中不可或缺的符號格式，符號型態及用途如下表所列:

3.2.1.1 // (單行註解)

在符號之後所有字元均不予編譯，如下列範例中所示「宣告無號數字元變數 x」這些字元無論中英文均不被編譯，跨行後則不在註解範圍內。

```
unsigned  char  x ; //宣告無號數字元變數 x
```

3.2.1.2 /**/ (整段註解)

在符號/*之後及*/之前所有字元均不予編譯，如下列範例中所示「宣告無號數字元變數 x」這些字元無論中英文均不被編譯。

```
unsigned char x ;
/*宣告無號數字元變數 x
*/
```

3.2.1.3 {} (敘述)

幾乎所有函式與迴圈之後均須加上敘述符號，而敘述符號內即可填入函式或迴圈所必須執行的算式或命令，除部份迴圈如 if、for 迴圈之後如命令或算式僅有一行，則可以省略敘述迴圈。

void main (void) { ……; }	//函式宣告 //所有運算或執行命令均置於敘述括弧內
while(1) { ……; }	//while 判斷式 //所有運算或執行命令均置於敘述括弧內
if(x!=1) 算式或命令;	//if 判斷式 //僅一行命令，可免敘述括弧執行

3.2.1.4 ; (分號)

除了前置作業處理指令的所有命令及迴圈、函式之外，每一行算式或命令之結尾都應加上分號，代表本行到此結束，但迴圈命令中 do..while 後條件迴圈則例外。

void main (void) { ……; }	//函式內的每一段算式或命令結尾都需加上分號「;」
do {……; } while() ;	//函式內的每一段算式或命令結尾都需加上分號「;」 //本行結尾必須加上分號符號，否則編譯器將視為錯誤命令格式

3.2.1.5 void (無效)

使用在每一段函式之前的回傳功能與之後()符號中的引入功能，如無需回傳資料或是無須引入資料或位置則須加上 void 使功能失效，但是在 keil C 軟體中主函式 main()的回傳與引入功能及一般函式的引入功能，可以不輸入 void 符號且編譯器自動視為無效功能編譯。

main () { ……; }	//主函式的回傳與引入括弧均自動視為無效功能 //命令或運算式
void main (void) { ……;}	//主函式的回傳與引入括弧宣告為無效功能 //命令或運算式

```
void  delay  ()          //副函式的引入括弧自動視為無效功能
{ ……;}                   //命令或運算式
```

3.2.2 前置作業處理指令

在 C++的語言格式中當出現了「#」符號命令時，代表編譯程式時必須由#符號所表述的該行指令開始編譯起，因為這些指令在開始編譯前已被前置處理器(preprocessor)置換成某些程式碼，所以前置作業處理指令又被稱為假指令，以下而列舉#符號的使用命令類型說明。

3.2.2.1 # define (巨集)

以一個巨集名稱來代表一個字串，而巨集的定義可以是一個常數、算式、含有引數的算式或含有已宣告的變數算式皆可，在編譯時編譯器會將所以的巨集名稱代換成字串後再進行程式編譯，使用巨集會佔用較多的記憶體空間，但執行速度會比較快。

define 巨集名稱 字串

```
# define  count   100                  //定義常數
# define  data    idata                //定義字串
# define  uchar   unsigned char        //定義型態
# define  clock   ((65536-5000)*10)    //定義算式
# define  clock   (0xAA&0xBB)          //定義位元邏輯算式
# define  DATA(A)  (100*(A+5))         //含引數算式
```

```
unsigned  char  A;
# define  DATA  ((A&0xFF)!=0)          //含變數算式
```

3.2.2.2 # include (含括)

含括指令是將指定的檔案含括道主程式內，含括的檔案稱之為含括檔(include file)，如果含括的檔案內存放的是常數定義、巨集定義，則此含括檔案稱之為標頭檔(header file)，而使用含括檔或標頭檔的意義在於可將不同定義或通用的副程式在外部撰寫後，直接引入程式內使用，而這個含括檔或標頭檔通常可適用於其他的程式使用，如此可簡化每次程式撰寫便需要重新編寫副程式或定義資料的繁複手續。而範例中的<AT89S51.h>檔案為軟體安裝同時已存在且已指定存取路徑之標頭檔，“delay.h”則為外部撰寫的延遲副程式檔案

```
# include  <檔名>    或    # include  “檔名”
# include  <AT89S51.h>  //系統指定檔案路徑
# include  “delay.h”    //同一檔案存取路徑
```

3.2.2.3 # if , # else , # endif (條件式編譯)

if 條件式的編譯指令就是符合條件時才執行編譯，若不符合條件則略過或執行# else，因為編譯器並未編譯已略過的程式區段，所以並不佔用記憶體空間。

```
# if  條件式              //如果 if 條件式成立則執行以下命令
   定義資料型態、宣告副函式型態或命令;
# else                    //當條件式不成立時則執行以下命令
   定義資料型態、宣告副函式型態或命令;
# endif                   //結束條件式編譯
```

```
# if  1                   //條件永遠成立
  P1=0xFF;                //若 if 條件成立執行本區段，P1 輸出為 1
# endif                   //結束條件式編譯
```

```
# if   0                    //條件永遠不成立
   P1=0xFF;                 //若 if 條件成立執行本區段，P1 輸出為 1
# else                     // if 條件不成立，執行以下區段
   P1=P1&0xFF;              //保留 P1 為 1 的資料
# endif                    //結束條件式編譯
```

```
# if   DATA                 //如果 DATA 資料未定義則為 0
   P1=0xFF;                 //若 if 條件成立執行本區段，P1 輸出為 1
# else                     // if 條件不成立，執行以下區段
   A[0]=P1&0xFF;            //陣列 A[ ]第 1 格= P1 為 1 的資料
# endif                    //結束條件式編譯
```

3.2.2.4 # ifdef , # ifndef , # endif (條件式編譯)

該條件式的編譯指令通常用來除錯，或撰寫外部的副程式後編撰標頭檔與主程式連結使用，又或者有兩種狀況而須做不同的程式編寫時使用。

```
#ifndef  巨集名稱  //如果巨集名稱未被定義過，則編譯以下程式
  定義資料型態、宣告副函式型態或命令
#endif                    //結束條件式編譯
```

```
# ifdef    _DELAY_H              //如果 DELAY 巨集名稱已經被定義過
# define   _DELAY_H              //定義_DELAY_H 標頭檔
# define   delay_1   20          //定義 delay_1 字節相等於常數 20
# define   delay_2   (a&b)       //定義 delay_2 字節為(a&b)運算結果
   void   delay (unsigned   int) //定義存在 delay 副程式且具有 int 引數值
# endif                          //結束條件式編譯
```

```
# ifndef    _DELAY_H             //如果 DELAY 巨集名稱沒有被定義過
# define   _DELAY_H              //定義_DELAY_H 標頭檔
# define   delay_1   20          //定義 delay_1 字節相等於常數 20
# define   delay_2   (a&b)       //定義 delay_2 字節為(a&b)運算結果
   void   delay (unsigned   int) //定義存在 delay 副程式且具有 int 引數值
# endif                          //結束條件式編譯
```

3.3 變數宣告與修飾字

3.3.1 變數的等級與視野

C 語言提供了許多不同等級的變數並以不同的方式來存放使用，而不同等級的變數也造成了不一樣的生命週期及視野，共計區分為全域變數、區域變數、靜態變數及暫存器變數，其中，所謂的生命週期就是指變數存在的時間，而視野則是指變數涵蓋的使用範圍，以下將逐一解釋並例舉說明。

3.3.1.1 全域變數(Global)

是定義於函式外部的變數，其生命週期會存在於整個程式執行的過程直到結束(或系統停機)，而視野含擴於所有的函式，亦即在視野範圍內之所有程式均可存取，而不在視野環境內的亦可透過外部宣告(extern)來加以使用，但，全域變數將在系統運作期間保留記憶體空間並不釋放，而且同一程式含引入程式中不可以重複變數名稱。以下範例中，全域變數 x 同時被主函式及副函式使用，其值從宣告為 100 並累加 101~150 後結束，除非系統停止，否則 x 將持續保持最後運算數據不會消失。

```
#include    <reg51.h>              //編譯的程式包含 reg51.h 所引述的資料
unsigned    char   x;              //宣告程式存在一個字節且無正負符號的全域變數 x
void   delay   (unsigned   int);   //宣告在主程式後存在著 delay 的副程式
void   main (void)                 //主函式
{   x=100;                         //x 初值=100
    delay(50);                     //區域變數中 x 累加從 101~150
}
void   delay   (unsigned   int   j)   // 延遲副函式
{   unsigned   int   i = 0;        //宣告一個區域變數 i 但名稱不可與全域變數相同
    do   {   i++;   x++; }         // i = i+1   ,   x = x+1
    while ( i < j );               //當 i<j 條件成立時繼續執行迴圈
}
```

3.3.1.2 區域變數(Local)

是定義於函式內部的變數，其生命週期僅存在於函式中，隨著函式的宣告而產生直到函式的結束也隨之消失，其數據也隨著消失，並同時釋放其佔有的記憶體空間，而其視野僅限制於宣告產生的函式內，其它函式並不可以使用，所以，即使其它函式以相同的名稱命名區域變數也不會互相影響，雖然變數將隨著函式結束而消失，但是其數據仍可透過回傳(return)而給予其它函式銜接使用，其使用技巧將在章節函式中詳述說明。在下列範例中，主函式及副函式均宣告了區域變數 i ，但變數 i 於函式執行結束時已釋放記憶體空間，並不相互影響。

```
#include    <reg51.h>              //編譯的程式包含 reg51.h 所引述的資料
void   delay   (unsigned   int);   //宣告在主程式後存在著 delay 的副程式
void   main (void)                 //主函式
{   unsigned   int   i;            //宣告一個區域變數 i
    for( i=0 ; i<50 ; i++)
      delay(50);
}
void   delay   (unsigned   int   j)   // 延遲副函式
{   unsigned   int   i = 0;        //宣告一個區域變數 i 名稱可與主程式相同
   do   {   i++;   x++; }          // i = i+1   ,   x=x+1
   while ( i < j );                //當 i<j 條件成立時繼續執行迴圈
```

```
}
```

3.3.1.3 靜態變數(static)

區分為函式內部定義及外部定義兩種:

●**內部定義**：其視野與區域變數相同，除宣告的函式以外的其它函式是不可以使用的，但是生命週期與全域變數相同，而是會保留記憶體位置及數據，等待下一次呼叫時使用，除非系統停止，否則資料將不會消失。在下列範例中，靜態變數 i 在 delay 函式執行完後仍保持著最後運算數據 50，保留至下一次呼叫函式使用。

static 修飾字 變數資料型態 變數名稱;

```
#include   <reg51.h>                  //編譯的程式包含 reg51.h 所引述的資料
void  delay  (unsigned  int);         //宣告在主程式後存在著 delay 的副程式
void  main (void)                     //主函式
{  delay(50);                         //命令或運算式
}
void  delay  (unsigned  int  j )      // 延遲副函式
{  static  unsigned  int  i = 0;      //宣告程式存在一個整數且無正負符號的靜態變數 i
   do  {  i++;  x++; }                // i = i+1 ， x=x+1
while ( i < j );                      //當 i<j 條件成立時繼續執行迴圈
}
```

●**外部定義**：其視野與全域變數相同，但僅侷限於宣告的檔案內使用，義即外部定義的檔案即使宣告名稱相同的靜態變數亦不互相影響，但是生命週期仍與全域變數相同，而是會保留記憶體位置及數據，等待下一次呼叫時使用，除非系統停止，否則資料將不會消失。在下列範例中，靜態變數 i 在主程式 main()被設定初始值為 10，並在 delay 函式第一次執行完後仍保持著最後運算數據 50，保留至下一次呼叫函式使用。

static 修飾字 變數資料型態 變數名稱;

```
#include   <reg51.h>                  //編譯的程式包含 reg51.h 所引述的資料
static  unsigned  int  i = 0;         //宣告程式存在一個整數且無正負符號的靜態變數 i
void  delay  (unsigned  int);         //宣告在主程式後存在著 delay 的副程式
void  main (void)                     //主函式
{  i=10;                              //設定靜態變數初值為 10
  delay(50);                          //命令或運算式
}
void  delay  (unsigned  int  j)       // 延遲副函式
{  do  {  i++;  x++; }                // i = i+1 ， x=x+1
while ( i < j );                      //當 i<j 條件成立時繼續執行迴圈
}
```

3.3.1.4 外部變數(external)

如果想要跨檔案存取其它檔案的全域變數，則必須要在全域變數加上外部變數的宣告語法，所以，當變數宣告加上「**extern**」關鍵字之後，編譯器就會知道該變數已經在別的地方宣告過了，或是在之後的其他地方宣告，而無需在此保留而外的記憶體空間給該變數。

external 修飾字 變數資料型態 變數名稱;

而外部變數宣告也可以是外部副函式型態，適用語法格式如下。

external 回傳值 函式名稱 (引入值);

```
/*以下是主程式的主函式，呼叫外部程式*/
#include   <reg51.h>                  //編譯的程式包含 reg51.h 所引述的資料
unsigned  int  i;                     //宣告全域變數 i
extern  void  delay  (void);          //宣告存在一個外部副函式 delay
```

```
void   main (void)
{   i++;                                    //全域變數所有函式均可直接取用
   delay();                                 //呼叫外部延遲副函式
}
/*以下是外部程式的片段, 引入所需的資料長
度及需存取變數位置*/
void   delay   (void)
{   unsigned   char   s;                    //宣告區域變數 s
    extern   unsigned   int   i;            //宣告外部存在一個變數 i
   for (s=0; s<8; s++)
   {   i++; }                               //外部宣告的全域變數 i = i+1
}
```

3.3.1.5 暫存器變數(register)

其視野及生命週期與區域變數相同，但是暫存器變數的優點是運算速度較區域變數更快，而且是僅有區域變數才可以宣告為暫存器變數來使用，資料型態必須是整數類，也就是說必須要在函式內宣告暫存器變數。

register 修飾字 變數資料型態 變數名稱;

```
#include    <reg51.h>                   //編譯的程式包含 reg51.h 所引述的資料
void   delay   (unsigned   int);        //宣告在主程式後存在著 delay 的副程式
void   main (void)                      //主函式
{   register   unsigned   int   i;      //宣告函式存在區域變數形態的暫存器變數 i
    for( i=0 ; i<50 ; i++)              //區域變數中 x 累加
    delay(50);
}
void   delay   (unsigned   int   j)     // 延遲副函式
{   register   unsigned   int   i = 0;  //宣告函式存在區域變數形態的暫存器變數 i
    do   {   i++;   x++; }              // i = i+1   ,   x=x+1
while ( i < j );                        //當 i<j 條件成立時繼續執行迴圈
}
```

3.3.2 變數的資料型態

每一個變數的設定都需要有名稱和型別, 所謂名稱通常是指一個字節(不可與保留字相同且第一個字必須是英文), 而型別是指變數內存放資料的類別, 不同的資料運算與處理相對應著不同的變數型別，選擇合適的型別才能有效的運用記憶體空間且正確的執行資料運算，以下表列各種資料型態的位元數及數值範圍並逐一詮釋。

資料型態	位元數	位元組	數值範圍
bit	1		0 ~ 1
char	8	1	-128 ~ +127
signed char	8	1	-128 ~ +127
unsigned char	8	1	0 ~ 255
enum	16	2	-32768 ~ +32767
short	16	2	-32768 ~ +32767
signed short	16	2	-32768 ~ +32767
unsigned short	16	2	0 ~ 65535
int	16	2	-32768 ~ +32767
signed int	16	2	-32768 ~ +32767
unsigned int	16	2	0 ~ 65535
signed long	32	4	-2147483648 ~ +2147483647
unsigned long	32	4	0 ~ 4294967295
float	32	4	±1.175494351E+38 ~ 3.402823466 E-38
double	64	8	±2.2250738585072014E+308 ~

			±1.7976931348623158E+308

3.3.2.1 bit(位元變數)

位元變數型態用來儲存邏輯位元資料，通常僅用來判斷邏輯狀態是否成立，與許多應用軟體宣告的 bool(布林變數)功能相同。以下範例假設呼叫某副函式並判斷副函式的回傳值是否為 1，如果為 1 則 P1 輸出為 0xFF。

bit 變數名稱;

```
bit   x;                  //宣告位元變數 x
x = DATA();               //呼叫 DATA()副函式並將回傳資料儲存於 x 變數
while(x)                  //判斷 x 變數是否為 1，如是則執行以下敘述
{   ……..; }               //命令或運算式
```

3.3.2.2 char(字元變數)

字元變數型態可用來儲存字元資料如英文字母'A'，亦可用來儲存 0～255 大小的數據資料，差異在於如果儲存為字元資料時，經過編譯器編譯會將字元自動轉換為 ASCII 碼型態存在，亦即輸出 LCD 等相關顯示介面時不需要再轉換一次 ASCII 碼。以下範例儲存兩種字元型態，陣列 A[]是以字節方式儲存字元資料，陣列 B[]是每個單一字元指向陣列中的每一格儲存，而兩種型態結果是相同的。

修飾字 char 變數名稱;

```
unsigned  char  A[] = "abcdefgh";                          //宣告字元陣列
unsigned  char  B[] = {' A ', ' B ', ' C ', ' D ', ' E ', ' F ', ' G ', ' H '};  //宣告字元陣列
unsigned  char  x;
for( x=0 ; x<8 ; x++ )                                     //宣告字元變數用來執行常態數據
{   LCD_Set_Cursor( 0, x );
 WriteDataLCD(A[x]);                                       //指向 LCD 第一行
}                                                          //從第一個字元 a 開始顯示
for( x=0 ; x<8 ; x++ )
{   LCD_Set_Cursor( 4, x );
 WriteDataLCD(B[x]);                                       //指向 LCD 第二行
}                                                          //從第一個字元 A 開始顯示
```

3.3.2.3 int(整數變數)

整數變數型態可用來儲存 0～65535 大小的數據資料，並不具有小數點以後位數的部分，但是 int 整數變數型態受編譯器系統的影響會產生不同的結果，在 MS-DOS 及 16 位元版本的 Windows 介面軟體它們的長度為 16 位元，在 32 位元操作系統中它們的長度為 32 位元，本書僅以 keil uVision 編譯軟體 16 位元為例。以下範例是將整數變數作為延遲累計計數。

修飾字 int 變數名稱;

```
void  main  (void)                  //宣告字元變數用來執行常態數據
{  unsigned  int  x = 65535;        //引入變數 j 必須大於 x ，否則將產生暫存器溢位
   delay(x); }                      //宣告程式存在區域變數形態的暫存器變數 i
void  delay  (unsigned  int  j)
{  unsigned  int  i = 0;
    do  {  i++; }                   // i = i+1
    while(i<j); }                   //當 i<j 條件成立時繼續執行迴圈
```

3.3.2.4 float(單精確浮點數變數型態)

單精確浮點數變數型態可用來儲存±1.175494351E+38～3.402823466 E-38 大小的數據

資料，E+38 及 E-38 是以科學記號表示法來標示這個變數的精確度範圍。以下範例是假設某圓的直徑為 5cm，計算該圓的圓週長度。

修飾字 float 變數名稱;

```
unsigned  int  a=5 ;        //設圓心直徑為 5cm
float  b=3.414 ;            //圓週率 π
float  c ;
while(1)
{  c = a * b ;  }           //計算圓週長度
```

3.3.2.5 double(雙精確浮點數變數型態)

雙精確浮點數變數型態定義上即為單精確浮點變數的提升精確變數，可用來儲存 ±2.2250738585072014E+308 ~ ±1.7976931348623158E+308 大小的數據資料，使用方法相同於 float 單精確浮點數變數型態，至於使用時機則視乎於撰寫程式的目的而自行選擇。

修飾字 double 變數名稱;

```
unsigned  int  a=5 ;        //設圓心直徑為 5cm
double  b=3.414 ;           //圓週率 π
double  c ;
while(1)
{  c = a * b ;  }           //計算圓週長度
```

3.3.2.6 enum(例舉型態，enumeration)

例舉型態是一種特殊的常數定義方式，可以使我們自行定義資料型態及設定值，藉以提高程式的可讀性，而在例舉型態的敘述裡，可以有一個以上的例舉型態常數及例舉型態變數，如果在沒有特別定義的情況下，編譯器會自動將常數 1 的值設為 0，常數 2 的值設為 1… 以此類推，而在例舉型態的使用上有兩種方法 :

●例舉型態第一種宣告方法 : 定義與宣告變數分開

enum 例舉型態名稱
{ 常數 1， 常數 2, 常數 2 ， … ， 常數 X };
enum 例舉型態名稱 變數 1 變數 2 … 變數 X ;

```
enum  color                     //宣告例舉型態名稱為 color
{  red = 10,  green,  blu  };   //宣告例舉型態常數，且 red=10
enum  color  hat,  cool;        //宣告例舉型態變數
hat = red;                      //變數 hat = 10
cool = blu;                     /*變數 cool = 2，blu 未設定常數值，編譯器自動編譯其存在位址順位為
                                  其常數值*/
```

●例舉型態第二種宣告方法 : 定義後立即宣告變數

enum 例舉型態名稱
{ 常數 1， 常數 2, 常數 2 ， … ， 常數 X } 變數 1 變數 2 … 變數 X;

以上兩種方法的語法結構雖然略有差別，但是其結果是相同的，以下便取用第一種型態來宣告例舉型態說明。

```
enum  color {  red = 10,  green,  blu };   //宣告例舉型態名稱及變數
struct                                      //宣告 struct 結構
{  char  one[10] ;                          //宣告結構陣列
   int  two ;                               //宣告結構變數
   enum  color  three ;                     //在結構內宣告例舉型態變數
}temp ;                                     //struct 結構名稱為 temp
```

```
temp. two =0xAA ;                    //結構變數 temp.two = 0xAA
temp. three = red ;                  //temp. three = 10 結構內的例舉型態變數等於 red
```

3.3.3 變數的修飾字

修飾字的使用在於改變變數的存放位置、屬性、大小及精確度而存在，透過修飾字便可以宣告更多種的資料型態，但是，並非每一種資料型態都可以使用全部的修飾字，以下將逐一說明使用方法及技巧:

3.3.3.1 short (簡短)

簡短修飾字是為了固定 int 變數的位元數而存在，避免 int 整數變數在不同的編譯軟體或作業系統中，被賦予超過 4 個位元組的空間而造成記憶體的浪費，如果宣告 short 修飾字時未宣告 int 變數，編譯器將自動編譯成修飾 int 變數型態。

short 變數資料型態 變數名稱;

```
short  int  x ;                  //無論任何編譯軟體都僅佔有 4 個位元組的空間
unsigned  short  int ;
unsigned  short  x ;
```

3.3.3.2 long (長整數)

長整數修飾字是為了擴充更大或更精確的數據範圍而設計，可以用來修飾 int 整數變數及 double 雙精確浮點數兩種變數型態，當修飾 int 變數時位元組將被擴充為 2 倍(8 bytes)，而且宣告 long 修飾字時可以不用宣告 int 變數，當修飾 double 變數時，則將精確度由 8 bytes 提升為 12 bytes 的容量使用。

long 變數資料型態 變數名稱;

```
long  int  x ;                   //長整數資料將佔有 int 整數資料的 2 倍記憶體空間
unsigned  long  int  x ;
unsigned  long  x ;
long  double  x = 3.1414 ;    //長整數雙精確浮點變數將佔有 12 bytes 記憶體空間
```

3.3.3.3 signed (有號數)

有號數修飾字是指變數存在著正負符號，當運算或宣告數據必須存在著正數與負數兩種型態時使用，但相對變數的正與負數的尺寸將只剩下一半，如果宣告變數時未加以定義 signed， 則編譯器將自動編譯為有號數型態。

signed(可省略) 變數資料型態 變數名稱;

```
int  x ;                    //有號數變數 x 的數值範圍-32768 ~ +32767
signed  int  x ;
```

3.3.3.4 unsigned (無號數)

無號數修飾字是指變數不存在著正負符號，變數宣告後將僅剩餘正數可使用，而變數的尺寸將與變數的位元組相符。

unsigned 變數資料型態 變數名稱;

```
unsigned  char  x ;    //無號數變數 x 的數值範圍 0 ~ 256
unsigned  int  x ;      //無號數變數 x 的數值範圍 0 ~ 65535
```

3.3.4 變數的強制轉換型別

變數之間可能存在著型別不同的差異，而造成運算或進位的過程中產生暫存器溢位，或數據上的錯誤甚至於系統當機，但又考量於變數型別的設定可節省記憶體的應用或多數變數間傳遞的便利性，而不適於改變變數型態時，便可使用以下技巧將變數型態強迫轉換後在執行運算。

變數名稱 = (強制轉換變數型態)變數名稱;

```
void  main  (void)
{  unsigned  int  a=50 ;        //宣告整數變數 a
   unsigned  char  b=10 ;       //宣告字元變數 b
   unsigned  int  c ;           //宣告整數變數 c
   while(1)
   {  c = a * (int) b ;  }      //強迫字元變數 b 轉換成整數變數後再與運算
}
```

3.3.5 記憶體的型別

當程式設計者將變數或常數宣告後，必定會在程式記憶體或資料記憶體中佔有一定的區塊，供應程式運作過程存取使用，也因為變數或常數存在於記憶體空間的不同，將影響程式在執行時的速度，而 keil C 編譯軟體提供以下的指令，讓程式設計者可以依據實際上的需求改變記憶體存在的位置或存取方式。

記憶體型別	範例	說明
code	MOVC @A+DPTR	程式記憶體空間
data	MOV A,#30H	內部記憶體空間，採用直接定址法，存取速度最快
idata	MOV A,@R0	內部記憶體空間，採用間接定址法
bdata	MOV A,#25H	內部記憶體空間，位元定址空間
pdata	MOVX A,@R0	外部記憶體空間，以 R0 或 R1 暫存器為指標
xdata	MOVX A,@DPTR	外部記憶體空間，以 DPTR 暫存器為指標，存取速度最慢

事實上，記憶體的型別使用方式與修飾字是相同的，以下針對每一種類型的記憶體型別使用方法逐一說明。

※以下由 keil C 的記憶體型別並不一定適用於其它編譯器軟體，但並不代表其它編譯器無法改變記憶體存取位置與方式，請另行參閱適用其它編譯器的語言指令教材。

3.3.5.1 code (核心記憶體)

核心記憶體與其它編譯系統之 const 記憶體意義相同，程式中除了變數是必須隨時存取的以外，仍有相當多的資訊是固定不變的常數，為了不佔用資料記憶體空間，我們可以在變數宣告前加上 code 修飾字，將資料存放到程式記憶體中加以運用。

code 修飾字 變數資料型態 變數名稱;

```
code  long  x = 99999999;                    //將 long 長整數變數 x 存放在程式記憶體中
code  unsigned  int  x = 50;
code  unsigned  short  int  x = 50;
code  unsigned  char  x[] = "ABCDEFGH";
```

※當變數宣告為核心記憶體型態即無法於程式過程中改變數據，必須在宣告同時賦予該變數常數初始資料，否則程式將視為錯誤。

3.3.5.2 data (內部資料記憶體)

內部資料記憶體只需要較少的的指令週期時間就可以存取，因為採用直接定址法，相對的存取速度最快，但是，data 型別的記憶體只有 0x00~0x7F 位元空間大小，所以一般是定義需要高速存取的變數使用。使用命令格式如下。

data 修飾字 變數資料型態 變數名稱;

```
data unsigned char i;     //宣告一個記憶體型態的無號數字元變數 i
unsigned char i;          //宣告一個記憶體型態的無號數字元變數 i
```

※keil C 編譯軟體將 data 設為預設的記憶體型態，當宣告變數沒有特別宣告記憶體型態時，編譯軟體將自動設為 data 記憶體型態。

3.3.5.3 idata (內部資料記憶體)

相同於 data 記憶體型態的內部資料記憶體，但是採用間接定址法，所以存取速度略慢於 data 記憶體型態，記憶體位元空間大小 0x00~0xFF 略大於 data 記憶體型態，通常將使用較不頻繁的全域變數宣告於此。使用命令格式如下。

ideta 修飾字 變數資料型態 變數名稱;

```
idata unsigned char i;     //宣告一個記憶體型態的無號數字元變數 i
idata unsigned int j;      //宣告一個記憶體型態的無號數字元變數 i
```

3.3.5.4 bdata (內部資料記憶體)

同樣屬於內部記憶體空間的 bdata 採用的是位元定址法，較特別的使用方式是必須與 sbit 指令配合使用，這是為了簡化結構體(struct)的位元存取使用方法而設定，其位址範圍為 0x20~0x2F 共 16 個位元組的記憶體空間可以使用位元定址法，其意義在於當宣告一個 bdata 記憶體型態的字元或整數變數時，可以搭配 sbit 指令將變數的每一個位元重新宣告為單一位元變數名稱使用。

bdeta 修飾字 變數資料型態 變數名稱;
sbit 位元變數名稱 = btata 變數名稱 ^ 位元 ;

```
bdata unsigned char i;  //宣告一個字元變數 i
sbit flag_i7 = i^7;     //宣告位元變數 flag_i7 相對位置 i 變數的第 7 位元
sbit flag_i6 = i^6;     //宣告位元變數 flag_i6 相對位置 i 變數的第 7 位元
sbit flag_i5 = i^5;     //宣告位元變數 flag_i5 相對位置 i 變數的第 7 位元
sbit flag_i4 = i^4;     //宣告位元變數 flag_i4 相對位置 i 變數的第 7 位元
sbit flag_i3 = i^3;     //宣告位元變數 flag_i3 相對位置 i 變數的第 7 位元
sbit flag_i2 = i^2;     //宣告位元變數 flag_i2 相對位置 i 變數的第 7 位元
sbit flag_i1 = i^1;     //宣告位元變數 flag_i1 相對位置 i 變數的第 7 位元
sbit flag_i0 = i^0;     //宣告位元變數 flag_i0 相對位置 i 變數的第 7 位元
```

3.3.5.5 pdata(外部資料記憶體)

外部擴充記憶體型態需要較多的指令週期時間存取資料，而 pdata 型別的記憶體僅具有 0x00~0xFF 位元空間大小，因定址空間較小，一般用於隨機存取記憶體或外部擴充埠使用，而在 8951 單晶片系統中，必須搭配 ALE、P0、RD、RW 腳位及外部電路使用。在命令格式必須搭配絕對位址指令「 at 」應用，命令格式使用以下任何一種均可。

pdeta 修飾字 變數資料型態 變數名稱 at 定址位置;
修飾字 變數資料型態 pdeta 變數名稱 _at_ 定址位置;

```
char pdata A_PortA _at_ 0x80;       //宣告外部記憶體字元型態 A_PortA 相對位置 0x80
char pdata A_PortB _at_ 0x81;       //宣告外部記憶體字元型態 A_PortA 相對位置 0x80
char pdata A_PortC _at_ 0x82;       //宣告外部記憶體字元型態 A_PortA 相對位置 0x80
/* P0 埠輸出 0x82 相對位置並由 ALE 栓鎖相對位置後，依存取命令執行 RW、RD 由 P0 發送訊號 */
A_PortC = 0xFF;                     //對外部記憶體 A_PortC 輸出資料 0xFF
```

※宣告相對位置將由 ALE、P0 腳位先輸出定址位置後，隨後再由 P0、RD、RW 執行資料的存取運作，可參考 8255 外部擴充埠電路接法。

3.3.5.6 xdata(外部程式記憶體)

相同於 pdata 外部擴充記憶體型態的運作方式，但是 xdata 具有 0x0000~0xFFFF 較大的存取空間，相對的需要最長的指令週期時間運作，一般具有時效運算的資料均不建議採用，而在使用上也必須搭配 ALE、P0、RD、RW 腳位及外部電路使用。命令格式可以搭配絕對位址指令「_at_」應用，也可依外部記憶體形式直接宣告變數使用，絕對位置定址與宣告變數型態命命格式如下。

```
xdeta 修飾字 變數資料型態 變數名稱 _at_ 定址位置;
修飾字 變數資料型態 xdeta 變數名稱;
```

```
char xdata DATA_A _at_ 0x01;        //宣告外部記憶體字元型態 DATA_A 相對位置 0x01
char xdata DATA_B _at_ 0x02;        //宣告外部記憶體字元型態 DATA_A 相對位置 0x02
char xdata DATA_C _at_ 0x03;        //宣告外部記憶體字元型態 DATA_A 相對位置 0x03
unsigned char xdata WORD[10];       //宣告外部記憶體一維陣列
```

3.3.6 特殊變數型別

在 8951 系列單晶片具有兩個特殊的內部記憶體指令，分別為 sfr (Special Function Register)特殊功能暫存器及 sbit 位元變數指令，而在各種不同的編譯器均有相對應於 sfr 及 sbit 的功能指令來使用，所以以下所述命令格式不一定使用於其它編譯軟體，請另行參閱相對應於其它編譯器的使用規範。

資料型態	位元數	位元組	數值範圍
sbit	1		0～1
sfr	8	1	0～255
sfr16	16	2	0～65535

3.3.6.1 sfr (Special Function Register，特殊功能暫存器) 變數

凡舉微電腦晶片或單晶片系列的所有產品，無論任何廠牌、形式、規格都需要對外部 I/O 埠或控制暫存器等所以功能加以定義，程式設計師才得以撰寫對應的程式控制及運用，在 8951 系列晶片使用的是 sfr 這個指令來宣告特殊功能的相對位置及功能名稱，再使用功能名稱改變特殊暫存器的內容來運用晶片的所有功能。

```
sfr 特殊暫存器名稱 = 特殊暫存器起始位址;
```

以下範例實際對應的是 89S51 的標頭檔<AT89x51.h>

```
sfr P0    = 0x80;       //宣告 P0 埠的特殊功能暫存器位置在 0x80
sfr P1    = 0x90;       //宣告 P1 埠的特殊功能暫存器位置在 0x90
sfr P2    = 0xA0;       //宣告 P2 埠的特殊功能暫存器位置在 0xA0
sfr P3    = 0xB0;       //宣告 P3 埠的特殊功能暫存器位置在 0xB0
sfr PSW   = 0xD0;       //宣告 PSW 的特殊功能暫存器位置在 0xD0
sfr ACC   = 0xE0;       //宣告 ACC 的特殊功能暫存器位置在 0xE0
sfr B     = 0xF0;       //宣告 B 的特殊功能暫存器位置在 0xF0
sfr SP    = 0x81;       //宣告 SP 的特殊功能暫存器位置在 0x81
sfr DPL   = 0x82;       //宣告 DPL 的特殊功能暫存器位置在 0x82
```

```
sfr   DPH   = 0x83;        //宣告 DPH 的特殊功能暫存器位置在 0x83
sfr   PCON = 0x87;        //宣告 PCON 的特殊功能暫存器位置在 0x87
sfr   TCON = 0x88;        //宣告 TCON 的特殊功能暫存器位置在 0x88
sfr   TMOD = 0x89;        //宣告 TMOD 的特殊功能暫存器位置在 0x89
sfr   TL0    = 0x8A;      //宣告 TL0 的特殊功能暫存器位置在 0x8A
sfr   TL1    = 0x8B;      //宣告 TL1 的特殊功能暫存器位置在 0x8B
sfr   TH0    = 0x8C;      //宣告 TH0 的特殊功能暫存器位置在 0x8C
sfr   TH1    = 0x8D;      //宣告 TH1 的特殊功能暫存器位置在 0x8D
sfr   IE     = 0xA8;      //宣告 IE 的特殊功能暫存器位置在 0xA8
sfr   IP     = 0xB8;      //宣告 IP 的特殊功能暫存器位置在 0xB8
sfr   SCON = 0x98;        //宣告 SCON 的特殊功能暫存器位置在 0x98
sfr   SBUF = 0x99;        //宣告 SBUF 的特殊功能暫存器位置在 0x99
```

※特殊暫存器是不能存放任何變數的，否則將影響晶片的正常功能運作。

※一般來說，各家廠牌均已撰寫單晶片的相對特殊功能暫存器空間，供應程式設計者使用，當引入「#include」單晶片的相對應用標頭檔後，即不需要在主程式中另行宣告。

3.3.6.2 sfr16 (Special Function Register 16bit, 特殊功能暫存器) 變數

sfr16 相對於 sfr 變數的定義功能，都使用在定義特殊功能暫存器的名稱及位置，但是在 8 位元單晶片裡，如果具有 16 位元功能的特殊功能暫存器變數需求，則需要這個特殊指令來完成，在 8951 系列單晶片裡，只有 DPTR 暫存器是屬於 16 位元的資料型態，且同樣的已經定義在 89S51 的標頭檔<AT89x51.h>裡可以直接使用。

```
sfr16   特殊暫存器名稱 = 特殊暫存器起始位址;
```

```
sfr16   DPTR = 0x82;          //宣告 P0 埠的特殊功能暫存器位置在 0x82
```

3.3.6.3 sbit (位元變數)

對於變數而言，當我們要對一個變數進行一個位元的讀取或判斷時，可以使用 bdata 記憶體型別來進行宣告，而對於特殊功能暫存器已經指向的位址裡，則可以使用 sbit 位元變數直接定義我們需要的名稱來指向暫存器空間的任何一個位元，在程式寫作時，可以直接撰寫 P0 = 0xFF 代表 P0 埠的所有腳位輸出均為 1，但是，如果僅需要改變第二腳位的值或輸出入狀態時，可以直接撰寫 P0^1 = 0 來改變腳位的狀態，這就是因為在標頭檔裡已經完成特殊暫存器的 sbit 宣告，以下範例實際對應 89S51 的標頭檔<AT89x51.h>參考使用。

```
sbit   特殊暫存器位元名稱 = 特殊暫存器位元位址;
```

```
sbit   P0_0 = 0x80;       //宣告位元名稱 P0_0 相對於特殊暫存器位元位置 0x80
sbit   P0_1 = 0x81;       //宣告位元名稱 P0_0 相對於特殊暫存器位元位置 0x81
sbit   P0_2 = 0x82;       //宣告位元名稱 P0_0 相對於特殊暫存器位元位置 0x82
sbit   P0_3 = 0x83;       //宣告位元名稱 P0_0 相對於特殊暫存器位元位置 0x83
sbit   P0_4 = 0x84;       //宣告位元名稱 P0_0 相對於特殊暫存器位元位置 0x84
sbit   P0_5 = 0x85;       //宣告位元名稱 P0_0 相對於特殊暫存器位元位置 0x85
sbit   P0_6 = 0x86;       //宣告位元名稱 P0_0 相對於特殊暫存器位元位置 0x86
sbit   P0_7 = 0x87;       //宣告位元名稱 P0_0 相對於特殊暫存器位元位置 0x87
```

3.3.6 絕對位置指令

對於變數的存取位置而言，是可以指定其存取的的絕對位置，而大部分的是在指定擴充暫存器的位置時使用，當然，一般變數如果有使用上的需要，也是可以指定絕對位置的。相對於記憶體型別的外部記憶體宣告格式，絕對位置具有相同的兩種語法格式。

記憶體型別 修飾字 變數資料型態 變數名稱 at 絕對位置;
修飾字 變數資料型態 記憶體型別 變數名稱 _at_ 絕對位置;

```
char  pdata  A_PortA  _at_  0x80;        //宣告外部記憶體字元型態 A_PortA 相對位置 0x80
pdata  char  A_PortB  _at_  0x90;        //宣告外部記憶體字元型態 A_PortB 相對位置 0x90
char  idata  DATA  _at_  0xFF;           //宣告內部記憶體字元型態 DATA 相對位置 0xFF
```

3.4 運算式(運算子及運算元)

3.4.1 設定運算子

設定運算子即是所謂的等號「=」，運算範圍可以是任何進制間的轉換，在編譯器編譯後都將成為一連續的二進位編碼，符號的定義是將等號右邊的數據指定給左邊的變數。

運算子	功能	範例	說明
=	相等	A =B	A 變數內容等於 B 變數內容
		A=100	A 變數內容等於十進制 100
		A=0xFF	A 變數內容等於十六進制 FF

3.4.2 算數運算子

算數運算子是作為數學式運算所使用，運算範圍可以是任何進制間的運算，而 C/C++ 提供的算術運算子共有五種，分別是「+」、「-」、「*」、「/」、「%」。

運算子	功能	運算子	功能	運算子	功能
+	加法	*	乘法	%	除法取餘數
-	減法	/	除法取商數		

使用算數運算子必須注意，除法取餘數運算式中分母不可以為 0，而乘法如 B*C 必須加上運算子符號不可以簡寫成 BC，以下提供簡單的運算結果範例參考。

```
unsigned   int   A , B=100, C=9 ;   //設定 A ,B ,C 三變數，且 B 初值 100，C 初值 5
A = B+C;            //計算結果 A = 109
A = B-C;            //計算結果 A = 91
A = B*C;            //計算結果 A = 900
A = B/C;            //計算結果 A = 11
A = B%C;            //計算結果 A = 1
```

3.4.3 比較運算子

比較運算子是當有兩個或兩個以上的數據，必須要以數學式互相比較時使用，而比較的結果只有比較式成立或不成立兩種情形，其運算結果為布林值，也就是真「true」或假「false」，結果為真即以整數「1」表示，若為假則以「0」表示。

運算子	功能	運算子	功能	運算子	功能
==	等於	>	大於	>=	大於等於
!=	不等於	<	小於	<=	小於等於

比較運算子通常使用於迴圈的條件判斷式，以下範例是將兩變數比較結果後，判斷為真「true」則執行迴圈敘述命令，為假「false」則跳過不理會。

```
unsigned   int   A=10 , B=5 ;        //設定變數 A=10 , B=5
if( A==B){ ……;}                      //A 等於 B 條件不成立，迴圈敘述命令跳過不理會
if( A !=B){ ……;}                     //A 不等於 B 條件成立，執行迴圈敘述命令
if( A > B){ ……;}                     //A 大於 B 條件成立，執行迴圈敘述命令
if( A < B){ ……;}                     //A 小於 B 條件不成立，迴圈敘述命令跳過不理會
if( A>=B){ ……;}                      //A 大於或等於 B 條件成立，執行迴圈敘述命令
if( A<=B){ ……;}                      //A 小於或等於 B 條件不成立，迴圈敘述命令跳過不理會
```

3.4.4 邏輯運算子

邏輯運算子是將兩個或兩個以上的數據，必須以邏輯或布林數進行比較運算時，其運算方法是以二進制的方式執行，運算結果相同於比較運算子為布林數，無論兩個比較的數據是否大於 1，結果都只能有真「true」或假「false」來表示。

運算子	功能	運算子	功能	運算子	功能
&&	AND	||	OR	!	NOT

●「AND」、「OR」、「NOT」、「XOR」真值表如下所示，

X	Y	AND	X	Y	OR	X	NOT	X	Y	XOR
0	0	0	0	0	0	1	0	0	0	0
0	1	0	0	1	1	0	1	1	0	1
1	0	0	1	0	1			0	1	1
1	1	1	1	1	1			1	1	0

●邏輯運算方式如以下範例所示，設兩變數 A=0xFF、B=0x00、C=0xBB，舉例 C=0xBB 是為了證明當邏輯比較結果只要不等於 0，判斷結果都為真「true」。

A&&B		A&&C		B||C		!(B||C)	
A	1 1 1 1 1 1 1 1	A	1 1 1 1 1 1 1 1	B	0 0 0 0 0 0 0 0	B	0 0 0 0 0 0 0 0
B	0 0 0 0 0 0 0 0	C	1 0 1 1 1 0 1 1	C	1 0 1 1 1 0 1 1	C	1 0 1 1 1 0 1 1
0	0 0 0 0 0 0 0 0	1	1 0 1 1 1 0 1 1	1	1 0 1 1 1 0 1 1	1	0 1 0 0 0 1 0 0

邏輯運算子通常使用於迴圈的條件判斷式，以下範例是將兩變數比較結果後，判斷為真「true」則執行迴圈敘述命令，為假「false」則跳過不理會。

```
unsigned  int  A=0xFF , B=0x00 , C=0xBB;
if( A&&B){ ……;}          //A&&B=0 條件不成立，迴圈敘述命令跳過不理會
if( A&&C){ ……;}          //A&&C=1 條件成立，執行迴圈敘述命令
if( A ||B){ ……;}         //A || B=1 條件成立，執行迴圈敘述命令
if( ! A ){ ……;}          //!A=0 條件不成立，迴圈敘述命令跳過不理會
if( ! C ){ ……;}          //!C=1 條件成立，執行迴圈敘述命令
/*判斷式執行多重或程序型邏輯運算*/
if(( A&&B)||( A&&C)){ ……;}   //((0)||(1))=1 條件成立，執行迴圈敘述命令
if(( A&&B)&&C){ ……;}         //((0)&&C)=0 條件不成立，迴圈敘述命令跳過不理會
```

3.4.5 位元邏輯運算子

位元邏輯運算子是對位元數直接進行運算，我們可以運用程式寫作技巧，只針對整組位元數的某幾個位元進行運算，無論撰寫程式時設定的變數是十進位還是十六進位，編譯器都將編譯為二進位模式進行位元邏輯運算。

運算子	功能	運算子	功能	運算子	功能
&	AND	~	NOT	>>	RIGHT
|	OR	^	XOR	<<	LIFT

●位元邏輯運算方式如以下範例所示，設兩變數 A=0xAA、B=0xBB。

A&B		A|C		B^C	
A	1 0 1 0 1 0 1 0	A	1 0 1 0 1 0 1 0	A	1 0 1 0 1 0 1 0
B	1 0 1 1 1 0 1 1	C	1 0 1 1 1 0 1 1	B	1 0 1 1 1 0 1 1
Ans	1 0 1 0 1 0 1 0	Ans	1 0 1 1 1 0 1 1	Ans	0 0 0 1 0 0 0 1
A>>1		A<<1		~A	
B	1 0 1 1 1 0 1 1	B	1 0 1 1 1 0 1 1	B	1 0 1 1 1 0 1 1
Ans	0 1 0 1 1 1 0 1	Ans	0 1 1 1 0 1 1 0	Ans	0 1 0 0 0 1 0 0

在程式寫作上，有時候為了輸出入狀態或兩變數結合應用的需求，必須保留或去除某幾個位元又或者取代資料，便需要使用一些邏輯設計的技巧來實現。

```
unsigned   int   A=0xAA , B=0xBB , C=0xFF , D;
D = A & 0x0F                          //刪除高四位元保留低四位元資料，D=00001010
D = B | 0xF0                          //高四位元以 1 取代，D=11111011
D = ((A & 0x0F)|( B & 0xF0))          //A 低四位元與 B 高四位元結合，D=10111010
/*單一位元輸出設 A 為必須輸出數據*/
unsigned   char   x;
for( x=0 ; x<8 ; x++ )
{   D=A & 0x01;                       //8 次迴圈，由最低位元 LSB 逐一輸出至最高位元 MSB
    if(!D){   P1=0; }                 //將 A 變數的最低位元存入變數 D 中
    else {   P1=1; }                  //如果 D 等於 0 則輸出 0
   A++;                               //如果 D 等於 1 則輸出 1
}
```

3.4.6 複合式指定運算子

複合式指定運算子是為了簡化運算式的書寫而設定，其運算結果與算數運算子、邏輯運算子及位元邏輯運算子相同。

運算子	功能	運算子	功能	運算子	功能
+=	相加後等於	%=	取餘數後等於	>>=	向右移後等於
-=	相減後等於	|=	OR 後等於	<<=	向左移後等於
*=	相乘後等於	&=	AND 後等於		
/=	相除後等於	^=	XOR 後等於		

複合式指定運算子功能相對於其它運算子如以下列表所示，執行結果請參閱算數運算子、邏輯運算子及位元邏輯運算子，以下將不再贅述。

複合式運算	等同功能	複合式運算	等同功能	複合式運算	等同功能
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&=B	A=A&B		
A/=B	A=A/B	A^=B	A=A^B		

3.4.7 遞增遞減與條件運算子

遞增與遞減運算子是程式撰寫時相當常用的一項功能，可以使用最小指令週期執行某變數循序的增加或減少，而條件運算子雖然使用上較不頻繁，但是當某迴圈敘述與其功能相仿時，條件運算子仍是簡化程式相當有用的功能。

運算子	功能	運算子	功能	運算子	功能
++	遞增	--	遞減	?:	條件運算子

以下使用比較性的範例來解釋遞增與遞減與條件式運算子的功能，但是必須注意，如果變數宣告使用 code 存放在程式記憶體中，則不能夠使用遞增與遞減運算子。

```
unsigned   int   x , A=20 , B=10 , C=0xFF , D=0xAA;
for( x=0 ; x<8 ; x++ )                // x++相等於  x=x+1   , ++x 結果相等
{ ……; }                               //由於 x 每次僅加 1，所以敘述內的命令或運算式將執行 8 次
for( x=8 ; x>1 ; x-- )                // x--相等於  x=x-1   , --x 結果相等
{ ……; }                               //由於 x 每次僅減 1，所以敘述內的命令或運算式將執行 7 次
/* if()執行結果與條件運算子相等 */
if(A>B){ E= C ; }                     //如果 A>B 則將 C 變數內容複製至 E 變數內容
else   { E=D ; }                      //如果 A<B 則將 D 變數內容複製至 E 變數內容
/*使用條件運算子取代判斷迴圈 */
E = ( A > B ) ? C : D ;
```

3.4.8 記憶體型態與運算子

C/C++提供了一個比較特別的運算子，讓初學者或使用新系統時，對於不確定的記憶體使用空間(如 int)的變數可以查閱其所佔用的空間，而其計算單位為「bytes」。

```
unsigned   int   A , B=10 ;
unsigned   char x[] = "ABCDEFGH!";
A = sizeof   B ;                          //填入變數 B，變數 A 等於 4，代表 4 bytes
A = sizeof (B) ;                          //填入變數 B，變數 A 等於 4
A = sizeof (x) ;                          //填入變數 x，變數 A 等於 9
A = sizeof(double) ;                      //填入變數資料型態，變數 A 等於 8
```

3.4.9 指標與取址運算子

指標與記憶體位置有著很大的關係，當我們宣告一個變數或陣列時就意謂著佔著某一區塊的記憶體空間，而當我們需要延續這個記憶體空間執行命令，或獲取這個記憶體空間位置時，便需要指標變數來執行，而指標變數與一般變數一樣，都是用來存放資料的，不同的是指標變數所存放的是指向記憶體的位置。而取址運算子顧名思義就是取出某個變數的記憶體位置，通常與指標變數合併使用。

運算子	功能	運算子	功能
*(前置)	指標運算子	=&	取址運算子

指標運算子最方便使用的時機，是當一個完整程式有需多的外掛副程式需要引入或回傳多個資料時，因為外部宣告會變的相當複雜，如果使用指標運算子指向欲存取的陣列或直接指向記憶體位置，程式將會相當的簡單化，以下使用簡單的範例說明。

```
unsigned   int   x ;              //宣告 x 整數變數
unsigned   int   *p ;             //宣告 p 指標變數
p =& x;                           //將指標變數 p 的內容指向變數 x 的位置
*p = 100;                         //指標變數 p 位置=100，即 x 變數內容等於 100
```

以下假設主程式出現另一個.C 檔的外部程式，而外部程式必須回傳某一連續的資料給予主程式，我們使用指標變數直接指向陣列的記憶體空間完成存取動作。

```
/*以下是主程式的片段，呼叫外部程式*/
unsigned   char   EEPROM[128];                  //宣告一維陣列，陣列長度 128
READ_EEPROM (128 , EEPROM )                     //將執行次數及陣列位置引入副程式
/*以下是外部程式的片段，引入所需的資料長度及需存取變數位置*/
void   READ_EEPROM   (unsigned int DATALONG , unsigned char *DATA)
{   for (s=0; s<DATALONG; s++)                  //視引入長度改變執行次數
{   *DATA= READ_BYTE();                         //將指標變數位置指向讀取副程式
DATA++;                                         //指標變數內容+1
}   }
```

3.4.10 運算子結合性及優先等級

在數學運算符號上的「先乘除後加減」的運算規則，其實就是運算子的優先等級劃分，在 C/C++的程式裡同樣的規定了所有運算子的優先等級，但是為了避免運算時產生的錯誤，建議撰寫程式時仍應該使用括弧()將運算是分開。而結合性的意思是運算符號對於變數的運算流程，譬如減號「-」是指左邊的變數減右邊的變數，所以結合性上即是由左而右(以下以符號→代表)，優先權及結合性詳如以下列表所示。

優先權	運算子	結合性	優先權	運算子	結合性
最高	() [] . ++(後置) --(後置)	→	↓	^	→
↓	! ~ ++(前置) --(前置) + - *(type) sizeof	←	↓	|	→
↓	* / %	→	↓	&&	→
↓	+ -	→	↓	||	→
↓	<< >>	→	↓	?:	←
↓	< <= > >=	→	↓	= += -= *= /= %= &= ^= |= <<= >>=	←
↓	== !=	→	最低	,	→
↓	&	→			

3.5 函式與流程控制設計

3.5.1 認識函式

C/C++提供的函式功能與數學的函數類似，我們可以輸入輸入函式的參數，經過函式處理後，將運算結果回傳或直接輸出，而函式的名稱除了主函式 main()是固定格式以外，其餘均可自由命名，但名稱不可重複使用。以下 5 點為函式撰寫較容易錯誤的格式:
以下範例為函式的宣告格式:
迴傳值型態 函式名稱 (引入值 1 , 引入值 2 , …… , 引入值 x) {敘述;}

```
//回傳 int 變數 , 名稱 delay0 , 引入 char  x 變數
int  delay0  (char  x)
{  unsigned  a ;  ……;    //宣告變數 a , 執行命令或運算
   return  x;             //直接將 x 變數值回傳至呼叫的程式或函式
}
```

```
//回傳無號數 char 變數 , 名稱 delay1 , 引入無號數 char  x[]陣列
unsigned  char  delay1  (unsigned  char  x[] )
{  unsigned  a ;  ……;    //宣告變數 a , 執行命令或運算
   return  a;             //將 a 變數值回傳至呼叫的程式或函式
}
```

```
//外部函數 , 回傳無號數 char 變數 , 名稱 delay2 , 以指標 x 指向引入 char 變數
extern  unsigned  char  delay2  (unsigned  char  *x )
{  unsigned  a ;  ……;    //宣告變數 a , 執行命令或運算
   return  *x;            //將 x 指標變數最後的值回傳至呼叫的程式或函式
}
```

```
//回傳無號數 char 變數 , 名稱 delay3 , 引入無號數 char  x 及無號數 int  y 兩個變數
unsigned  char  delay3  (unsigned  char  x , unsigned  int  y)
{  unsigned  a ;  ……;    //宣告變數 a , 執行命令或運算
   return  a;             //將 a 變數值回傳至呼叫的程式或函式
}
```

```
//無回傳, 名稱 delay4 , 無引入
void  delay4  (void)
{  ……; }                 //執行命令或運算
```

※函式的開頭第一個字必須是英文字母，不可以為數字或符號。
※每個程式只能有一個且必須有一個主函式 main()。
※函式的名稱(含副程式所宣告的函式名稱)不可重複命名。
※函式內的變數除非特別宣告，否則均為區域變數。
※呼叫有引入功能函式，請注意引入的變數資料型態與函式的變數型態是否相符。
而函式編撰程式上依據主函式及副函式的編譯順序，可以區分為以下兩種型態:

3.5.1.1 「有函式宣告」

是指副函式編撰在主函式之後，而在主函式之前必須定義副函式的型態予編譯器，避免編譯時造成錯誤。

```
void  function1  (void);        //定義主函式後存在著副函式 function1
void  function1  (void);        //定義主函式後存在著副函式 function2
void  main  (void)              //主函式 main
{ ……; }                         //運算或執行命令
void  function1  (void)         //副函式 function1
{ ……; }                         //運算或執行命令
```

```
void  function2  (void)          //副函式 function2
{ ……; }                          //運算或執行命令
```

※除非，副程式編撰後並未在程式內任何位置(包括函式)呼叫，則編譯器將略過錯誤出現警告指示，表示此副函式僅佔有記憶體而無實際功能。

3.5.1.2 「無函式宣告」

是指副函式編撰在主函式之前，因為編譯順序已預知副函式的存在及型態，所以無需再次定義。

```
void  function1  (void)          //副函式 function1
{ ……; }                          //運算或執行命令
void  function2  (void)          //副函式 function2
{ ……; }                          //運算或執行命令
void  main  (void)               //主函式 main
{ ……; }                          //運算或執行命令
```

當我們宣告了一個具有回傳值及引入值的函式後，我們可以使用變數直接指向該函式執行並接收回傳，或是忽略回傳的值不予理會，函式呼叫方法參閱以下範例:

```
/* 假設前述範例的 delay 函式已在主函式之後編寫 */
#include   <reg51.h>              //編譯的程式包含 reg51.h 所引述的資料
unsigned  char  a , b=50 ;        //宣告無號數字元變數 a , b
unsigned  char  c[20] ;           //宣告一維陣列 c
//宣告主函式後存在一個 delay0 副函式，引入變數名稱無須宣告
int  delay0  (char );
//宣告主函式後存在一個 delay1 副函式，陣列引入變數名稱必須宣告
unsigned  char  delay1  (unsigned  char  x[] );
//宣告外部函式存在一個 delay2 副函式，指標引入變數名稱必須宣告
extern  unsigned  char  delay2  (unsigned  char  *x )
//宣告主函式後存在一個 delay3 副函式，引入變數名稱無須宣告
unsigned  char  delay3  (unsigned  char , unsigned  int )
void  main  (void)
{  delay0();   //直接呼叫使用
  a = delay1( c );                // a 變數等於函式運算後迴傳值，引入陣列 c
  a = delay2( c );                // a 變數等於函式運算後迴傳值，引入陣列 c
  a = delay3( b , 30 );           // a 變數等於函式運算後迴傳值，引入變數 b 及常數 30
}
```

上述範例是將副函式定義在主函式之後撰寫，所以必須要在主函式之前作一次「有函式宣告」的動作，讓編譯器知道在主函式以後還存在著某些副函式，避免編譯過程產生錯誤。

3.5.2 return 回傳命令

return 迴傳敘述有兩種基本功能，分別為回傳函式運算資料及離開迴圈，只要宣告回傳(return)指令就相等於函式已執行完畢即將離開迴圈，並不一定要有資料回傳，所以回傳指令通常設置於函式最後執行，但亦可以運用程式寫作技巧將指令設置於程式中端，或多重判斷式內執行，以下範例將解釋三種基本運用方式。

return 變數資料型態;

```
/* 有回傳資料宣告方式，函式宣告的回傳型態必須宣告 */
unsigned  int  function  (void)   //副函式 function
{  unsigned  int  a;              //宣告整數變數 a
……;                              //運算或執行命令
  return  a;  }                   //回傳變數 a 資訊予呼叫的函式或變數
```

```
/* 無回傳資料宣告方式，函式執行完畢則跳離函式 */
void  function  (void)      //副函式 function
{  ……;                     //運算或執行命令
   return ;  }              //回傳變數 a 資訊予呼叫的函式或變數
/* 多重判斷式內執行回傳命令，並強制結束跳離函式 */
unsigned  int  function  (void)    //副函式 function
{  unsigned  int  a , b , c , d;
……;                        //運算或執行命令
if (a>b)
{  ……;                     //運算或執行命令
    return  c ; }           //函式執行至此行即跳離函式
else  if (a<b)
{  ……;                     //運算或執行命令
    return  c ; }           //函式執行至此行即跳離函式
    return  d ;             //回傳變數 a 資訊予呼叫的函式或變數
}
```

3.5.3 一維與多維陣列

陣列是一種連續性的資料結構型態，幾乎所有的高階語言都離不開陣列，其結構型態上與數學式的矩陣是相當類似的，而陣列中每一個元件都相當於一個變數，所以，使用陣列必須要注意一點，如果資料結構是不會改變的，最好使用 code 指令將陣列存放於程式記憶體中，可以避免資料記憶體使用過度。當然可以使用變數

3.5.3.1 一維陣列型態

相對涵義是指在一行資料群裡，有需多的獨立資料存在，而存放記憶體位置是一連續有規則的狀態，其佔用的記憶體空間相等於(資料型態位元組數*陣列長度)，如以下範例(1*10)便佔有 10 bytes 的記憶體空間，其宣告語法如下。

資料型態 陣列名稱(元素個數)

```
unsigned  char  x[ ];     //宣告一個 x 陣列，且保留陣列的資料長度限制
unsigned  char  x[10];  //宣告一個 x 陣列，具有 10 個 unsigned  char 變數空間
unsigned  char  x[ ]={'0','1','2','3','4','5','6','7','8','9'};  /*宣告一個 x 陣列，具有 10 個
    無號數字元變數空間，並指定變數內容 */
```

上述的宣告形同於 unsigned char x , x1 , x2 , … , x9 ;連續宣告 10 個名稱不同的變數，但是記憶體存放的位置卻是無法預測的，而陣列的存放資料型態如下所示，每一個資料格式均是同一個 unsigned char 型態的變數:

記憶體位置	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9
資料格式	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]

3.5.3.2 多維陣列型態

相同於一維陣列的連續資料格式，在二維以上的陣列依其維度命名如「二維陣列」、「三維陣列」、「x 維陣列」等，與一維陣列略有不同的是，不可以保留資料長度的限制，佔用的記憶體空間相等於(資料型態位元組數*維度*陣列長度)，如以下範例(1*3*10)便佔有 30 bytes 的記憶體空間，其宣告語法如下。

```
unsigned  char  x[3][10];    //宣告一個 x 陣列、二維、unsigned  char 型態變數空間
```

上述的宣告形同於 unsigned char x[10] ;連續宣告三組名稱不同的一維陣列，但是記憶體存放的位置卻是一連續的型態:

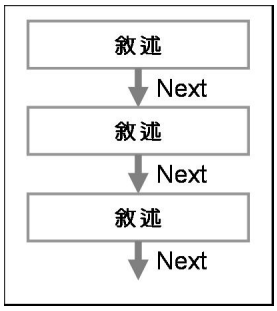
記憶體位置	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9
資料格式	x[0][0]	x[0][1]	x[0][2]	x[0][3]	x[0][4]	x[0][5]	x[0][6]	x[0][7]	x[0][8]	x[0][9]
記憶體位置	n+10	n+11	n+12	n+13	n+14	n+15	n+16	n+17	n+18	n+19
資料格式	x[1][0]	x[1][1]	x[1][2]	x[1][3]	x[1][4]	x[1][5]	x[1][6]	x[1][7]	x[1][8]	x[1][9]
記憶體位置	n+20	n+21	n+22	n+23	n+24	n+25	n+26	n+27	n+28	n+29
資料格式	x[2][0]	x[2][1]	x[2][2]	x[2][3]	x[2][4]	x[2][5]	x[2][6]	x[2][7]	x[2][8]	x[2][9]

其餘多維陣列型態宣告如二維陣列相同

3.5.4 「循序」結構、「選擇」結構與「迴圈」結構

撰寫程式語言最直接的方式是一行一行循序的執行下去，這樣的結構稱為循序結構，而除此之外，為了因應功能的變化需求，程式語言必須具備其它種類的程式流程控制能力，這些非循序結構都是由決策與跳躍來組成，但是，在一般的結構化程式語言中，並不允許使用者自行定義跳耀，而把跳耀移到決策與迴圈之內，在 C/C++程式語言內主要的循序與非循序結構有以下三種:

3.5.4.1 「循序」結構(sequence structure)



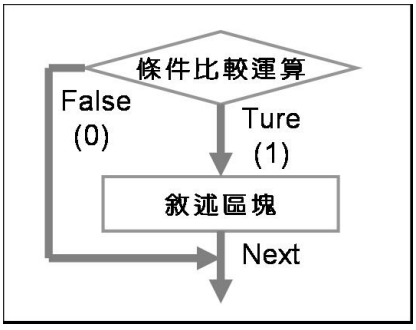
循序結構是程式撰寫的基礎應用，在未撰寫「選擇」與「迴圈」的情況之下，程式均依循著循序結構來運作，而循序結構的定義就是「程式碼執行順序由上而下，一個敘述接著一個敘述依序執行」，並不限定程式碼是在主程式或是副程式中執行，運作方式如流程圖所示。

3.5.4.2 「選擇」結構(selection structure)

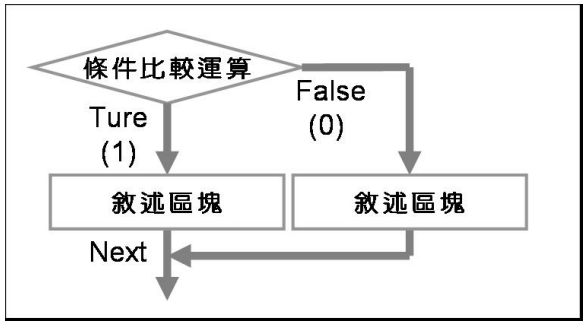
選擇結構代表程式在執行時，會依據條件(運算是或判斷)的結果適當的改變程式執行的順序如果滿足條件就會執行某一敘述區塊，或是當條件都不滿足時執行另一敘述區塊。一般來說，選擇結構可分為「單一選擇」、「雙向選擇」、「多向選擇」等三種。

●**單一選擇結構**：單一選擇結構只能註明當條件成立時，必須要執行的區塊，當條件不成立時則略過敘述區塊不執行。

●**雙向選擇結構**：雙向選擇結構註明了無論條件成立與否，都有相對應的敘述區塊必須執行，二選一的機會程式必須執行其中一段。

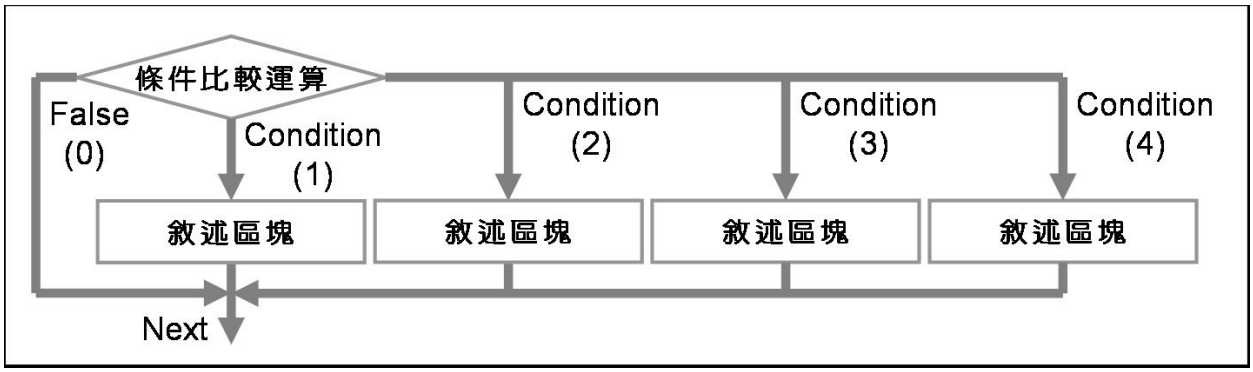


(單向選擇結構)



(雙向選擇結構)

●**多向選擇結構**：多向選擇結構使得比較運算式可以一個以上的結果存在，且可指定每一個結果所相對應的敘述區塊，而多向選擇結構具有單一及雙向結構的特點，可以指定當條件都不成立時略過所有敘述區塊不執行，或是指定必須執行某一區塊。



3.5.4.3 「迴圈」結構(loop structure)

多向選擇結構

迴圈結構代表程式將不斷的執行某一個敘述區塊，直到判斷條件不成立離開迴圈，或是執行跳躍或強制離開迴圈指令為止，一般來說迴圈可分為「」計數迴圈」、「前測式迴圈」、「後測式迴圈」等三種。

●**計數迴圈**：計數迴圈具有「執行指定的迴圈初始運算式」、「檢測迴圈終止條件運算式」、「執行迴圈增量變化」的能力，一般來說計數迴圈可用於預測迴圈執行次數的情況。

●**前測式迴圈**：前測試迴圈即先判斷運算式，如果條件成立才執行迴圈內敘述區塊，當敘述完畢後程式再次回到迴圈的判斷運算式，如果條件依舊成立繼續執行迴圈內敘述區塊，如此週而復始的循環，一直到判斷運算式條件不成立才離開。

●**後測式迴圈**：後測式迴圈則必須先執行敘述區塊一次，再進行條件運算式判斷，重複判斷與執行的特性與前測式迴圈相等，最大的不同是，後測式迴圈無論判斷運算式條件是否成立，至少會執行敘述區塊一次，注意事項與前測式迴圈相同。

※必須注意的是，在程式優先等級較高的敘述區塊或判斷運算式的敘述區塊內，必須要有條件改變的可能性，否則將陷入無窮迴圈(如 while(1){ }迴圈)。

3.5.5 if、else if、else 條件式敘述

if 條件式敘述是屬於「選擇結構」的判斷式之一，具有單一、雙向、多向選擇三種條件敘述方式，當使用多向選擇敘述條件時，可以依據不同的條件作多向的條件式判斷，如以下 if()…else if()範例程式，而當撰寫 if 指令時有以下 3 點是必須注意的。

※if 條件式只執行條件成立的該條件式敘述區塊，執行完畢後即離開整個迴圈，其餘條件判斷與敘述區塊都將略過。

※而敘述區塊執行完畢後並不會回到條件式繼續判斷，而是直接離開條件式敘述往下循序執行程式。

※判斷式「()」裡必須產生比較、邏輯運算子或經過定義產生比較值的巨集，否則程式將出現無法預期的錯誤。

以下 4 種為 if 條件式敘述的命令格式:

3.5.5.1 if 條件式敘述

```
if (條件運算式)      //將條件運算至於括弧內執行判斷
{……; }              //條件成立執行敘述內執行命令或運算式
```

屬於單一選擇結構，當僅有一個敘述區塊且必須於條件成立時執行使用。

```
unsigned  char  a;        //宣告整數變數 a
if ( a!=1)                //如果 a 不等於 1 時條件成立執行下列敘述
{  ……; }                  //執行命令或運算
if ( P1==0xFF)            //直接判斷 I/O port 暫存器空間是否條件成立
{  ……; }                  //執行命令或運算
```

3.5.5.2 if、else 條件式敘述

```
if (條件運算式)     //將條件運算至於括弧內執行判斷
{……; }             //條件成立執行敘述內執行命令或運算式
else               //如果條件不成立指令
{……; }             //如果條件不成立則執行敘述內執行命令或運算式
```

屬於雙向選擇結構，無論條件成立與不成立，均有對應的敘述區塊執行。

```
unsigned  char  a;        //宣告整數變數 a
if (a>50)                 //如果 a 大於 50 時條件成立執行下列敘述
{  ……; }                  //執行命令或運算
else                      //如果 if 條件不成立則執行下列敘述
{  ……; }                  //執行命令或運算
```

3.5.5.3 if、else if 條件式敘述

```
if (條件運算式)          //將條件運算至於括弧內執行判斷
{……; }                  //條件成立執行敘述內執行命令或運算式
else  if (條件運算式)    //次要條件運算式，條件或邏輯變數不一定要與 if()相等
{……; }                  //次要條件成立執行敘述內執行命令或運算式
```

屬於多向選擇結構，當第一條件不成立時，則往第二個條件作判斷，直到某一個條件成立執行該條件的敘述區塊，而 else if()條件式並不限定可以使用幾次，所以判斷式最多將等於系統或晶片的程式記憶體長度，而當沒有任何條件成立，則離開條件敘述式不執行。

```
unsigned  char  a;              //宣告整數變數 a
if (a==1)                       //如果 a 等於 1 時條件成立執行下列敘述
{  ……; }                        //執行命令或運算
else  if (a==2)                 //如果 a 等於 2 時條件成立執行下列敘述
{  ……; }                        //執行命令或運算
else  if (a==3)                 //如果 a 等於 3 時條件成立執行下列敘述
{  ……; }                        //執行命令或運算
```

```
unsigned  char  a , b , c;      //宣告整數變數 a ,b , c 作三種不同的條件式判斷
if (a==1)                       //如果 a 等於 1 時條件成立執行下列敘述
{  ……; }                        //執行命令或運算
else  if (b==2)                 //如果 b 等於 2 時條件成立執行下列敘述
{  ……; }                        //執行命令或運算
else  if (c==3)                 //如果 c 等於 3 時條件成立執行下列敘述
{  ……; }                        //執行命令或運算
```

3.5.5.4 if、else if、else 條件式敘述

```
if (條件運算式)          //將條件運算至於括弧內執行判斷
{……; }                  //條件成立執行敘述內執行命令或運算式
else  if (條件運算式)    //次要條件運算式，條件或邏輯變數不一定要與 if()相等
{……; }                  //次要條件成立執行敘述內執行命令或運算式
else                    //如果條件不成立指令
{……; }                  //如果上述條件都不成立則執行敘述內執行命令或運算式
```

屬於多向選擇結構，當第一條件不成立時，則往第二個條件作判斷，直到某一個條件成立執行該條件的敘述區塊，如果都沒有條件成立，就執行 else 敘述區塊。

```
unsigned  char  a;        //宣告整數變數 a
if (a==1)                 //如果 a 等於 1 時條件成立執行下列敘述
{  ……; }                  //執行命令或運算
```

```
else   if (a==2)              //如果 a 等於 2 時條件成立執行下列敘述
{   ……; }                    //執行命令或運算
else   if (a==3)              //如果 a 等於 3 時條件成立執行下列敘述
{   ……; }                    //執行命令或運算
else                          //當上述條件沒有任何一項成立時則執行下列敘述
{   ……; }                    //執行命令或運算
```

3.5.5.5 if()條件式敘述常用技巧

使用條件運算式的重要規定，是判斷式括弧「()」內必須產生可以被判斷的邏輯式，但並不是限定必須只能帶入邏輯運算式得到結果，事實上，括弧內的結果只要不為零「false」都會被當成條件成立來運作，以下範例提供幾種高階語言撰寫的使用技巧供使用者參考。

```
unsigned   int   char   a=10 , b=5;
If ( (a–b) >= 5)      //判斷式 a-b=5,比較 5>=5 條件成立，執行下列敘述
{ ……;}         //命令或運算式
```

```
Unsigned   int   char   a=0xbb , b=0xaa;
If ( ((a&0x0F) | (b&0xF0))==0xba )   //判斷式條件成立，執行下列敘述
{ ……;}         //命令或運算式
```

```
If ( DATA() )         //如果副函式的回傳值不等於 0 則條件成立，執行下列敘述
{ ……;}         //命令或運算式
unsigned   char   DATA   (void)    //DATA 副函式，具有回傳功能
{ ……;          //命令或運算式
return   a; }
```

```
unsigned   char   A;
# define   DATA   ((A&0xFF)!=0)       //定義含變數算式
if (!DATA)            //如果巨集 DATA 的運算結果為 0 則條件成立，執行下列敘述
{ ……;}         //命令或運算式
```

3.5.6 while

while 條件迴圈是屬於「迴圈結構」的判斷式之一，基本判斷條件與單一選擇敘述方式雷同，當條件判斷成立時執行敘述迴圈，如果條件不成立則略過該迴圈敘述，使用 while 條件迴圈有以下 3 點是必須注意的。

※敘述迴圈內必須有可以改變判斷條件的指令，或是在優先等級較高的中斷函式敘述改變條件，否則程式將進入無窮迴圈無法離開。

※主函式,main()的敘述「{}」內撰寫 while(1)無窮迴圈，是為了避免系統設定、變數宣告、結構初始化及某些副程式的設定被一再的重複宣告，這是一種撰寫技巧，但並不是絕對必須使用。

以下 2 種為 while 條件式敘述的命令格式:

3.5.6.1 while()前測式迴圈

```
while (條件運算式)      //將邏輯運算至於括弧內
{……; }                 //條件成立執行敘述內執行命令或運算式
```

前測式迴圈的意義，是指在進入這個迴圈工作前，必須先滿足判斷式的條件，當條件成立(ture)進入迴圈內執行命令，並停滯在迴圈以內，如果不成立則略過該迴圈敘述往下循序執行程式。如以下範例，因為 a 只執行一次所以不可以是判斷條件，如果變數必須是判斷條件，則可以加入條件敘述式在迴圈敘述內來改變變數條件。

```
unsigned  char  a=0 , b=0 ;   //宣告整數變數 a , b
while(P1<10)                  //P1 為外部輸入 port 當條件成立則進入敘述，直到條件不成立才離開
{  ……;                       //執行命令或運算
   a=P1 ;                     //將 a 指向輸入 port1 ,儲存 P1
   b++                        //b++只執行一次，並不會在等待判斷條件不成立的
}
```

※前測式迴圈當條件持續成立，且敘述「{ }」內無其它迴圈或條件敘述式存在時，則敘述內的命令或運算式並不會重複執行。

3.5.6.2 do…while();後測式迴圈

```
do {……; }          //先執行敘述內執行命令或運算式一次，當條件成立再重複執行
while (條件運算式)   //將邏輯運算至於括弧內
```

前測式迴圈的意義，是指迴圈內的敘述先執行一次之後，再進行判斷式的條件判斷，當條件成立(ture)繼續執行迴圈內命令，並停滯在迴圈敘述內重複執行，如果不成立則略過該迴圈敘述往下循序執行程式。意即迴圈的敘述會被無條件執行一次，再進行條件式的滿足。

```
unsigned  char  a=0;          //宣告整數變數 a
do{  ……;  a=P1 ; }            //先執行命令或運算一次，並將 a 指向輸入 port
while(a<10);                  //如果條件成立則回到以上敘述，直到條件不成立才離開
```

```
unsigned  char  a=0 , b=0 ;   //宣告整數變數 a , b
do
{  ……;                       //執行命令或運算
a=P1 ;                        //將 a 指向輸入 port1， a 將在每次條件成立時重複執行,可視為判斷條件
b++;                          //b++將持續累加至暫存器溢位
}  while(a<10)                //如果條件成立則回到以上敘述，直到條件不成立才離開
```

※後測式迴圈當條件持續成立，則敘述「{ }」內的命令或運算式在每次比較條件之後，都將重複的執行，這是後測式與前測式迴圈最大的不同。

※後測式迴圈的結尾必須加上分號「;」否則編譯將產生錯誤，格式請參考以下範例。

3.5.6.3 while()迴圈的使用技巧

迴圈的使用因為不斷的重複判斷條件成立與否，來決定是否執行敘述內的命令或運算式，則判斷括弧內的邏輯式最好是越簡短越好，可以避免判斷時間過長，造成其它運算式無法執行或外部資料漏失，所以，在 while()判斷式中並不建議置入副函式作為判斷值。

```
while(1)              //判斷式直接等於 1，條件永遠成立進入無窮迴圈
{  ……; }              //執行命令或運算，迴圈外的命令將不再執行
```

```
while( P1)            //將外部輸入直接指向判斷式，當 P1=1 時則條件成立
{  ……; }              //執行命令或運算
```

```
unsigned  char  A;                      //定義含變數算式
# define  DATA  ((A&0xFF)|(P1)!=0)      //當巨集運算條件不等於 0 時條件成立
while(DATA)                             //執行命令或運算
{  ……; }
```

```
void  delay  (unsigned  int  b)
{  unsigned  char  a=0;                 //宣告整數變數 a
   do{  a++ ; }                         //執行命令或運算
   while(a<b);                          //如果條件成立則回到以上敘述，直到條件不成立才離開
}
```

3.5.7 switch 多向條件式敘述

switch 屬於「選擇結構」的多向選擇判斷式，當然，並不是一定要多向選擇才可以執行 switch 敘述，但是如果是單一選擇則使用 if()敘述即可處理。在 switch 多向條件式敘述與

if()…else if 多項條件敘述的差異性上，switch 是在單一的條件作不同的判斷式中選擇條件是否成立，而 if()…else if 多條件敘述可以在多項不同的條件上運作選擇條件是否成立，這是直得注意的地方，撰寫程式技巧上應視乎不同情況選擇應用。

※switch 條件式只執行條件成立 case 敘述區塊，執行完畢後即離開整個迴圈，其餘條件判斷與敘述區塊都將略過。

※case 的敘述區塊可以僅執行 break 命令，但是判斷條件值必須是資料常數不可以是變數，且每一個 case 的條件值必須不同。

※case 與 default 的順序並不重要，default 可以在條件式內任意撰寫，但是，當 default 並不是在敘述「{}」的最後時，則必須加上 break 跳離條件式。

※當 case 敘述區塊並未宣告 break 跳離指令時，程式以循序方式判斷某一個 case 條件成立並執行敘述區塊後，則會將這個 case 以下的所有敘述區塊執行完畢，無論以下的 case 條件是否成立，產生 falling through 現象。但是在程式編譯上並不會產生錯誤，所以不宣告 break 跳離指令也可以是程式寫作的另一種技巧。

3.5.2.1 switch()…case 多向條件式敘述

```
switch (條件運算式)     //將條件運算置於括弧內
{   case   條件值:       //第一判斷式   (條件運算式) ==條件值  則條件成立
      ……;   break;     //命令或運算無須敘述括弧，但必須要有離開敘述指令 break
      case   條件值:    //第二判斷式   (條件運算式) ==條件值  則條件成立
      ……;   break;     //命令或運算無須敘述括弧，但必須要有離開敘述指令 break
}
```

當一項條件需要被判斷是否在某幾個條件成立下運作使用，以循序的方式向下判斷其中是否有成立的 case 程序，而相同於 else if()條件式敘述，並不限定 case 判斷式的數量，最大限制將等同於系統或晶片的程式記憶體尺寸。

```
switch (P0)                                  //將條件運算式指向外部輸入 port 暫存器資料
{   case   0x11:   P1=0x01;   break;         //如果 a=0x11 則條件成立，執行完畢後直接離開敘述括弧
   case   0x12:   P1=0x02;   break;          //如果 a=0x12 則條件成立，執行完畢後直接離開敘述括弧
   case   0x14:   P1=0x04;   break;          //如果 a=0x14 則條件成立，執行完畢後直接離開敘述括弧
   case   0x18:   P1=0x08;   break;          //如果 a=0x18 則條件成立，執行完畢後直接離開敘述括弧
}
```

```
unsigned   char   a=0x12;                    //宣告變數 a，並設定初值為 0x12
switch (a)                                   //將條件運算式置於括弧內
{   case   0x11:   P0=0x01;                  //此行不成立，並不執行
   case   0x12:   P1=0x02;                   //此行條件成立，並於命令執行完畢後往下繼續執行
   case   0x14:   P2=0x04;                   //此行無論條件成立與否，都將執行命令且完畢後往下繼續執行
   case   0x18:   P3=0x08;    break;         //此行無論條件成立與否，都將執行命令，且執行完畢後直接離開
   case   0x10:   P0=0x10;                   //此行不成立，不執行
}
```

3.5.2.1 switch()…case…default 多向條件式敘述

```
switch (條件運算式)     //將條件運算至於括弧內
{   case   條件值:       //第一判斷式   (條件運算式) ==條件值  則條件成立
      ……;   break;     //命令或運算無須敘述括弧，但必須要有離開敘述指令 break
      case   條件值:    //第二判斷式   (條件運算式) ==條件值  則條件成立
      ……;   break;     //命令或運算無須敘述括弧，但必須要有離開敘述指令 break
      default:   ……;   //當所有 case 都不成立時執行
}
```

相同於 if()、else 條件敘述式功能，switch()敘述式一定會執行某一個敘述區塊，而第二個

範例並未將 default 指令置於程式末端，所以必須加上 break 指令離開迴圈。

```
unsigned  char  a;
switch (a)                              //將條件運算式置於判斷式
{  case  0x11:  P1=0x01;  break;        //如果 a=0x11 則條件成立
   case  0x12:  P1=0x02;  break;        //如果 a=0x12 則條件成立
   case  0x14:  P1=0x04;  break;        //如果 a=0x14 則條件成立
   case  0x18:  P1=0x08;  break;        //如果 a=0x18 則條件成立
   default:     P1=0xFF;                //當所有條件不成立時，則執行這個敘述區塊
}
```

```
unsigned  char  a;
switch (a)                              //將條件運算式置於判斷式
{  case  0x11:  P1=0x01;  break;        //如果 a=0x11 則條件成立
   case  0x12:  P1=0x02;  break;        //如果 a=0x12 則條件成立
default:  P1=0xFF;  break;              //當所有條件不成立時，則執行這個敘述區塊
   case  0x14:  P1=0x04;  break;        //如果 a=0x14 則條件成立
   case  0x18:  P1=0x08;  break;        //如果 a=0x18 則條件成立
}
```

3.5.8 for()計數迴圈

for()迴圈是屬於「迴圈結構」的「計數迴圈」結構，是程式寫作的一項重要指令工具，經常被用作有順序性及重複性的命令撰寫格式，與 while()迴圈最大的不同，是當正常運作的情況下 for()迴圈並不是無窮迴圈，當測試比較運算式條件不被滿足時，則離開迴圈往下循序執行其它程式。以下為 for()迴圈的語法格式。

```
for ( 初值設定運算式 ; 測試比較運算式 ; 增量設定運算式 )
```

```
void  main  (void)
{  unsigned  int  a=1000;               //宣告 int 資料型態的變數 a 且初值為 1000
   DATA(a);  }                          //呼叫 DATA 副程式並將 a 變數引入
void  DATA  (unsigned  char  b)         //引入型態為 char 資料型態
{  unsigned  int  c;                    //宣告「初值設定運算式」使用的變數 c
   for ( c=0 ; c<b ; c++ )  }           //當 a 變數大於 255 時，程式將產生錯誤
```

以上範例經常用於撰寫延遲副程式時使用，當程式寫作相當龐長時，很容易疏忽產生以上的錯誤，務必小心在意。當程式進入 for()迴圈後，首先必須了解迴圈內的運作流程，以下以執行 2 次的迴圈為例來解釋作業流程(流程中條件成立以(1)表示)。

```
for ( a=0 ; a<2 ; a++ )     //當 a=2 時條件不成立即可離開迴圈
{  ……;}                     //此行敘述將執行 2 次
```

初值設定運算式→測試比較運算式(1)→敘述→增量設定運算式→測試比較運算式(1)→敘述→增量設定運算式→測試比較運算式(0)→離開迴圈

流程中可以清楚發現「初值設定運算式」僅執行 1 次，接下來便不斷的進行邏輯比較，只要條件成立便進入敘述內執行命令或運算式，然後再進行增量運算，如此週而復始直到條件不成立為止，以下範例為撰寫 for()迴圈的程式方法。

```
//以下副程式可以在呼叫時，引入執行迴圈次數，通常用於延遲
void  DATA  (unsigned  char  b)
{  unsigned  cahr  a;
for ( a=0 ; a<b ; a++  )      //如果為空敘述則無須加入敘述括弧
}
```

```
//以下副程式在呼叫時，引入執行迴圈次數並將變數指定給引入陣列輸出及輸入;
void  DATA  (unsigned  char  b , unsigned  char  temp[ ])
{  unsigned  cahr  a;
   for ( a=0 ; a<b ; a++  )    //當 a 變數小於引入變數 b 時則進入敘述執行命令
```

```
{   P1 = temp[a ] ;              //PORT1 輸出等於陣列 temp[a ]
   temp[a ] = P2 ;               //將陣列 temp[a ]的內容指向 PORT2 輸入
}}
```

※當 for()迴圈置於副函式內，經常使用引入資料型態來測試比較運算式，但是，當引入的資料型態與副函式宣告的變數型態不相同時，可能造成無窮迴圈或無法執行的錯誤。

※for()迴圈的「初值設定運算式」不可以在迴圈括弧內宣告，且不可以使用浮點變數型態(float)及負數。

※for()迴圈的「測試比較運算式」必須使用比較運算子產生 ture 及 false 邏輯判斷型態，且被比較值必須是常數。

※for()迴圈的「增量設定運算式」不可以使用邏輯運算子，否則程式將產生錯誤。

3.5.9 goto、break、continue

在程式執行過程中，有時候必須要強制離開迴圈或是跳離函式，有時候必須在函式內跳躍至某個區段以後執行，這種時候就需要 goto、break、countinue 及 return 指令來強制執行，以下就 4 種指令的語法格式及限制逐一說明。

3.5.9.1 goto 單一函式內跳躍 無條件跳躍指令

goto 敘述是用來引導程式的執行順序到指定的位置，當程式在某一個函式內(可以是主、副函式)執行命令時，可能經過判斷或當某個數據產生時，必須從當前段落跳躍或前進至某個區段執行命令或算試時使用。以下為 goto 指令的 2 種語法格式。

●第 1 種語法為迴序跳躍，可以在同一個敘述層裡執行，但必須注意是否產生無窮迴序。

```
標籤位置:              //指定跳躍的位置，必須加上「:」符號
goto  標籤位置;    //跳躍至標籤位置指令
```

●第 2 種語法為循序跳躍，必須在內層敘述往外層敘述跳躍時才可使用。

```
{   if(條件判斷式)
      {  goto  標籤位置;  }  //if()內層敘述的跳躍至外層敘述標籤位置指令
      ……;          //此區段命令或算式將無法執行
      標籤位置:        //指定跳躍的位置，必須加上「:」符號
}                    //外層敘述結尾
```

```
void  DATA  (unsigned  char  d)
{  unsigned  char  a,b,c
   for ( a=0 ; a<10 ; a++)                  //第一層迴圈
     for ( b=0 ; b<10 ; b++)                //第二層迴圈
       for ( c=0 ; c<10 ; c++)              //第三層迴圈
       {  ……;                              //命令或運算式
          if (d>=100)
          {  goto  list;  }                 //離開所有迴圈跳躍至標籤位置
       }
   ……;                                     //此行命令可能因為 goto 指令跳躍而無法執行
   list:                                    //標籤位置
   ……;                                     //命令或運算式
}
```

```
void  DATA  (unsigned  char  a)
{  ……;                                     //此命令或運算式只執行一次
   list:                                    //標籤位置
   ……;                                     //命令或運算式
   if ( a>=100 )
   {  return;}                              //如果條件式成立則離開迴圈
```

```
   goto   list;                          //跳躍返回標籤位置
}
```

※goto 指令跳躍的範圍僅限於函式內使用，不可以從 A()函式的敘述跳躍至 B()函式的敘述內執行命令。

※goto 指令將破壞程式結構，使得程式更難閱讀及維護，而所有的程式其實都可以在不使用 goto 指令下完成，所以，建議撰寫程式還是應該避開使用 goto 指令。

3.5.9.2 break 強制離開迴圈指令

當某一個迴圈在執行時，可能因為錯誤的敘述指令而形成無窮迴圈時，則需要撰寫 break 強制離開迴圈指令讓程式得以繼續往下執行，另一種使用方法是撰寫 switch 敘述指令時，為了讓條件成立的 case 敘述執行完畢後即可離開敘述指令，而撰寫 break 離開迴圈指令(請參考 3.5.2.1 switch()… case 多向條件敘述式)。

```
unsigned   char   a;
while (P1==0xFF)                 //當 P1 輸入值等於 0xFF 時條件成立
{    for ( a=0 ; a<500 ; a++)    //執行 500 次的 for()迴圈
     {   ……;                     //命令或運算式
         if (P2==0xFF)           //當條件成立時執行以下命令
         {   break;   } }        //強制離開 if()單一敘述所屬的 for()迴圈，無論條件是否成立
   ……;
}                                //因為離開了無窮迴圈，所以此區段程式將被執行
```

```
                                 //當 P1 輸入值等於 0xFF 時條件成立
while (P1==0xFF)                 //條件永遠成立的無窮迴圈
{    while (1)                   //此命令或運算式只執行一次
     {   ……;                     //強制離開無窮迴圈
         break; }
   ……;
}                                //因為離開了無窮迴圈，所以此區段程式將被執行
```

```
unsigned   char   a=0 , b=30 , c=5, d=0x00 ;
If (!a)    //如果 a 等於 0 條件成立
{   if (b>20)   //如果 b 大於 20 條件成立
    {   if (c == 5)   //如果 c 等於 5
        {   while (!(d&0xFF))   //如果 d AND 0xFF 等於 0
            {   ……;             //無論迴圈條件是否永遠成立，此命令或運算式只執行一次
                break;           //強制離開迴圈
}
        ……;                     //離開迴圈後將從此行開始執行
} } }
```

※1 次 break 離開迴圈指令只能跳離 1 層迴圈，假設位於巢狀式迴圈內的最內層宣告，也僅離開最內層到次級迴圈，並非離開整個巢狀式迴圈的敘述。

※除了 switch()… case 多向敘述式語法要求外，其它如 if()敘述內宣告 break 指令，if()敘述式的敘述括弧並不屬於迴圈的層次(如以下範例所示)，不算跳離迴圈層次。

3.5.9.3 continue 重複迴圈指令

coutinue 重複迴圈指令的意義在於「強制掠過該次迴圈的重複，直接跳到下一個重複」，其用法為當迴圈敘述執行過程中，如果遇到 coutinue 重複迴圈指令，將省略指令以後的所有命令與運算，直接回到迴圈的起點重新判斷條件是否繼續執行迴圈。

※當程式回到迴圈的判斷式重新判斷條件，如果條件不成立則會直接離開迴圈，所以重複迴圈指令「coutinue」並沒有造成無窮迴圈的疑慮。

```
unsigned   char   a;
for ( a=0 ; a<100 ; a++)
{   ……;                                  //命令或運算式
    continue;                             //回到迴圈判斷式重新判斷條件
    ……;                                  //此行命令或運算式將永遠不執行
}
```

```
unsigned   char   a , b=0xAA , c=0xBB;
for ( a=0 ; a<100 ; a++)
{   ……;                                  //命令或運算式
    if ( P1==b)                           //當條件成立時執行以下命令
    {   continue;   }                     //回到迴圈判斷式重新判斷條件
    ……;                                  //此行命令或運算式可能不執行
}
```

3.5.9.4 return 離開函式指令

一般來說，當主函式或副函式敘述執行完畢後即會離開函式，並不需要宣告 return 離開函式指令，但是，return 語法卻讓程式寫作有更靈活的運用方式，當函式內具有多向敘述或多層次的迴圈層次時，可以在適當的地方宣告 return 指令，選擇某部分程式並不需要執行而離開函式，或是迴傳資料予呼叫副函式的函式，所以，通常建議撰寫副函式時都應該將離開函式指令宣告在敘述的最末端，不僅避免程式發生錯誤，也提醒程式設計者，這個指令的存在可以靈活運用(請參考 3.5.2 return 回傳命令)。

3.5.10 巢狀式條件敘述

巢狀式條件敘述常用於當程式判斷需要「兩個以上的選擇條件時」或是「多重條件且程序化執行時」使用，當然，如果僅需要兩個以上的選擇條件，是可以使用下例來執行:

```
unsigned   char   a , b , c;                  //宣告字元變數 a , b , c
if( (a>5) && (b!=0) && (c<=10) )   //當所有條件都成立時，才執行下列敘述
{   ……; }                                    //執行命令或運算
```

但是，上述的判斷條件卻不可以指定如果僅成立其中一項，可以先做哪個部分指令，或是當其中一項成立時，其它兩項(或多項)才執行判斷，所以，我們會需要巢狀式迴圈來執行，而且，巢狀式條件敘述並未規定每一層的選擇應該是何種結構，可以讓我們依不同的需求來撰寫程式。

以下第一個格式僅使用 if()敘述來完成 4 層的巢狀式條件敘述，其中，第 3 及第 4 層敘述代表著第 3 與第 4 層的條件成立有時間上的差異，所以，必須使用兩層敘述將不同時間成立的條件分開判斷，這是一種程式寫作的技巧。

```
if (條件運算式)      //巢狀式敘述第 1 層
{   ……;              //執行命令或運算式
    if (條件運算式)      //巢狀式敘述第 2 層
    {    ……;           //執行命令或運算式
         if (條件運算式)      //巢狀式敘述第 3 層
         {    delay() ;       //進入敘述後先延遲一段時間不動作
              if (條件運算式)    //巢狀式敘述第 4 層
              {   ……;   }      //執行命令或運算式
         }
    }
```

```
  }
```

以下第二個格式使用多種敘述結構來完成巢狀式條件敘述，其中，當第一層 if()巢狀式敘述如果條件成立，則 else if()該層巢狀式敘述將忽略，其意義與 if()敘述結構相同。

```
  if (條件運算式)              //巢狀式敘述第 1 層
  {   ……;                     //執行命令或運算式
      switch (條件運算式)      //巢狀式敘述第 2 層 switch()敘述式
  {   case   條件值:           //第一判斷式
          ……;   break ;       //執行命令或運算式
          case   條件值:       //第二判斷式
          ……;   break;        //執行命令或運算式
      }                        //第二層 switch()敘述式的敘述結尾
  }                            //if 敘述式的敘述結尾
  else   if (條件運算式)       //巢狀式敘述第 1 層
  {     if (條件運算式)        //巢狀式敘述第 2 層 if()敘述式
        {……;   }              //執行命令或運算式
  }                            //else   if 敘述式的敘述結尾
```

以下第三個格式綜合使用敘述及迴圈結構來完成巢狀式條件敘述，但必須注意的是，敘述括弧內的指令是否有重複執行的可能性存在。

```
  while (條件運算式)           //巢狀式敘述第 1 層為迴圈
  {   ……;                     //此行命令將於每次第 2 層 if()敘述式執行離開後再執行一次
      if (條件運算式)          //巢狀式敘述第 2 層 if()敘述式
      {……;   }                //執行命令或運算式
  }
```

※if()、switch()…case 條件式敘述執行完畢後將離開敘述「{ }」，而 while()、for()迴圈必須等待條件不成立才會離開敘述「{ }」，撰寫時必須注意這個意義。

※當程式離開該層敘述「{ }」進入內層或外層敘述「{ }」時，程式將從敘述的第一行從新執行，如果程式停滯在迴圈的敘述末端等待條件不成立，該迴圈的敘述並不會重複執行。

3.6 C/C++程式語言結構化設計

3.6.1 C/C++語言的進階型態

c/c++語言允許程式設計師自定資料型態,也將使得程式得以多種組合型態來建構出更適用於程式邏輯的資料型態，而 keil C (keil uVision)的編譯軟體支援結構化定義 struct 及 union 兩種結構型態,但是,物件導向程式設計(class)並不在編譯軟體適用範圍內,所以,關於物件導向的語法格式請參閱軟體 c/c++程式設計相關書籍，本書將不在贅述。

3.6.2 typedef 型態定義

c/c++的基本資料型態區分其實並不精細,就數值方面僅粗分為整數與浮點數兩種基本資料型態，而在實際應用上，經常是需要更具意義的名字來命名，以提升程式的可閱讀性及維護上的便利性，而型態定義的功能便在於「讓程式設計師可以依據個人寫作的習慣與適應性，重新定義宣告命令文法與格式」,而宣告的命令格式如以下所示。

typedef 資料型態 識別字;

雖然前置作業處理指令(#define)也具有同樣的功能，但是，使用 typedef 型態定義卻可以在程式過程中的任何地方「重新定義所需要的資料型態」,這是前置作業處理指令所不能夠的，以下是幾種慣用的宣告型態範例。

```
# define  uint  unsigned  int        //定義 unsigned  int 的巨集名稱為 unit
typedef  char  A;                    //型態定義 char 字元型態識別字為 A
typedef  code  unsigned  int  B;     //型態定義 unsigned  int 整數型態識別字為 B
typedef  uint  C;                    //型態定義巨集 uint 識別字為 C
uint  I ;                            //宣告無號數的整數變數 i
A  j ;                               //宣告有號數的字元變數 j
B  k ;                               //宣告無號數的整數變數 k 且存放於程式記憶體中
C  l ;                               //宣告無號數的整數變數 l
```

※資料型態是識別字所對應的真實資料型態，所以，資料型態可以是 C 語言的基本資料型態，或其它已經定義(#define)過的自訂資料型態。

※識別字一但經過型態定義，在程式中便可取代資料型態的原型來使用，但是，型態定義並不是取代原型資料型態的功能，所以該原型資料型態仍具有原本功能且屬於保留字不可重複。

3.6.3 struct 結構體

所謂結構體的定義，是將一群有關聯性的型態變數或函式，經過結構宣告後有規則的集合再同一個結構體內，可以假設結構體是一種「機具」,經過為這個機具「命名」並宣告該儀器裡具有多少「不同型態」或「連結週邊介面」的零組件後，訂定有多少「使用者」且這些使用者將擁有各自的機具可以使用，這樣的意義在程式裡便稱之為結構體，如此，在一個複雜且攏長的程式系統裡便出現了多個的群組化，對於後續的撰寫或維護增加了相當的便利性與結合性。而結構體的命令格式如以下所示，主要宣告型態共有兩種。

●第一種型態是先宣告完成結構體後，在適當的時機另行宣告結構體變數，當然也可以同時宣告結構變數使用。

```
struct  結構體名稱
{  修飾字  資料型態  變數名稱;     //敘述括弧內總稱為結構主體
   … …   …. …     … …    ;  };
struct  結構體名稱  結構體變數名稱;
```

或是在宣告結構體的結構變數同時直接宣告初值。

```
struct  結構體名稱
{  修飾字  資料型態  變數名稱 1;     //敘述括弧內總稱為結構主體
   修飾字  資料型態  變數名稱 2;
   … …   …. …     … …    ;  };
struct  結構體名稱  結構體變數名稱 =
{  變數 1 初值 , 變數 2 初值 , 變數…初值  };
```

●第二種型態是宣告完成結構體後同時宣告結構變數使用。

```
struct  結構體名稱
{  修飾字  資料型態  變數名稱;     //敘述括弧內總稱為結構主體
   … …   …. …     … …    ;
}結構體變數名稱;
```

或是在宣告結構體的結構變數同時直接宣告初值。

```
struct  結構體名稱
{  修飾字  資料型態  變數名稱 1;     //敘述括弧內總稱為結構主體
   修飾字  資料型態  變數名稱 2;
   … …   …. …     … …    ;
};結構體變數名稱 =
{  變數 1 初值 , 變數 2 初值 , 變數…初值  };
```

```
struct  ABC                     //宣告結構體，結構體名稱為 ABC
{  unsigned  int  a;            //宣告結構主體含有一個整數變數 a
   unsigned  char  b;           //宣告結構主體含有一個字元變數 b
   unsigned  long  int  c;      //宣告結構主體含有一個長整數變數 c
   unsigned  char  d[6];        //宣告結構主體含有一個一維陣列 d 且具有 6 個陣列元素
}Principal;                     //宣告結構變數名稱 Principal
struct  ABC  A[3];              //宣告結構變數名稱 Teacher []陣列，且具有 3 個元素
```

在宣告結構體的範例格式中，n 代表著 8bit 單晶片的記憶體位置，則此結構體共佔有 39 個位元組空間。

記憶體位置	n	n+2	n+3	n+7	n+8	n+9	n+10	n+11	n+12
BYTE	2	1	4	1	1	1	1	1	1
指標位置	k	k+1	k+2	k+3	k+4	k+5	k+6	k+7	k+8
結構變數	A[0].a	A[0].b	A[0].c	A[0].d[0]	A[0].d[1]	A[0].d[2]	A[0].d[3]	A[0].d[4]	A[0].d[5]
記憶體位置	n+13	n+15	n+16	n+20	n+21	n+22	n+23	n+24	n+25
BYTE	2	1	4	1	1	1	1	1	1
指標位置	k+9	k+10	k+11	k+12	k+13	k+14	k+15	k+16	k+17
結構變數	A[1].a	A[1].b	A[1].c	A[1].d[0]	A[1].d[1]	A[1].d[2]	A[1].d[3]	A[1].d[4]	A[1].d[5]
記憶體位置	n+26	n+28	n+29	n+33	n+34	n+35	n+36	n+37	n+38
BYTE	2	1	4	1	1	1	1	1	1
指標位置	k+18	k+19	k+20	k+21	k+22	k+23	k+24	k+25	k+26
結構變數	A[2].a	A[2].b	A[2].c	A[2].d[0]	A[2].d[1]	A[2].d[2]	A[2].d[3]	A[2].d[4]	A[2].d[5]

而撰寫結構主體與一般變數宣告相同的，可以在宣告結構變數時相對賦予結構主體初始值，以下範例的結構體排列位置順序，雖然主體結構變數 a 是整數型態佔有 2 個位元組，

但是在結構體僅屬於 1 個位置空間。

```
struct   ABC                    //宣告結構體，結構體名稱為 ABC
{  unsigned   int   a;          //宣告結構主體含有一個整數變數 a
   unsigned   char   b;         //宣告結構主體含有一個字元變數 b
   unsigned   char   c[10];     //宣告結構主體含有一個一維陣列 c 且具有 10 個陣列元素
};
struct   ABC   Student=         //宣告新的結構變數名稱 Student 並賦予結構主體初始值
{  10;                          //a=10
   20;                          //b=20
   30;  31;  32;  33;  34;      //c[0]=30 , c[1]=31 , c[2]=32 , c[3]=33 , c[4]=34
   35;  36;  37;  38;  39;      //c[5]=35 , c[6]=36 , c[7]=37 , c[8]=38 , c[9]=39
};
void   main(void)
{  Student.a=100                //在函式或迴圈敘述內改變單一的結構變數值
}
```

以下範例為兩種不同的結構型態宣告合併使用方式，每個結構變數佔有 13 個位元組的記憶體空間，總共三個結構變數佔用 39 個位元組的記憶體空間。

```
#include    <AT89x51.h>         //編譯的程式包含 reg51.h 所引述的資料
struct   ABC                    //宣告結構體，結構體名稱為 ABC
{  unsigned   int   a;          //宣告結構主體含有一個整數變數 a
   unsigned   char   b;         //宣告結構主體含有一個字元變數 b
   unsigned   char   c[10];     //宣告結構主體含有一個一維陣列 c 且具有 10 個陣列元素
}Principal;                     //宣告結構變數名稱 Principal
……;                            //執行命令或運算式
struct   ABC   Student;         //宣告新的結構變數名稱 Student
void   main (void)              //主函式
{  Principal.a=5;               //結構變數 Principal 的主體變數 a=5
   Principal.c[1]=50;           //結構變數 Principal 的一維陣列 c[1]=50
   Student.a = 10;              //結構變數 Student 的主體變數 a=10
   Student.b = 30;              //結構變數 Student 的主體變數 b=30
   Student.c[0] = 0xFF;         //結構變數 Student 的一維陣列 c[0]=0xFF
   ……;                          //執行命令或運算式
   struct   ABC   Teacher;      //宣告新的結構變數名稱 Teacher
   while(1)                     //無窮迴圈
   {  Teacher.a = 50;           //結構變數 Teacher 的主體變數 a=50
      Teacher.b = 100;          //結構變數 Teacher 的主體變數 b=100
}  }
```

如果結構體的具有多個結構變數，且在程式中是有相關聯性的，程式撰寫最有效率的方式是宣告一個結構變數陣列來運用，可以將上述程式改寫如下

```
#include    <AT89x51.h>         //編譯的程式包含 reg51.h 所引述的資料
struct   ABC                    //宣告結構體，結構體名稱為 ABC
{  unsigned   int   a;          //宣告結構主體含有一個整數變數 a
   unsigned   char   b;         //宣告結構主體含有一個字元變數 b
   unsigned   char   c[10];     //宣告結構主體含有一個一維陣列 c 且具有 10 個陣列元素
};
……;                            //執行命令或運算式
struct   ABC   Student[3];      //宣告結構變數名稱 Student[]陣列，且具有 3 個元素
void   main (void)              //主函式
{  Student[0].a=5;              //結構變數 Student[0]的主體變數 a=5
   Student[0].c[1]=50;          //結構變數 Student[0]的一維陣列 c[1]=50
   Student[1].a = 10;           //結構變數 Student[1]的主體變數 a=10
   Student[1].b = 30;           //結構變數 Student[1]的主體變數 b=30
   Student[1].c[0] = 0xFF;      //結構變數 Student[1]的一維陣列 c[0]=0xFF
   ……;                          //執行命令或運算式
   struct   ABC   Teacher;      //宣告新的結構變數名稱 Teacher
   while(1)                     //無窮迴圈
```

```
    {   Student[2].a = 50;          //結構變數 Student[2]的主體變數 a=50
        Student[2].b = 100;         //結構變數 Student[2]的主體變數 b=100
}   }
```

※struct 結構主體在函式外層與函式內的資料設定方式並不相同，一般來說，賦予初始值的宣告在函式內外均適用，但是單一結構變數的資料設定則僅可於函式內執行。

※各種程式編譯器支援的宣告格式一般均適用以上兩種型態的語法，但其它的編譯器或許有更便利或縮減的宣告方式，請另行參閱。

※結構體的視野與變數相同，當結構體在函式以外宣告則為全域變數等級，結構體「變數名稱」及「結構名稱」不可以被重複宣告，反之，當結構體在函式內被宣告時為區域變數等級，除了宣告的函式以外的其它函式是不可以使用的。

※結構體的宣告與變數宣告相同，均須於函式敘述的開端宣告，否則將造成編譯錯誤。

3.6.4 struct 結構體與函式

函式的引數也可以是結構體變數，而功能上可以是傳值呼叫與傳送指標(位址)兩種型態，函式的回傳也相同具有回傳結構的功能，使用目的就是將數據資料帶入函式中執行數據運算或命令傳輸，在決定是否將運算後的結構變數回傳給呼叫函式的結構變數使用，應用性質及目的與一般函式運用相同，

3.6.4.1 傳入結構體引數(傳值呼叫)

函式呼叫時具有傳入基本資料的功能(詳細介紹如 3.5 函式與流程控制設計)，而結構體可視為一種新的資料結構型態，因此，程式撰寫自然也可以將結構體引入函式中加以運用，但是將整個結構引入函式中所需耗費的指令週期相較的要比引入指標來的費時，且隨著結構體的複雜程度而增加。

```
struct   結構體名稱
{   修飾字   資料型態   變數名稱;      //敘述括弧內總稱為結構主體
    … …    …. …       … …    ;
}結構體變數名稱;
回傳值   函式名稱   (struct   結構體名稱   結構體變數名稱)
{}
```

```
#include    <AT89x51.h>                 //編譯的程式包含 reg51.h 所引述的資料
struct   ABC                            //宣告結構體，結構體名稱為 ABC
{   unsigned   int   a;                 //宣告結構主體含有一個整數變數 a
    unsigned   char   b;                //宣告結構主體含有一個字元變數 b
    unsigned   char   c[3];             //宣告結構主體含有一個一維陣列 c 且具有 10 個陣列元素
}Principal;                             //宣告結構變數名稱 Principal
struct   ABC   Teacher[3];              //宣告結構變數名稱 Teacher []陣列，且具有 3 個元素
void   DATA   (struct   ABC);           //宣告程式中具有一個引入 ABC 結構體的副函式
void   main   (void)
{   Principal.a = 100;                  //結構變數 Principal 的主體變數 a=100
    Principal.b = 50;                   //結構變數 Principal 的主體變數 b=50
    Principal.c[0] = 0xFF;              //結構變數 Principal 的一維陣列 c[0]=0xFF
    Teacher[0].a = 100;                 //結構變數 Teacher[0]的主體變數 a=100
    Teacher[0].b = 50;                  //  結構變數 Teacher[0]的主體變數 b=50
    Teacher[0].c[0] = 0xFF;             //結構變數 Teacher[0]的一維陣列 c[0]=0xFF
    DATA(Principal);                    //呼叫結構體副函式，並將 Principal 引入運算
    DATA(Teacher[0]);                   //呼叫結構體副函式，並將 Teacher[0]引入運算
}
```

```
void  DATA  (struct  ABC  Student)      //宣告一個有引入結構體的副函式
{  umsigned  int  a,b;                  //宣告整數變數 a,b
   a = Student.a- Student.b;            //a=100-50
   b = Student.c[0];                    //b=0xFF
}
```

※當呼叫引入結構的函式時，如果引入的是「結構變數陣列」，其它編譯器可能僅輸入陣列名稱即以第一維第一個元素代表位置，但在 keil C 編譯器中必須將陣列元素寫入。

3.6.4.2 傳入結構體指標(傳送指標呼叫)

既然結構體是一個規則性儲存資料的結構，當然可以在呼叫函式時將結構體的指標引入，取代將整個結構體引入函式的費時運作，但是相對的指向結構體的指標位置就必須清楚宣告時的順序位置問題，否則仍將產生錯誤。

```
struct  結構體名稱
{  修飾字  資料型態  變數名稱;     //敘述括弧內總稱為結構主體
   … …   …. …      … …   ;
}結構體變數名稱;
回傳值  函式名稱  (struct  結構體名稱  *結構體變數名稱)
{}
```

```
#include   <AT89x51.h>                  //編譯的程式包含 reg51.h 所引述的資料
struct  ABC                             //宣告結構體，結構體名稱為 ABC
{  unsigned  int  a;                    //宣告結構主體含有一個整數變數 a
   unsigned  char  b;                   //宣告結構主體含有一個字元變數 b
   unsigned  char  c[10];               //宣告結構主體含有一個一維陣列 c 且具有 10 個陣列元素
}Principal;                             //宣告結構變數名稱 Principal
struct  ABC  Teacher[3];                //宣告結構變數名稱 Teacher []陣列，且具有 3 個元素
void  DATA  (struct  ABC)               //宣告程式中具有一個引入 ABC 結構體的副函式
void  main  (void)
{  umsigned  char  i;
   Principal.a = 100;                   //結構變數 Principal 的主體變數 a=100
   Principal.b = 50;                    //結構變數 Principal 的主體變數 b=50
   for( i=0 ; i<10 ; i++ )
   {  Principal.c[i] = i+1;}            //結構變數 Principal 的一維陣列 c[0]=1…. c[9]=10
   Teacher[0].a = 100;                  //結構變數 Teacher[0]的主體變數 a=100
   Teacher[0].b = 50;                   //結構變數 Teacher[0]的主體變數 b=50
   for( i=0 ; i<10 ; i++ )
   {  Teacher[0].c[i] = i+1;}           //結構變數 Teacher[0]的一維陣列 c[0]=1…. c[9]=10
/* 將結構變數引入結構體副函式，函式中的變數 c 將等於 100+50+1+2+3+….+10 = 205 */
   DATA(Principal);                     //呼叫結構體副函式，並將 Principal 引入運算
   DATA(Teacher[0]);                    //呼叫結構體副函式，並將 Teacher[0]引入運算
}
void  DATA  (struct  ABC  *Student)     //宣告一個有引入指標結構體的副函式
{  umsigned  int  a,b,c=0;              //宣告整數變數 a,b,c
   for( a=0 ; a<12 ; a++)               //執行 12 次的迴圈
   {  b = *Student;                     //變數 b 的內容等於指標位置內容
      c += a ;                          //c=c+a
      Student++;                        //指標位置+1
   }
}
```

※引入「結構變數陣列」在 keil C 編譯器中仍必須將陣列元素寫入。

3.6.4.3 結構體的回傳值

既然可以在函式引入結構體執行運算，當然也可以回傳結構體予呼叫該結構體的函式運作，將函式內結構變數內容指向給呼叫該函式的結構變數。

```
struct  結構體名稱
{  修飾字  資料型態  變數名稱;      //敘述括弧內總稱為結構主體
    … …   …. …      … …   ;
}結構體變數名稱;
struct  結構體名稱  函式名稱  (引入值)
{  struct  結構體名稱  結構變數;
    ……;          //執行命令或運算式
    return  結構變數;
}
```

```
#include   <AT89x51.h>                  //編譯的程式包含 reg51.h 所引述的資料
struct  ABC                             //宣告結構體，結構體名稱為 ABC
{  unsigned  int  a;                    //宣告結構主體含有一個整數變數 a
   unsigned  char  b;                   //宣告結構主體含有一個字元變數 b
   unsigned  char  c[10];               //宣告結構主體含有一個一維陣列 c 且具有 10 個陣列元素
}Principal;                             //宣告結構變數名稱 Principal
struct  ABC  DATA  (void)               //宣告程式中具有一個引入 ABC 結構體的副函式
/* 將 DATA()副函式的回傳結構指向 Principal 結構變數*/
void  main  (void)
{
   Principal =DATA();                   // Principal 結構變數內容等於回傳結構的值
}
struct  ABC  DATA  (void)               //宣告一個有回傳結構體的副函式
{  struct  ABC  Teacher;                //宣告新的結構變數名稱 Principal
   Teacher.a = 100;                     //結構變數 Teacher 的主體變數 a=100
   Teacher.b = 50;                      //結構變數 Teacher 的主體變數 b=50
   for( i=0 ; i<10 ; i++ )
   {  Teacher[0].c[i] = i+1;}           //結構變數 Teacher[0]的一維陣列 c[0]=1…. c[9]=10
   return  Teacher;                     //回傳 Teacher 結構變數的所有值
}
```

※結構體必須是相同的「結構體名稱」的結構變數才可以回傳，否則編譯過程將產生錯誤。
※相同於變數回傳，結構體仍必須使用 return 指令將結構變數回傳。

3.6.5 struct 結構體與 typedef

型態定義 typedef 同樣可以使用在結構體的定義上，當程式設計者大量的使用結構化定義時，通常冀望程式能更趨於簡短且解讀容易，此時使用型態定義來縮減命令的語法格式長度是相當合理的方法之一。而一般型態定義語法可區分為前置定義與後置定義兩種型態。

3.6.5.1 前置定義型態

前置定義型態是指在宣告結構體的同時，將結構體的命令語法直接作型態定義，所以當完成前置定義後，即可使用定義的型態來宣告結構變數。

```
typedef  struct  結構體名稱
{  修飾字  資料型態  變數名稱;      //敘述括弧內總稱為結構主體
    … …   …. …      … …   ;
}結構體名稱;
```

```
/* 宣告結構變數 */
結構體名稱  結構變數;
```

以下範例為周邊系統或通訊協定的氾用型結構體型態定義宣告。

```
#include    <AT89x51.h>                 //編譯的程式包含 reg51.h 所引述的資料
#define   RFConfig_    0x30             //定義某一個週邊設備的通訊協定第一個 byte 為 0x30
#define   RFTX    _    0x20             //定義某一個週邊設備的通訊協定第一個 byte 為 0x20
#define   RFRX    _    0x10             //定義某一個週邊設備的通訊協定第一個 byte 為 0x10
typedef   struct   ABC                  //宣告結構體，結構體名稱為 ABC
{  unsigned   int   a;                  //宣告結構主體含有一個整數變數 a
   unsigned   char   b;                 //宣告結構主體含有一個字元變數 b
   unsigned   char   c;                 //宣告結構主體含有一個字元變數 c
   unsigned   char   data[10];          //宣告結構主體含有一個一維陣列 data 且具有 10 個陣列元素
} ABC;
ABC   RFMOD =                           //宣告結構變數名稱 RFMOD
{  RFConfig , RFTX , RFRX,              //結構主體 a=0x30 , b=0x20 , c=0x10
   0x01 , 0x02 , 0x03 , 0x04 , 0x05,    //結構主體 data[0]=0x01… data[9]=0xA0
   0x06 , 0x07 , 0x08 , 0x09 , 0xA0
};
```

3.6.5.2 後置定義型態

後置定義型態則與一般的型態定義格式相仿，將結構體的命令語法以另一個可辨識的文字取代使用，所以當完成前置定義後，即可使用定義的型態來宣告結構變數。

```
struct   結構體名稱
{  修飾字   資料型態   變數名稱;      //敘述括弧內總稱為結構主體
    … …    …. …      … …    ;  };
typedef   struct   結構體名稱   識別字;
/* 宣告結構變數 */
識別字   結構變數;
```

以下範例為修改前置定義型態後的宣告格式。

```
#include    <AT89x51.h>                 //編譯的程式包含 reg51.h 所引述的資料
#define   RFConfig_    0x30             //定義某一個週邊設備的通訊協定第一個 byte 為 0x30
#define   RFTX    _    0x20             //定義某一個週邊設備的通訊協定第一個 byte 為 0x20
#define   RFRX    _    0x10             //定義某一個週邊設備的通訊協定第一個 byte 為 0x10
struct   ABC                            //宣告結構體，結構體名稱為 ABC
{  unsigned   int   a;                  //宣告結構主體含有一個整數變數 a
   unsigned   char   b;                 //宣告結構主體含有一個字元變數 b
   unsigned   char   c;                 //宣告結構主體含有一個字元變數 c
   unsigned   char   data[10];          //宣告結構主體含有一個一維陣列 data 且具有 10 個陣列元素
} ;
typedef   struct   ABC   DEF;           //型態定義 struct   ABC 的識別字為 DEF
DEF   RFMOD =                           //宣告結構變數名稱 RFMOD
{  RFConfig , RFTX , RFRX,              //結構主體 a=0x30 , b=0x20 , c=0x10
   0x01 , 0x02 , 0x03 , 0x04 , 0x05,    //結構主體 data[0]=0x01… data[9]=0xA0
   0x06 , 0x07 , 0x08 , 0x09 , 0xA0
};
```

3.6.6 struct 結構體的位元存取

在一個結構主體除了可以宣告完整的整數、浮點數與陣列結構變數使用之外，如果程式設計中部分數據是固定最大數的、較小的或是連續的但是僅有單一位元的，假設均宣告為最小一個位元組的資料 char 型態，則會浪費許多的記憶體空間，所以，結構體提供了「位元變數」的設定方式，可以用來儲存一個位元以上的資料變數，而除了結構主體的語法

格式稍作修訂以外，其餘適用語法與前述各節用法均相同，宣告的語法如下所示。

```
struct  結構體名稱
{  修飾字  資料型態  變數名稱 : 位元長度;
   修飾字  資料型態  變數名稱 : 位元長度;
};
struct  結構體名稱  結構變數;
```

可以發現在上述的語法中結構主體做了改變，當出現符號「:」代表此變數為「位元變數」型態，而位元長度則以資料型態的最大位元數為界線(例如整數資料型態 int 為 16)，以下為實際宣告的指令範例。

```
#include   <AT89x51.h>          //編譯的程式包含 reg51.h 所引述的資料
struct  ABC                     //宣告結構體，結構體名稱為 ABC
{  unsigned  int  A1 : 1 ;      //宣告結構主體變數 A1 大小為 1bit
   unsigned  int  A2 : 2 ;      //宣告結構主體變數 A2 大小為 2bit
   unsigned  int  A3 : 4 ;      //宣告結構主體變數 A3 大小為 4bit
   unsigned  int  A4 : 8;       //宣告結構主體變數 A4 大小為 8bit
   unsigned  int  B;            //宣告結構主體變數 B
} ;
struct  ABC  student;           //宣告結構變數名稱 student
void  main  (void)
{  student.A1 = 1;              //結構變數 A1 的最大數為 1
   student.A2 = 3;              //結構變數 A1 的最大數為 3
   student.A3 = 15;             //結構變數 A1 的最大數為 15
   student.A4 = 255;            //結構變數 A1 的最大數為 255
   student.B=65535;             //結構變數 B 的最大數為 65535
}
```

在以上的結構體中，記憶體總共佔有 2 個整數變數(int)的位元空間(4Byte)，空間分配如以下列表所示，最後一個位元因為沒有宣告使用，所以被捨棄不用，而指標位置則越過無用的空間延續。

位元數	1	2	4	8	1	16
指標位置	k	k+2	k+3	k+4	無	k+5
結構變數	A1	A2	A3	A4	無	B

※位元變數是以資料型態大小的空間切割分配，當宣告的變數太多超出位元數時則切割第二個資料型態大小的空間延續，但是，如果變數使用的位元數總合低於資料型態的位元數時，則多餘的位元將自動被捨棄不用，避免造成後續資料的定址錯誤，所以，建議將所有「位元變數」排列在結構主體中連續的位置，避免造成多餘的位元空間浪費。

3.6.7 union 聯合結構體

聯合結構體 union 與結構體的語法大致上是相仿的，但是，聯合結構體的結構主體共同享有同一塊記憶體空間，所以，分享記憶空間的總容量是「該結構體中佔最多空間的資料變數」, 這是不同於 struct 結構體的記憶體配置型態。在宣告語法上 union 相同於 struct 結構體，可區分為兩種標準的語法型態。

●第一種型態是先宣告完成結構體後，在適當的時機另行宣告結構體變數。

```
union  結構體名稱
{  結構主體;  };
union  結構體名稱  結構體變數名稱;
```

●第二種型態是宣告完成結構體後同時宣告結構變數使用。

union 結構體名稱
{ 結構主體;
}結構體變數名稱;

以下範例為 union 聯合結構體的綜合應用方法，而聯合結構體 ABC 結構主體內最大的變數資料型態為 long int，所以總共佔用 4byte 的記憶體空間。

```
union   ABC                         //宣告聯合結構體，結構體名稱為 ABC
{   char   first;
    unsigned   char   second;
    unsigned   int   A1 : 1 ;       //宣告結構主體變數 A1 大小為 1bit
    unsigned   int   A2 : 2 ;       //宣告結構主體變數 A2 大小為 2bit
    unsigned   int   A3 : 4 ;       //宣告結構主體變數 A3 大小為 4bit
    unsigned   int   A4 : 8;        //宣告結構主體變數 A4 大小為 8bit
    unsigned   long   int   B;      //宣告結構主體變數 B
} ;
union   ABC   student;              //宣告聯合結構變數名稱 student
```

※由於 union 結構體的結構主體共享同一塊記憶體空間，所以，當改變結構主體的某一變數資料時，相對於是改變所有的變數的數據。

※同一記憶體空間型態對於函式內引入功能，並不適用指標變數指向位置。

※初值宣告僅可宣告一個資料數據，而且以該結構主體內最大的變數資料型態為限制。

3.6.7.1 聯合結構體的初值宣告

因為聯合結構體僅佔用結構主體內最大的資料變數，所以，當聯合結構體需要在宣告同時賦予結構主體初值時，僅可以最大的資料變數為限賦予 1 筆資料。

union 結構體名稱
{ 結構主體; };
union 結構體名稱 結構體變數名稱 =
{ 變數初值 };

```
union   ABC                             //宣告聯合結構體，結構體名稱為 ABC
{   char   first;                       //宣告一個有號數字元變數 first
    unsigned   char   second;           //宣告一個無號數字元變數 second
    unsigned   short   int   B;         //宣告一個無號數短整數變數 B
} ;
union   ABC   student ={   65535   };   //宣告聯合結構變數名稱 student
```

3.6.7.2 函式的傳入結構引數與回傳值

聯合結構體相對於函式的引述資料而言，相同的可以是視一種新的資料型態，一樣可以將結構主體使用引數傳入函式內應用，而相較於 stract 結構體而言，即使定義的變數資料型態一樣繁雜，但是實際上僅有一個記憶體空間可以使用，這是必須注意的。而回傳值的語法功能其實也是與 stract 結構體是相仿的，只要將指令更改為 union 即可，以下語法格式及範例僅以傳入結構解釋。

union 結構體名稱
{ 結構主體; };
union 結構體名稱 結構體變數名稱;
迴傳值 函式名稱 (union 結構體名稱 結構體變數名稱)
{}

```
#include    <AT89x51.h>                  //編譯的程式包含 reg51.h 所引述的資料
union   ABC                              //宣告聯合結構體，結構體名稱為 ABC
{   char   first;                        //宣告一個有號數字元變數 first
    unsigned   char   second;            //宣告一個無號數字元變數 second
    unsigned   short   int   B;          //宣告一個無號數短整數變數 B
} Principal;                             //宣告結構變數名稱 Principal
void   DATA   (union   ABC);             //宣告程式中具有一個引入 ABC 聯合結構體的副函式
void   main   (void)
{   DATA(Principal);   }                 //呼叫結構體副函式，並將 Principal 引入運算
void   DATA   (struct   ABC   Student)   //宣告一個有引入結構體的副函式
{   umsigned   int   a=10;               //宣告整數變數 a
    Student. First = a;                  //令聯合結構體主體結構變數等於 a
}
```