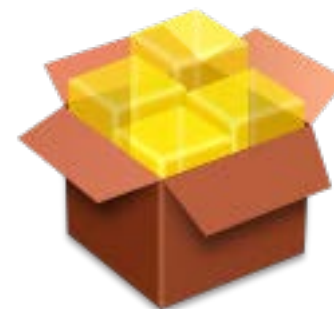


3-3 程序定義和使用



3-1 資料型態

- ➡ 陣列
- ➡ 結構
- ➡ 指標





3-1 資料型態

- ➡ 當我們要利用某個程式語言撰寫一個應用系統的時候，我們必須要將處理的對象，以該程式語言提供的資料型態，適當的定義在程式中。
- ➡ 譬如說，要表示月和日組合起來的日期，如2月1日，可以使用字串表示成「0201」，或是利用整數「32」，來表示是1年的第32天，有的語言甚至直接提供日期型態。



3-1 資料型態

- ➡ 一般來講，高階程式語言都會提供以數字和字串為基礎的資料型態。
- ➡ 數字而言，多分為整數(int)、長整數(long int)、浮點數(float)、雙精準數(double)等，這些型態的差別在於可表示數值資料的大小範圍。
- ➡ 文字方面，有的只能定義一個字元(char)，有的則直接可定義較長的字串(string)。



3-1 資料型態

- ➡ 當我們為一個變數宣告好其資料型態之後，系統就知道應該為該變數保留多少記憶體的空間，而空間的大小會決定該型態可表示的數值範圍。
- ➡ 下表顯示C所支援的資料型態，所需的空間和資料範圍會因為機器的規格而有所不同，此表是以64位元的電腦為例，C語言的long int至少是32bits，也可能是64bits。



C 的資料型態

資料型態	所需空間	資料範圍
char	8 bits	ASCII
int	32 bits	-2147483648 ~ 2147483647
short int	16 bits	-32768 ~ 32767
long int	32 bits	-2147483648 ~ 2147483647
float	32 bits	3.4E-38 ~ 3.4E+38
double	64 bits	1.7E-308 ~ 1.7E+308



3-1 資料型態

- ➡ 為一個變數宣告好資料型態後，編譯器就會檢查該變數在程式任何地方出現的時候，是不是使用恰當。假設我們宣告「x」是一個字元的資料型態，將符號「a」指定給x就是恰當的，但是將x乘以100就是沒有意義的。
- ➡ 基於這些好處，很多高階語言如PASCAL和C語言，都要求在使用一個變數前，必須先宣告它的資料型態。



陣列

- ➡ 當有一系列相同型態的資料想要處理，如全班50個同學的數學成績，就可以使用陣列(array)的資料型態。
- ➡ 以下宣告一個包含50個整數的陣列：

```
int score[50];
```




陣列

- ➡ 陣列的名稱為「score」，陣列裡的每個資料為整數(int)型態，而陣列第一個位置為score[0]，第二個位置為score[1]，依序一直到score[49]，這是因為C語言預設以註標0來表示陣列的第一個元素。
- ➡ 定義了陣列之後，就很容易從這個序列中取出一個特定的資料。



陣列

- ➡ 假設這個陣列是以學生的學號依序建立的，那當我們要取出學號5的同學的成績，我們就可以寫 `score[4]`，而學號20的同學的成績，則可以利用 `score[19]` 取出。



結構

- ➡ 當有一些相關資料，想要聚集成一個單元一起處理，可以使用結構(structure)的資料型態。譬如說，針對一個同學，我們想要表示他的姓名、系別、年級等3種資料，可以宣告如下：

```
struct student {  
    char(6) name;  
    char(10) major;  
    int year;  
};
```



結構

- ➡ 結構的名稱為 **student**，其中欄位 **name** 的資料型態為6個字元(char)，欄位 **major** 的資料型態為10個字元，欄位 **year** 的資料型態為整數。
- ➡ 假設我們之後再宣告變數 **x** 的資料型態為 **student** 結構，如下所示：

```
struct student {  
    char(6) name;  
    char(10) major;  
    int year;  
};
```

```
struct student x;
```



結構

- ➡ 則以後我們可以利用小數點加上欄位名稱，來指出變數 `x` 其中的某一個成分，如 `x.name`，`x.major`，和 `x.year`。
- ➡ 這種表示式可以代表該成分在記憶體的位置，也可回傳該成分目前的值。



指標

- ➡ 指標(pointer)是一種很特殊的資料型態，它記錄的是某個資料在記憶體的位置，也就是它提供了**非直接存取**(indirect accessing)的功能。
- ➡ 那麼為什麼我們不直接處理該資料，而要透過指標呢？通常有以下兩個理由：
 - ▶ 為了效率性的考量。
 - ▶ 我們不能確定資料的大小。



指標

為了效率性的考量

- ➡ 指標記錄一個記憶體的位置，所以其所需的空間是固定的，通常就是一個字元的大小。
- ➡ 假設每一個顧客資料，都是用複雜的結構表示，而每個結構大小為100位元，若是希望對所有的顧客資料做處理，像是依照購買金額排序，則在記憶體內我們必須搬動很多個100位元大小的顧客結構。
- ➡ 另一方面，若使用指標為代理人，則在記憶體內我們只須搬動1個字元大小的指標，則程式執行的效率會有顯著的改善。



指標

我們不能確定資料的大小

- ➡ 假設要記錄所有顧客的資料，其中一個方法是使用陣列，但是宣告陣列時必須很明確的告知陣列內元素的個數，如50或100，以便系統在記憶體裡預留空間。
- ➡ 假設宣告陣列大小為100，但是只來了10個顧客，則有90個元素的空間被浪費了；但是若宣告為50，但是卻來了60個顧客，則事先預留的空間則不夠，造成很大的問題。



指標

網址

<https://www.goodfreephotos.com/astrophotography/milky-way-and-starry-night-sky.jpg.php>

資料



資料名稱

Milky Way and
Starry Night Sky

指向這個資料



指標

記憶體位址

0X0012FF70	15
0X0012FF74	2
0X0012FF78	39
0X0012FF7C	180
0X0012FF80	67
0X0012FF84	...

從此位址連續寫入 4 個 byte
(因為整數型佔 4 個 byte)

跟記憶體要一塊空間

```
void main(){  
    int a = 15;  
    int b = 2;  
    int c = 39;  
    int d = 180;  
    int e = 67;  
}
```



指標

變數三要素

變數位址

0X0012FF74

變數值

2

變數名稱

b

寫入資料的起始位址
(依據不同型別所占 byte 數不同)

把某個變數的位址稱為
「指向該變數的指標」



指標

宣告一個指標變數

* : 表示這個變數是個指標

int *pointer

pointer : 表示這個變數的名稱

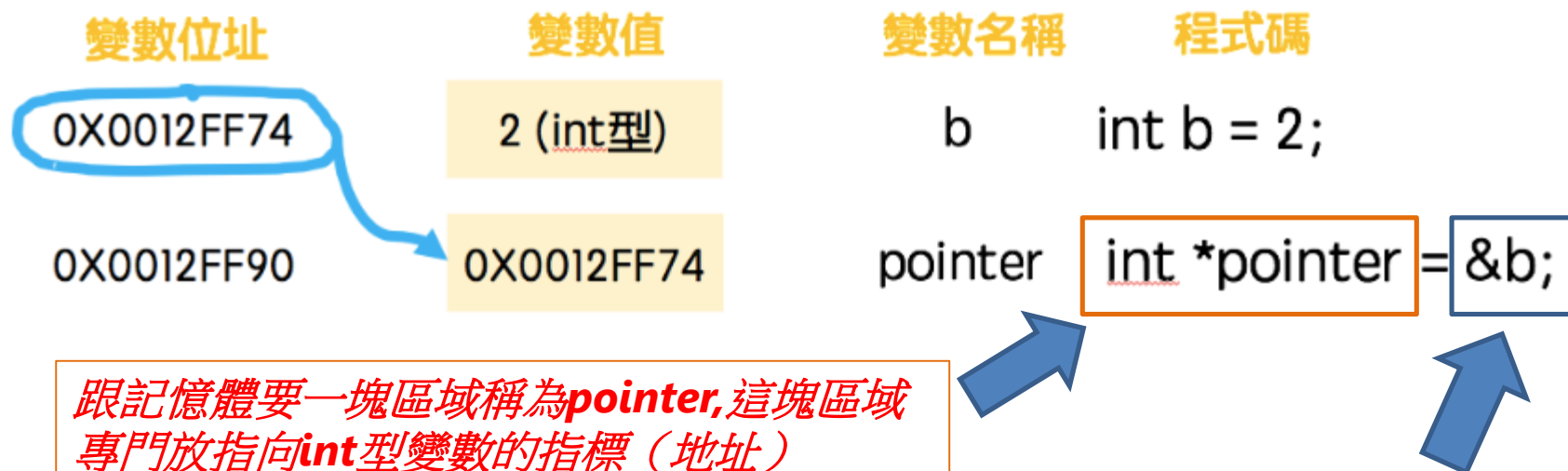
(或寫成 int* pointer 也可以)



指標

那 int 代表什麼？* 不就宣告是指標了嗎？

int 表示 pointer 指向變數的類型



把變數**b**的地址值給**pointer**，注意不能寫成 **pointer = b**



3-2 程式指令

- ➡ 比較 : if
- ➡ 固定次數的迴圈 : for
- ➡ 不固定次數的迴圈 : while
- ➡ 不固定次數的迴圈 : for





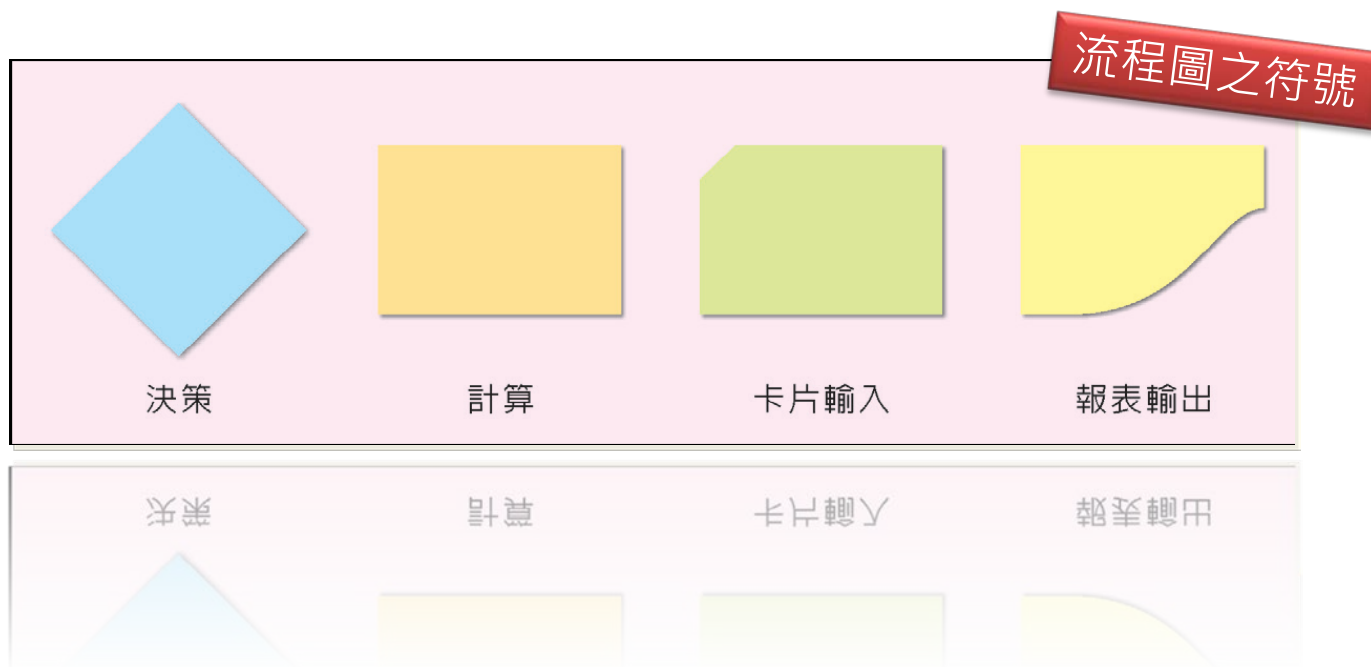
3-2 程式指令

- ➡ 為了清楚的表示邏輯結構和步驟間的關聯，我們常常會使用**流程圖**(flow chart)來輔助說明。
- ➡ 流程圖裡有幾個不同的符號，分別有其意義：
 - ▶ 決策(decision)的運算式是用菱形框表示。
 - ▶ 計算(computation)的敘述式是用長方框表示。
 - ▶ 輸入(input)和輸出(output)有時會以特定機件(device)有關的形狀來表示。



3-2 程式指令

➡ 相關的符號如下圖所示：





比較：if

- ➡ **if**指令提供了邏輯判斷式。
- ➡ 如果**if**後面接的運算式被判斷為真，則程式會繼續執行**then**後面的運算式。
- ➡ 如果**if**後面接的運算式被判斷為不真，且程式設計師提供了其他運算式在**else**之後，則程式會改而執行該運算式，否則就不會有任何動作。



比較：if

- 下面這個範例，在變數*i*的值大於0時，變數*x*的值設定為10，否則變數*y*的值設定為5。

```
C
if (i > 0)
    x = 10;
else
    y = 5;
```



比較：if

- 下面這個C語言的範例，與上例的差別，是在於變數*i*的值小於或等於0時，並不會再進一步執行任何命令，因為我們並沒有提供else子句。

C

```
if (i > 0)
    x = 10;
```

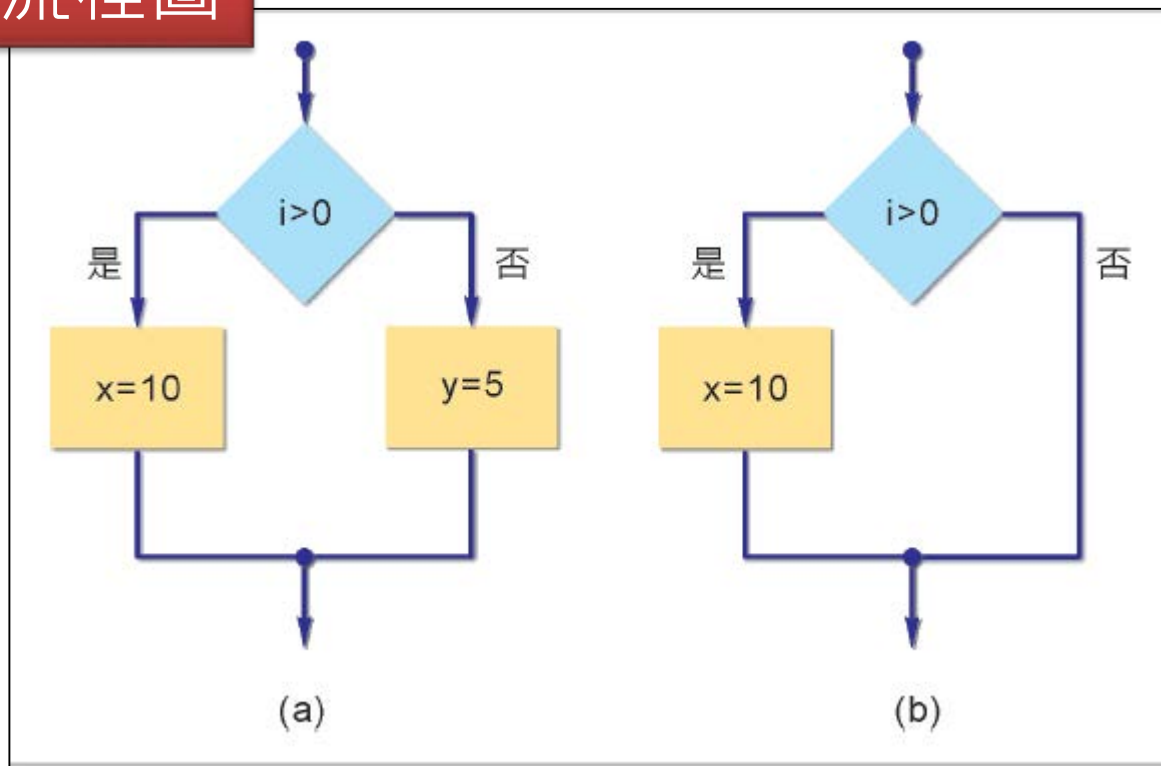


比較：if

- ➡ 寫在if之後的邏輯判斷式，會表示在菱形符號中，然後利用標示為「是」和「否」兩條線，分別指到不同的運算。
- ➡ 為了清楚的表示整個結構，分別利用兩個小圓圈，作為一個虛擬的開始和虛擬的結束。
- ➡ 在圖(a)中，判斷式「 $i > 0$ 」不論是否符合，都會有一個對應的運算；但是在圖(b)中，一旦判斷式不符合，則沒有任何的運算，整個結構直接結束，進入下一個命令。



if結構的流程圖





比較：if

- ➡ 下例顯示了巢狀if(nested if)的寫法，也就是我們可以在then或else的部分，再放入另一個if敘述。
- ➡ 以此例而言，當變數i的值被判斷為正之後，我們需要再確定變數a的值大於變數b的值，才會指定變數x為10。



比較：if

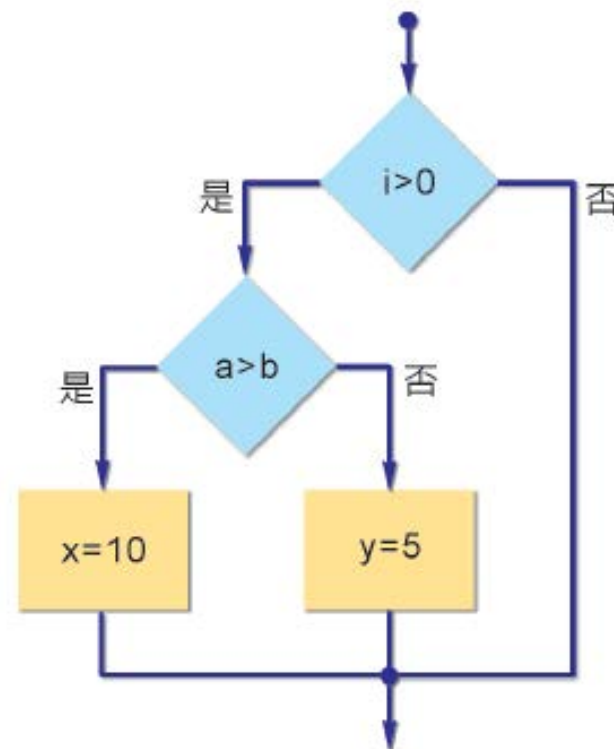
- ▶ 值得注意的是，變數**y**的值會被指定為「5」，是在當變數**i**的值為「正」，且變數**a**的值「不大於」變數**b**的值的值的情況下。

```
C
    if (i > 0)
        if (a > b)
            x = 10;
        else
            y = 5;
```



比較：if

- 在在這裡可以清楚地看出來，一旦判斷式「 $i > 0$ 」不符合，則整個結構沒有任何其他運算，直接結束；但是若判斷式「 $i > 0$ 」為真，則還要再做另一個判斷，亦即是否「 $a > b$ 」，才會決定相對應的動作。



巢狀if結構的流程圖



固定次數的迴圈：for

- ➡ 利用for指令，我們可以事先指定好迴圈的執行次數。
- ➡ 下面這個範例，透過變數i的值將迴圈的執行次數控制為9次。

```
void main()
{
    for(i=1;i==10;i++)
    {
        int data =0;
        data +=i;
    }
}
```



不固定次數的迴圈：while和repeat

- ➡ 所謂的不固定次數，就是迴圈的執行次數，並沒有很明確的在程式裡指定好，至於迴圈要執行幾次，則是利用一個特定的邏輯判斷式。
- ➡ **while**後面是接一個邏輯判斷式，也就是 $i < 6$ ，若是這個邏輯判斷式為真，則程式會進入此迴圈，執行**do**後面的指令，在此例中是更改變數**x**和變數**i**的值。



不固定次數的迴圈：while和repeat

- ➡ 等到這兩個指令執行完後，程式會回到邏輯判斷式，再一次判斷變數*i*的值是否小於6，如此不斷重複，直到變數*i*的值大於6或等於6的時候，才會跳出迴圈。
- ➡ 由於一開始設定變數*i*的值為1，且每跑一次就把變數*i*的值加1，所以此迴圈總共會執行5次；同時，變數*x*的值，會是整數1加到整數5的和。



不固定次數的迴圈：while和repeat

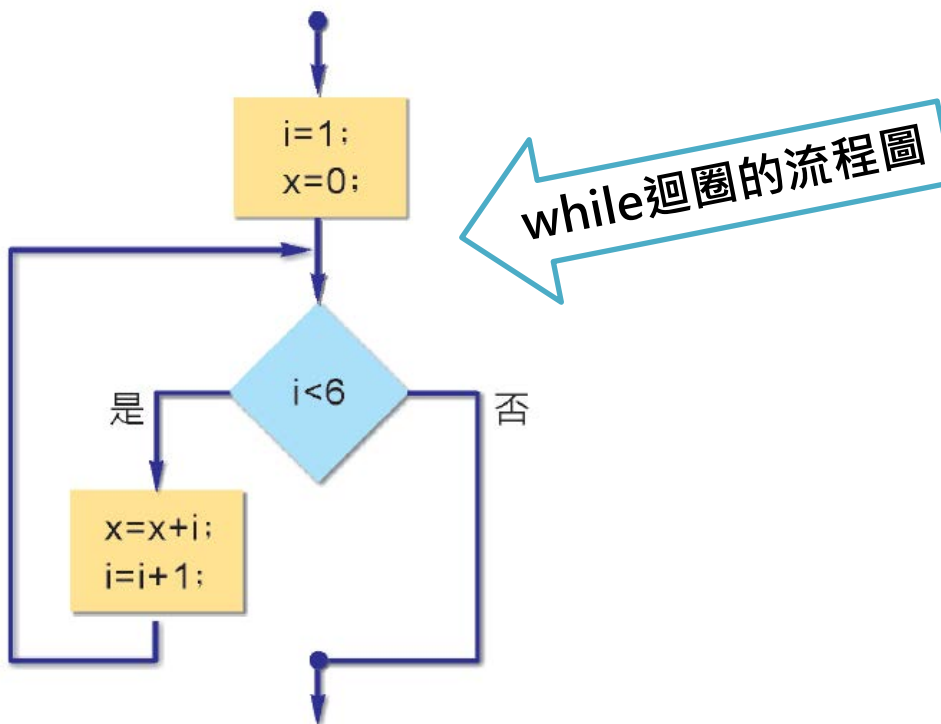
```
C
i = 1; x = 0;
while ( i < 6)
{
    x = x + i;
    i = i + 1;
}
```

- ➡ 進入迴圈之後，要先後執行兩個命令，來依序更改變數*x*和變數*i*的值，所以這兩個命令被關鍵字{和}包起來，被包起來的命令稱作**複合命令** (compound statement)，它可以被視作是一個擁有很多「小」指令的一個「大」指令。



不固定次數的迴圈：while和repeat

- 為了清楚地表示此迴圈代表的邏輯結構和執行順序，我們也將對應的流程圖表示在下圖中。





不固定次數的迴圈：while和repeat

- ➡ 首先，先指定好變數 “i” 和變數 “x” 的值。接著，我們進入邏輯判斷式，若是判斷式不成立，則程式會直接跳出此結構；若是判斷式成立，則會再回到之前邏輯判斷式的位置，根據最新的變數值再重複進行判斷。
- ➡ 若是沒有適當的改變變數值，使得邏輯判斷式的真假值改變，則會再度進入迴圈，甚至造成無窮迴圈的情況，這是撰寫程式時需要注意的地方。



不固定次數的迴圈：while和repeat

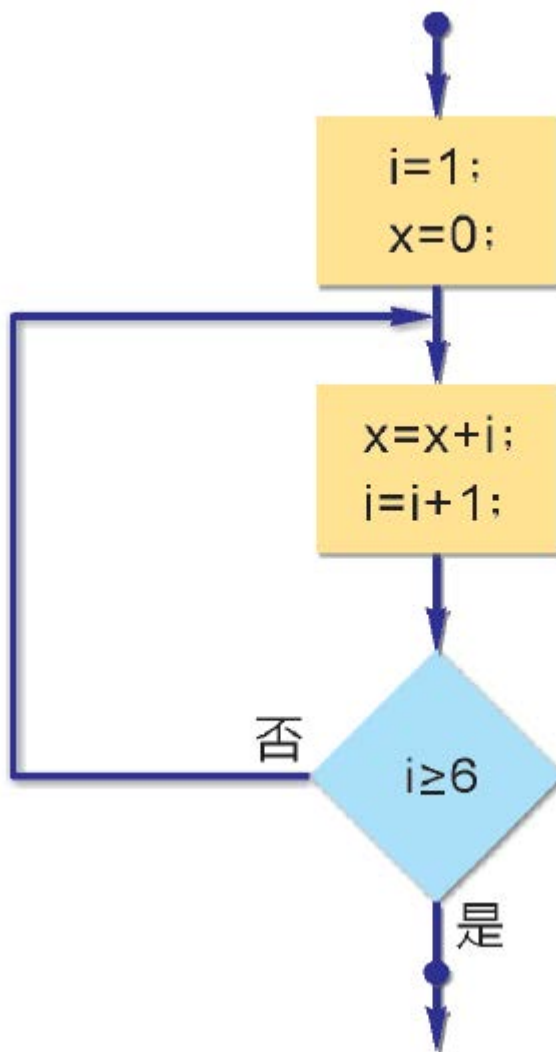
- ➡ 另一種迴圈的寫法，則是不先做判斷，而是直接先執行命令，等到執行完再做邏輯式的判斷。
- ➡ 在C，是利用關鍵字do和while。
- ➡ 先執行命令，再進行邏輯式的判斷，但是，當判斷式為真的時候，do-while的寫法會繼續留在迴圈裡



不固定次數的迴圈：while和repeat

➡ 所以在下例中，同樣是執行迴圈5次

```
C
i = 1; x = 0
do {
    x = x + i;
    i = i + 1;
} while ( i < 6);
```

repeat迴圈的流程圖



不固定次數的迴圈：for

- ➡ for指令後面接著的式子分三部分：
 - ▶ 第一是在執行迴圈之前，所需要先給定的初始值設定。
 - ▶ 第二是進入或留在迴圈的條件，有如while指令後面接著的判斷式。
 - ▶ 第三是在每當要執行下一次迴圈之前，所需要執行的式子。



不固定次數的迴圈：for

➡ 下面列出對應於之前while寫法的for的寫法：

while	for
<pre>i = 1; x = 0; while (i < 6) { x = x + i; i = i + 1; }</pre>	<pre>x = 0; for (i=1; i<6; i=i+1) { x = x + i; }</pre>



不固定次數的迴圈：for

- ➡ 由於控制迴圈執行次數的是變數*i*，所以可以將該變數的初始值、留在迴圈的條件、和每次迴圈更改的方式，都直接列在for指令的後面，如此可以更清楚分辨出迴圈內執行的內容，和迴圈執行的次數。



3-3 程序定義和使用

- ➡ 全域變數vs.局部變數
- ➡ 以值傳遞vs.以位址傳遞





3-3 程序定義和使用

- ➡ 在一個**程式**(program)中，可能會寫出冗長而難以理解的命令，所以大部分的程式語言都提供了**程序**(procedure)或**函數**(function)的定義。
- ➡ 一個程序對應到一段程式碼，稱作程序**本體**(body)，然後也指定一個對應的名稱，稱作程序**名稱**(name)。
- ➡ 等到定義完程序之後，只要利用該名稱**呼叫該程序**(procedure call)，對應的程式碼就會執行。



3-3 程序定義和使用

➡ 程序在定義時，必須提供下列資訊：

程序名稱

程序本體，含變數宣告和命令敘述

正式參數(formal parameter)宣告

程序回傳的資料型態



3-3 程序定義和使用

- 在下例中，定義一個程序叫作`square`，該程序定義了一個整數參數`x`，還有一個局部變數`y`，參數`x`的平方值會被計算出來然後回傳給呼叫者。

```
int square (int x)
{
    int y;

    y = x * x;
    return (y);
}
```




3-3 程序定義和使用

程序名稱

- Square

正式參數

- 資料型態為int的變數叫做x

局部變數

- 資料型態為int的區域變數叫作y

程序本體

- 將x 乘上 x 並把值賦予 y

回傳值

- 回傳值為y(x平方)



3-3 程序定義和使用

- ▶ 譬如在下面的程式碼中，先呼叫函數square以計算5的平方，然後將函數回傳的值乘以10之後，再將其值指定給變數 **x**。

```
x = square(5) * 10;
```

這個**x** 跟square內的**x** 一樣嗎？



3-3 程序定義和使用

- ▶ 至於一般沒有回傳值的程序，就如同一般命令的被呼叫，如同下例所示。

```
p->data = 3;  
q->data = 5;  
changehead(p, q);
```



全域變數VS.局部變數

- ➡ 在撰寫一個程式時，我們必須定義變數用來記錄不同的資料。但是根據變數可被使用的範圍，我們可以將變數分為兩類：
 - ▶ **全域變數**(global variable)：能被全部的程式碼使用到。
 - ▶ **局部/區域變數**(local variable)：只能被一部分程式碼使用到，通常定義在程序中。



全域變數VS.局部變數

➡ 以下面這個C程式的範例來說明：

```
int a;  
void proc(int b)  
{  
    a = 3;  
    b = 5;  
}  
main( )  
{  
    int c;  
  
    a = 7;  
    c = 9;  
    proc(11);  
}
```



全域變數VS.局部變數

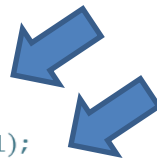
- ➡ 在C程式裡，定義在每個程序裡的變數，稱作**局部變數**(local variable)，只有該程序可以使用該變數。
- ➡ 譬如，變數**c**為程序**main**的局部變數，若是程序**proc**使用了變數**c**，則為不合法的使用。
- ➡ 至於定義在整個程式碼的最前端，就沒有隸屬於哪一個程序，所以任何程序都可以使用它，這樣的變數稱作**全域變數**(global variabe)。



全域變數VS.局部變數

- ▶ 範例中變數a即為全域變數，所以程序main和程序proc都可以使用它。
- ▶ 首先程序main先將它的值定義為7，接著呼叫程序proc，將其值重新定義為3，所以最後變數a的值會是3。

```
int a;  
void proc(int b)  
{  
    a = 3;  
    b = 5;  
}  
main( )  
{  
    int c;  
    a = 7;  
    c = 9;  
    proc(11);  
}
```





以值傳遞 VS. 以位址傳遞

- ➡ 定義程序時，必須定義**正式參數** (formal parameter)，同時宣告該參數的資料型態。
- ➡ 定義完之後，我們在呼叫該程序時，所提供的符合正式參數資料型態的參數，就稱作**真實參數** (actual parameter)。



以值傳遞 VS. 以位址傳遞

- ➡ 該函數定義了一個正式參數 **x**，其型態為整數，如下所列：

```
int square (int x)
{
    int y;

    y = x * x;
    return (y);
}
```



以值傳遞 VS. 以位址傳遞

- 在下列的運算式裡呼叫該函數時，所提供的真實參數為5：

```
z = square(5) * 10;
```

- 參數傳遞時的傳值就是只傳送變數的值給函式，這時就如同將變數的值指定給另一個變數，傳遞者與接受者兩個變數彼此各佔有一個記憶體，互不相干



以值傳遞 VS. 以位址傳遞

- ➡ 在C程式裡的作法，就是「以值傳遞」(passed by value)。
- ➡ 我們會把真實參數的「值」算出來，然後再傳給正式參數。所以，我們也可以提供一個運算式，作為真實參數。



以值傳遞 VS. 以位址傳遞

- 在下例中，我們會先算出5+3的值之後，再將其傳給正式參數x：

```
z = square(5+3) * 10;
```

- 以值傳遞是一個最方便也最常見的方式，但是它仍然有它的限制，就是沒有辦法改變真實參數的值。



以值傳遞 VS. 以位址傳遞

- ➡ 假設我們希望寫一個程序，把兩個整數值對調，我們寫出來的程序可能如下所示：

```
void donothing(int x, int y)
{
    int temp;

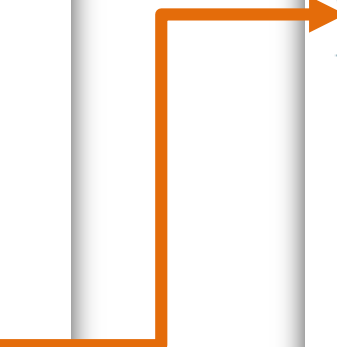
    temp = x;
    x = y;
    y = temp;
}
```



以值傳遞 VS. 以位址傳遞

- ▶ 然後我們在主程式裡，呼叫程序donothing幫我交換變數a和b的值，如下所示：

```
main ( )  
{  
    int a, b;  
  
    a = 3;  
    b = 5;  
    donothing(a, b);  
}
```



```
void donothing(int x, int y)  
{  
    int temp;  
  
    temp = x;  
    x = y;  
    y = temp;  
}
```



以值傳遞 VS. 以位址傳遞

- ➡ 則執行的狀況如下：
- ➡ 在程序裡面參數 x 和 y 的值被調換了
- ➡ 但是對真實參數 a 和 b 卻產生不了任何影響。
- ➡ 所以怎麼辦呢？

```
void donothing(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

```
1. x = 3
2. y = 5
3. temp = 3
4. x = 5
5. y = 3
```

NG!!!



以值傳遞 VS. 以位址傳遞

- 利用「以位址傳遞」(passed by reference)的觀念，也就是把真實參數在記憶體有位址傳給正式參數，讓程序裡的運算直接作用在真實參數上。



以值傳遞 VS. 以位址傳遞

➡ 下面列出 C 語言的寫法：

```
void swap (int *x, int *y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```

➡ 在呼叫的時候，則必須明確地把位址傳過去：

```
main ( )
{
    int a, b;

    a = 3;
    b = 5;
    swap(&a, &b);
}
```



以值傳遞 VS. 以位址傳遞

```
1. x = &a
2. y = &b
3. temp = *x(a) = 3
4. *x(a) = *y(b) = 5
5. *y(b) = temp = 3
```

- ➡ 注意到在第4步裡，雖然在**程序裡表面上是作用在正式參數x**，但因為正式參數**x**和真實參數**a**，其實是指到在記憶體裡的同一塊空間，所以等於是作用在真實參數**a**上面。
- ➡ 程序中我們使用取值運算子*取出這塊記憶體位址的值，並作賦值動作之後再指定回該記憶體位址