CHAPTER 09 程式語言

9-1 程式語言發展史

9-2 資料型態

9-3 程式指令

9-4 程序定義和使用

9-1 程式語言發展史

- FORTRAN
- LISP
- COBOL
- BASIC
- PASCAL
- C
- PROLOG
- ADA
- C++
- Python
- JAVA
- JavaScript 和 ASP.NET
- Kotlin 和 Swift

9-1 程式語言發展史

- 電腦只能接受 **0** 與 **1** 組成的機器語言 **(machine language)**。
- 這些機器語言所代表的意義，通常是做些簡單的加減運算，或是將特定的值指定給暫存器 **(register)**。
- 組合語言 **(assembly language)** 把一個以 **0**、**1** 組成的字串用較容易理解的符號表示，譬如相加之指令以機器語言表示為 **01011010**，而在組合語言則以 **ADD** 來表示。

9-1 程式語言發展史

- 組合語言撰寫出來的程式，須透過組合器 **(assembler)**，轉換成機器語言，才能為中央處理器接受。
- 組合語言缺點:
  - 由於組合語言是直接反應機器語言的指令，必須根據每個中央處理器的特性來設計，所以不同規格的電腦就各自有自己的組合語言，如此造成程式設計師學習上的困難，且寫出來的程式也只能在特定電腦上執行。
  - 組合語言只具備有簡單的指令，所以寫出來的程式通常不具結構性，程式冗長且難以閱讀，也就是我們雖然能夠理解各個指令的意義，但是整個程式所欲達到的功能卻不易理解。

9-1 程式語言發展史

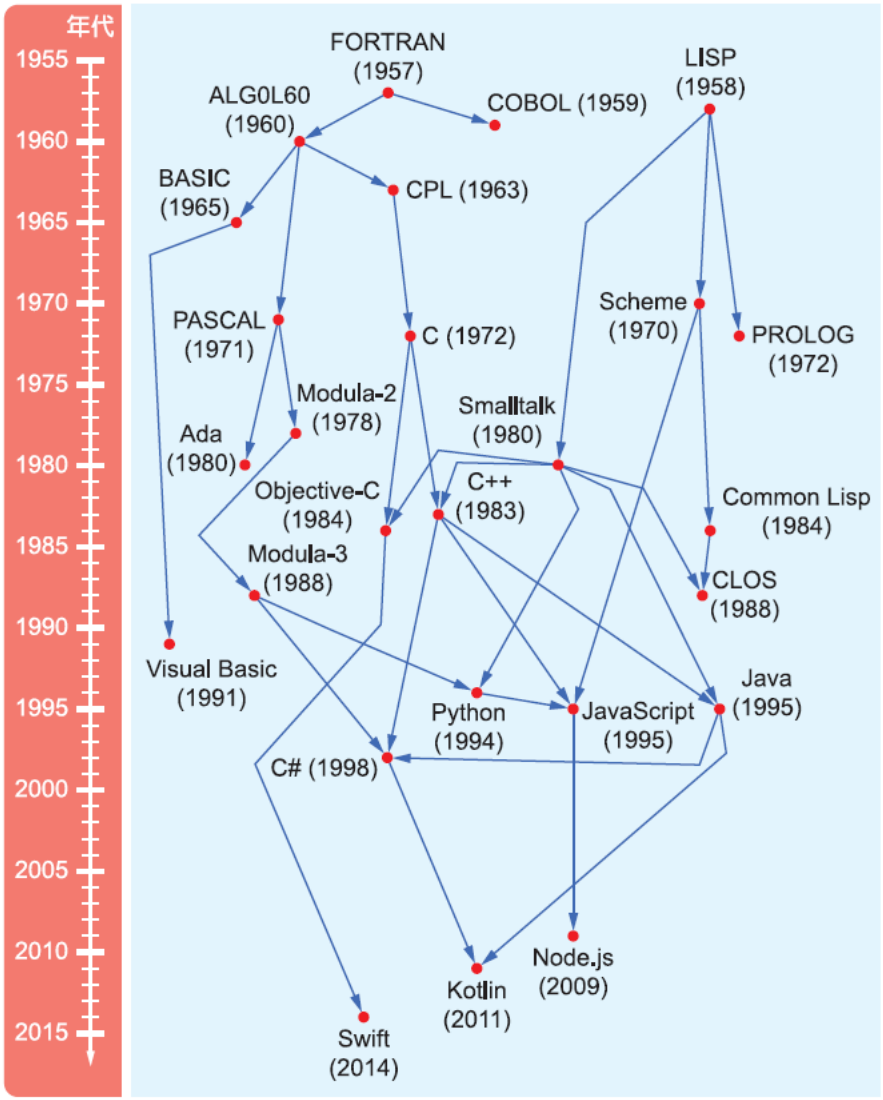
- 組合語言稱作低階語言 **(low level language)**，表示組合語言寫出來的程式可讀性 **(readability)** 很低，同時這也是高階語言 **(high level language)** 被發展設計出來的原因。
- 高階語言如 **C** 語言，寫出來的程式，比起組合語言寫出來的程式，更容易為一般人所理解。
- 高階語言和機器的特性並沒有很密切的對應，所以較具有可攜性 **(portability)**。

9-1 程式語言發展史

- 高階語言寫出來的程式還要經過**編譯(compile)**的步驟才能執行。
- 整個編譯的過程如下圖所示:

高階程式編譯和執行流程

- 第一個誕生的高階程式語言是 1957 年推出的 FORTRAN 。
- 1958 年推出的 LISP，程式的變數並不需要有固定型態，並採用直譯的執行方式，也是有其擁護者。



FORTRAN

- 第一個高階語言是 **IBM** 公司於 **1957** 年右推出的 **FORTRAN(FORmula TRANslation language)**，中文翻譯成「福傳語言」。
- 該語言當初是針對工程方面所需要的複雜科學計算所設計的，因此其程式敘述類似數學的式子。
- 不少工程數學或數值分析的程式及套裝軟體是利用 **FORTRAN** 所書寫的，尤其是需要大量計算的物理、氣象領域。

FORTRAN

```
      DO 7, LOOP = 1, 5
            READ *, X, Y
            AVG = (X + Y) / 2.0
            PRINT *, X, Y, AVG
7           CONTINUE
      END
```

FORTRAN 程式片段:
可以讓使用者輸入 **5** 對數字，然後把該數字和平均值印出來，其中第一行的數字“**7**”對應到第五行的數字“**7**”，用以表示迴圈的範圍。

LISP

- **LISP(LISt Processing)** 是美國學術重鎮麻省理工學院 **(MIT)** 的教授 **John McCarthy** 於 **1958** 年所推出的。
- **LISP** 並不強調數值運算的效率，反而提供很具彈性的符號表示與運算表示式，所以適合做符號運算 **(symbolic computation)**，因此在人工智慧的應用上特別重要。

LISP

- **COMMON LISP 是目前最通用的版本，之後也擴充了 CLOS(Common Lisp Object System)，提供物件導向的程式結構。**

```
(defun length (x)
       (cond ((null x) 0)
                (t    (+ 1 (length (cdr x))))))
(length  '(I love computers))
       3
```

LISP 程式片段:
前 3 行首先定義一個函數叫作“ length”，該函數計算一個串列 (list) 內包含幾個元素。接著在第 4 行呼叫該函數， 並且輸入串列「(I love computers) 」，則會回傳“ 3”。

COBOL

- **COBOL(Common Business Oriented Language)** 是專為商業資料處理而設計的語言，當時是由美國國防部推動成立的資料系統語言組織 **CODASYL(COnference of DAta SYstem Language)** 編定，而於 **1959** 年發表。
- **COBOL** 提供便利的檔案描述與處理，整個程式的結構，也特別重視資料的定義，適於描述不同類型的商業資料。目前仍然有一些早期開發的商業系統，繼續使用 **COBOL**，特別是銀行界。

```
01   EMPLOYEE-RECORD
     05  EMPLOYEE-NUMBER      PIC 9(5)
     05  EMPLOYEE-NAME             PIC X(30)
     05  BIRTH-DATE
         10  BIRTH-MONTH           PIC 99
         10  FILLER                PIC X
         10  BIRTH-DAY             PIC 99
     05  DATE-HIRED
         10  MONTH-HIRED           PIC 99
         10  FILLER                PIC X
         10  DAY-HIRED             PIC 99
```

COBOL 範例:
以階層式的方式定義員工的相關資料。其中“**EMPLOYEE**”稱作集體項，包含階層號碼和資料名稱；其餘的為基本項，除了階層號碼和資料名稱，還包含資料格式定義，譬如“**X**”符號代表文數字資料型態。至於 **FILLER** 主要是用來填補不用或不會參考到的位置，在程式中不會用到。

BASIC

- **1965 年推出 BASIC(Beginner's All purpose Symbolic Instruction Code)。**
- **早期個人電腦還在使用 DOS 作業系統的時候，裡面就附有 QBASIC 的開發環境，所以當時很多人第一個接觸的程式語言就是 BASIC 。**
- **微軟以該語言為基礎，於 1991 年推出 VISUAL BASIC(簡稱 VB)，為 BASIC 語言提供了視覺化的簡易開發環境。**

BASIC

- 目前以 **BASIC** 為主的商用程式語言版本只剩 **VB** 。
- **BASIC** 的好處是簡單易學，缺點則是不夠嚴

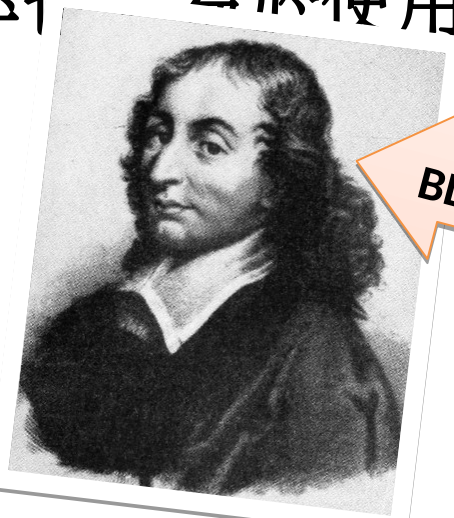
```
Dim i, sum

sum = 0
For i = 1 To 10
        sum = sum + i
Next i
```

BASIC 範例:
計算從 **1** 加到 **10** 的和，其中“ **Dim**” 表示後面要宣告變數，但是並不需要明確指出變數“ **i**” 和“ **sum**” 的資料型態。

PASCAL

- 1971 年推出的 PASCAL，該語言的名稱是紀念 17 世紀重要的法國數學家 BLAISE PASCAL。
- PASCAL 具有完備的資料型態，和結構化的控制結構，所以語言更有效率也更易於使用。
- 由於其程式可讀性高，所以常為教科書教導初學者所用。



PASCAL

- 目前較為人知的是物件化的 **PASCAL** 語言，由 **Borland** 公司的 **Delphi** 產品所支援。

```
function     gcd (m, n : integer) : integer;
       var
              remainder : integer;
       begin
              while n <> 0 do
              begin
                     remainder := m mod n;
                     m := n;
                     n := remainder;
              end;
              gcd := m
       end;
```

PASCAL 範例:
定義了一個 **PASCAL** 的函數叫作“**gcd**”，該函數會根據兩個參數“**m**”和“**n**”，計算他們的最大公因數，然後將該值回傳給呼叫此函數的式子。

C

- 美國 **AT&T** 貝爾實驗室，為了設計 **UNIX** 系統，而於 **1972** 年研發出 **C** 語言。
- **C** 語言和 **PASCAL** 類似，同樣具有高階的結構化敘述，但是為了因應作業系統控制硬體的需求，也具備了類似低階語言的控制硬體能力。
- 由於其強大的功能，在資訊工業興盛的臺灣，幾乎是電資學院以及理工科系學生必修的重要語言。

PROLOG

- **PROLOG(PROgramming LOGic)** 是**邏輯化程式設計 (logic programming)** 的代表。
- **1972** 年於法國所推出，當時的目的是為了自然語言處理的需求所發展出來。
- 常用於設計邏輯推論、專家系統等。
- 和 **LISP** 同樣是人工智慧領域重要的程式設計工具。

PROLOG

- 利用下列的範例，解釋邏輯化程式設計的概念。

```
Facts
        mother (mary, tom).
        father (john, tom).
Rules
        parent (X, Y) :- mother (X, Y).
        parent (X, Y) :- father (X, Y).
Queries
        ?- parent (mary, tom).
                yes
        ?- parent (john, X).
                X = tom
```

PROLOG 範例:
首先，我們先給定兩個事實(facts)，說明“tom”的父親和母親是誰。接著，我們再定義只要是父親(father)或母親(mother)，就是父母(parent)。然後我們可以利用這些事實和法則來詢問系統。第一個問題是想要確認“mary”是不是“tom”的父母，答案是肯定的；第二個問題則詢問“john”是誰的父母，而得到的回覆是“tom”。

ADA

- **ADA** 是由美國國防部於 **1980** 年代主導所設計出來的程式語言，此語言的名稱是紀念世界上第一位程式設計員 **Ada Byron** 。
- 當初此語言的目的是希望結合所有語言的特性，成為一個具有最強大功能的程式語言，但是也由於其語言過於複雜，造成推廣上的困難，目前所知的應用不多。

C++

- 第一個具有代表性的物件導向程式語言 **(object-oriented programming language)** 其實是 **1980** 年左右推出的 Smalltalk，其語言的特性強調物件的設計、訊息 **(message)** 的傳送，比傳統的結構化語言更具模組化的觀念，所以也更易於維護。
- **C++** 則是將物件導向的概念融入 **C** 語言而成，換句話說，**C** 語言可以看作是 **C++** 的子集合。

C++

```
class stack {
        private:
                int top;
                char components[50];

        public:
                stack( )        {top = 0};
                char pop( )     {
                        top = top - 1;
                        return components[top+1];
                }
                void push (char c)   {
                        top = top + 1;
                        components[top] = c;
                }
};
```

我們定義了一個類別（class）叫作“stack”。在類別中，我們除了可以定義資料（data member）外，還可以定義此類別的行為（function member）。以類別“stack”為例，變數“top”和“components”是用以記載此“stack”的相關資料，而函數“stack”、“pop”、“push”則會根據定義好的程式碼執行特定動作。這種把資料和行為一起定義的特性，稱作「封裝」（encapsulation）。

在類別中比較特殊的是，可以指定某個資料或函數的可使用範圍（accessibility）。若是定義為公開的（public），則類別外部的程式碼可使用該資料或函數，如“stack”、“pop”和“push”；若是定義為私自的（private），則只有定義在類別內部的程式碼可使用該資料或函數，如“top”和“components”。如此控管對資料的安全性和完整性更有保障。

Python

- **Python** 的創始者為荷蘭籍的電腦工程師吉多.范羅蘇姆（**Guido van Rossum**）。
- **Python** 本身設計為可擴充的，並不把所有的特性和功能置於語言的核心，而是提供豐富的 **API** 和工具整合其他模組。
- 基於開源的特性，有強大的社群（**community**）群策群力擴充其功能，所以此程式語言簡單但強大。

- Python 簡單易學，具有高度的可讀性與彈性，所以也常常作為初學者入門的程式語言。

```
sentence = ['I', 'love', 'computers']
NoWord = len(sentence)
print(NoWord)
mixed = ['I', 'love', 100]
NoToken = len(mixed)
print(NoToken)
```

Java

- Java 和 C++ 一樣具備有物件導向的特性，但是比 C++ 更容易學習，所以曾經一度是物件導向概念的主要教學語言。
- 但是此語言更前瞻的特性，是提供了跨平台的功能，也就是一個相同的程式，可以在不同的作業環境下執行，所以廣泛的應用於企業級的網路程式開發和行動裝置開發。

JAVA

```
public class stack
{
    private int top;
    private char[] components = new char[50];
    public stack() { top = 0;}
    public char pop( ) {
        top = top - 1;
        return components[top+1];
    }
    public void push(char c) {
        top = top + 1;
        components[top] = c;
    }
}
```

JavaScript 和 ASP.NET

- 於 **1995** 年推出的 **JavaScript**，讓程式碼可以直接編寫在網頁標記中，以便於程式設計師可以於網頁中組裝圖片和外掛程式、或在瀏覽器執行動畫或檢查使用者輸入的函數等等。
- 微軟公司提出一系列以「**.NET**」為名稱的解決方案，其中包含的 **ASP.NET** 開發平台，大幅度地改善了原先 **ASP** 的缺點，將程式分成 **HTML** 和 **Script** 不同的區塊，以便於撰寫和除錯，並具有物件導向語言的特性。

Kotlin 和 Swift

- **2014** 年的 **Swift**，優於 **Objective-C** 語言，更加的快速、現代、安全以及具互動性。
- **2011** 年的 **Kotlin**，主要特色是與 **Java** 的標準函式庫和執行環境相容，所以可於 **Android** 平台上執行，但又沒有如同 **Java** 一般的專利問題。
- 所以 **Swift** 和 **Kotlin** 這兩種程式語言，可以說是目前在手機上開發 **APP** 時，兩大平台各自最受歡迎的主流語言。

▶ 表9-1：程式語言依照特性分類

種類	程式語言	特性
程序式	FORTRAN、COBOL、BASIC、PASCAL、C 、ADA	程式由一連串有順序性的指令組成，相關的指令可定義為程序
物件導向式	C++、Python、Java、JavaScript、ASP.NET、Kotlin、Swift	以具有封裝特性的物件為程式的核心
函數式	LISP	程式視為由運算式組成的函數
邏輯式	PROLOG	提供邏輯判斷的寫法

9-2 資料型態

- 陣列
- 結構
- 指標

9-2 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。

9-2 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)**。

9-2 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

9-2 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 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

- 一般的作法，是將每筆資料用一個節點 **(node)** 表示，然後利用指標將節點串連起來，稱作鏈結串列 **(linked list)**。
- 假設現在要處理的資料是整數型態，則節點的定義如下所示：

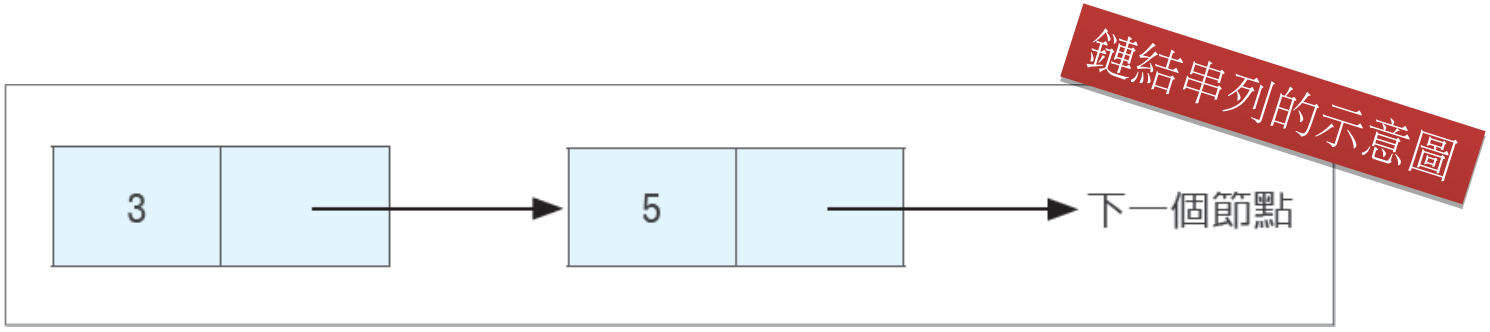
```
struct node
{
        int data;
        struct node *next;
};
```

指標

- 符號「＊」表示後面接的變數字串記錄了位址，也就是說， **next** 代表了記憶體中的一塊空間，而該空間存放的資料型態是 **node** 。
- 第一個節點裡的資料是整數 **3** ，它指到下一個節點，其資料是整數 **5** ，依此類推。

指標

- 如果要再新增資料，只需要建立一個新的節點，然後接到這個鏈結串列即可。
- 若是原先的資料不需要了，也可以將該節點移除，然後把指標重新指定，並不需要做太大的改變。



9-3 程式指令

- 比較： **if**
- 固定次數的迴圈： **for**
- 不固定次數的迴圈： **while** 和 **repeat**
- 不固定次數的迴圈： **for**

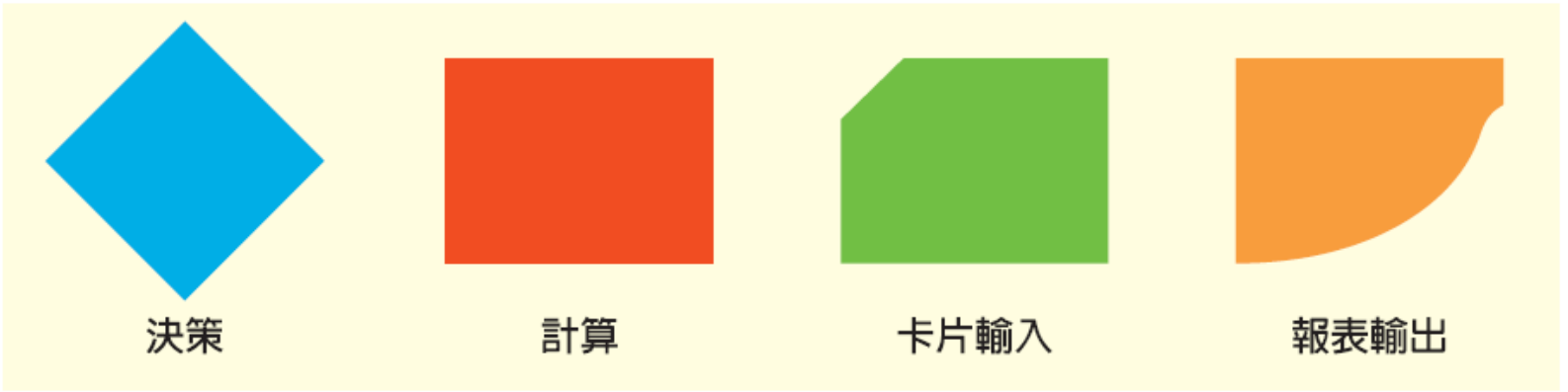
9-3 程式指令

- 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。

9-3 程式指令

- 相關的符號如下圖所示：

流程圖之符號



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，則程式會繼續執行緊跟在後的運算式。
- 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	Python

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

比較： if

- **Python** 和 **C** 的寫法
  - **Python** 在“ **if**” 的條件式和“ **else**” 的關鍵字之後，必須以冒號（ **:** ）結束。
  - **Python** 之後屬於同一個層級的指令，則使用相同的縮排區隔，預設是內縮 **4** 個空格（字元空間）。
  - **Python** 只要可以清楚地分辨出每個敘述（譬如利用換行），就不用在最後加上分號（ **;** ）。
  - **C** ，除了關鍵字之外的敘述，如指定、函數呼叫、變數宣告等，都必須以分號作為結尾。

比較： if

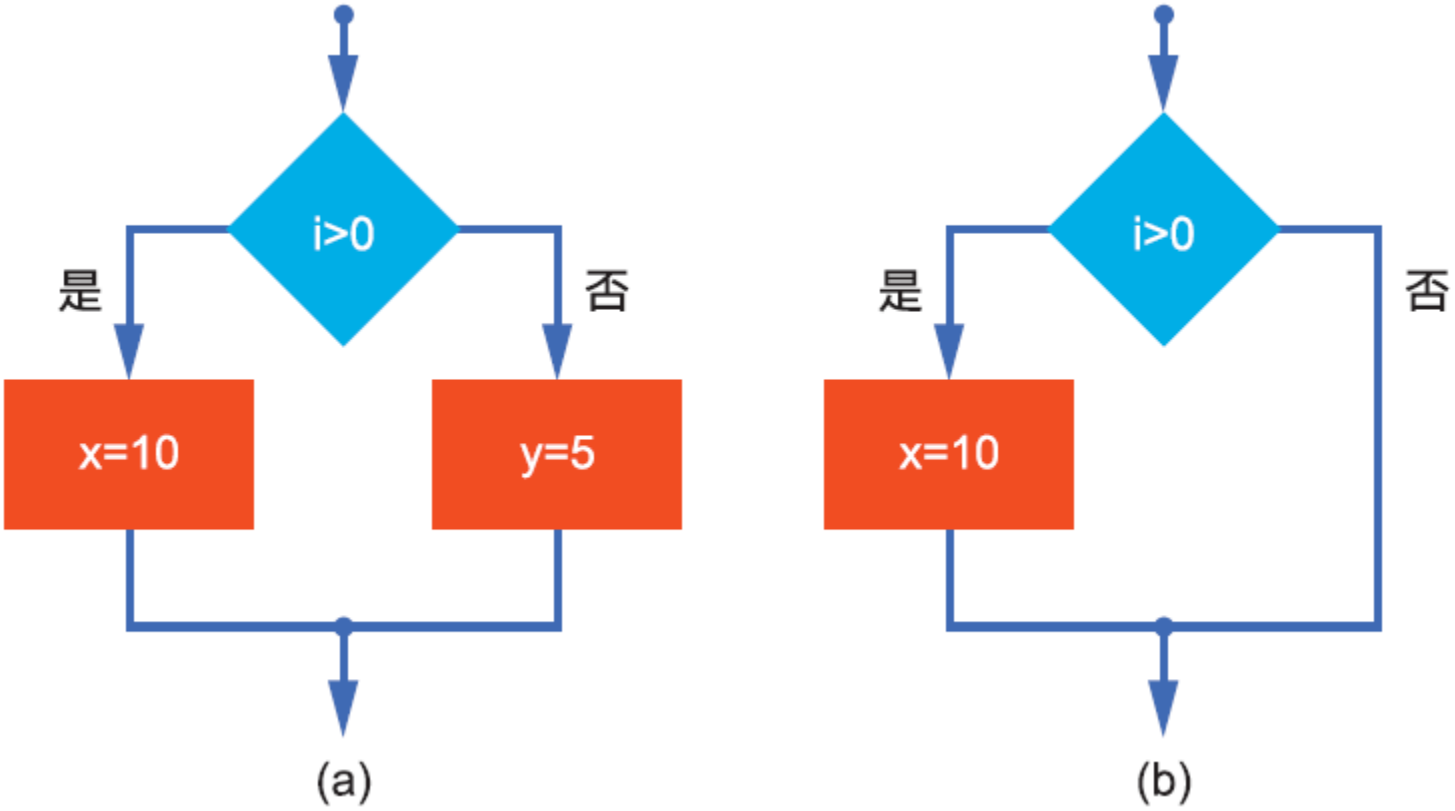
- **下面這個 C 語言的範例， 與上例的差別**，在於變數 i 的值小於或等於 0 時，並不會再進一步執行任何命令，因為我們並沒有提供 else 子句。

C	Python
`if (i > 0)` `    x = 10;`	`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，也就是我們可以在一個 **if** 敘述裡面，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

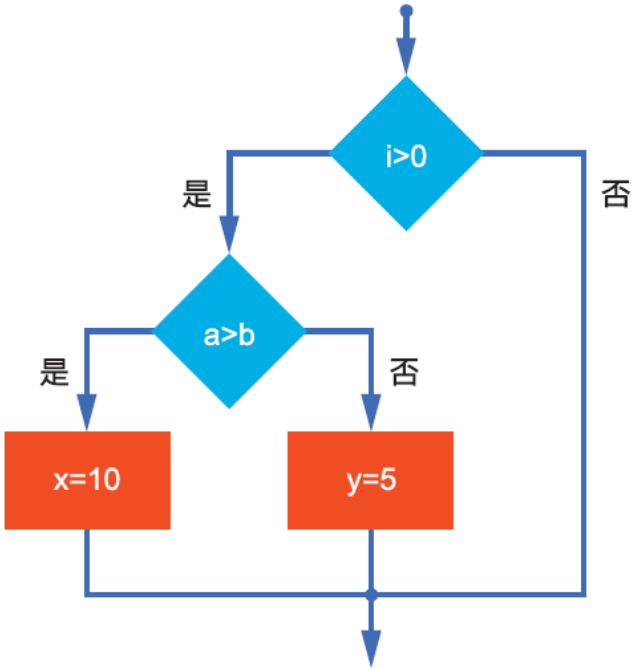
- 值得注意的是，變數 y 的值會被指定為「5」，是在當變數 i 的值為「正」，且變數 a 的值「不大於」變數 b 的值的情況下。

C

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

比較： if

- 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。
- 在下面這個 **Python** 範例中，我們首先透過 “**range(1, 6)**” 這個函數，產生從整數 **1** 到整數 **5** （**6-1**）的連續整數數列。

```
Python
x = 0
for i in range(1, 6):
        x = x + i
```

不固定次數的迴圈

- 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來控制。
- 在 **C** 和 **Python** 裡都提供了相對應的指令，以下我們使用範例加以說明。

- “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，執行定義於內的指令，更改變數“x”和變數“i”的值。

C	Python

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

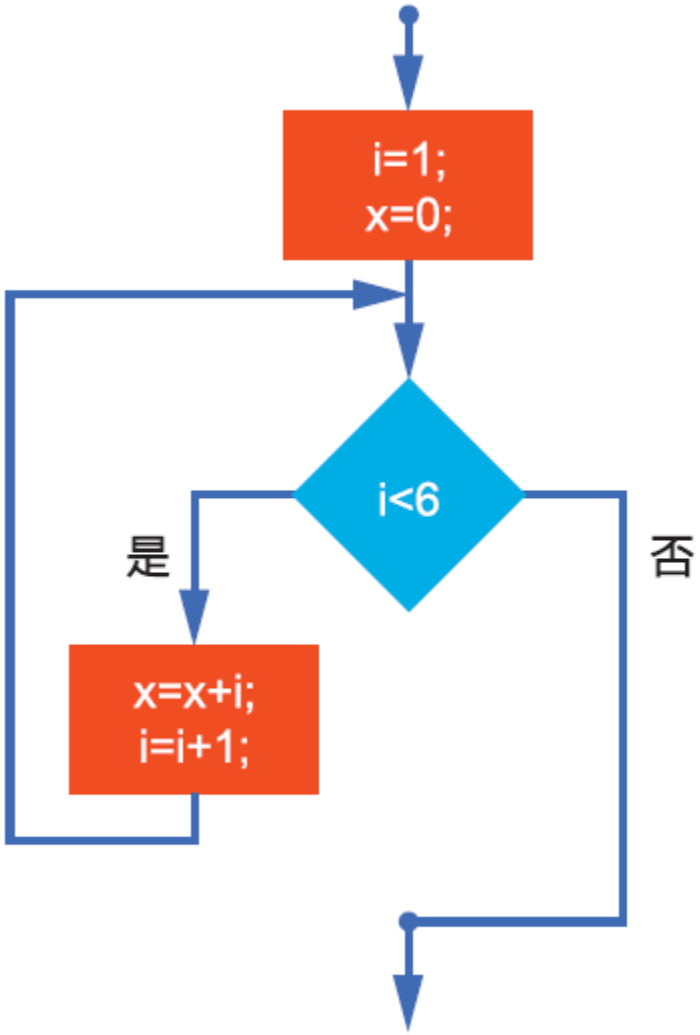


圖9-7 while迴圈的流程圖

- 另一種迴圈的寫法，則是不先做判斷，而是直接執行命令，等到執行完再做邏輯式的判斷。
- 當判斷式為真的時候，程式會回去繼續執行迴圈內的指令。
- Python 並沒有提供此類寫法，而在 C 裡面，則是利用關鍵字 “ do” 和 “ while” 來實作此功能。在下例中，程式同樣會執行迴圈 5 次，並讓變數 “ x” 的值為整數 1 加到整數 5 的和。

C

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

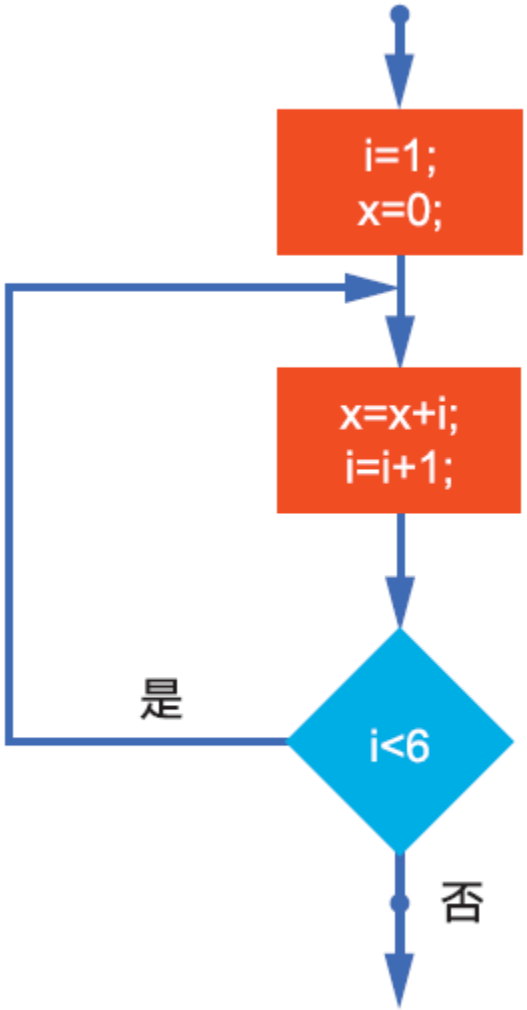


圖9-8 do-while迴圈的流程圖

9-4 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

9-4 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。

9-4 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

9-4 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);
}
```

9-4 程序定義和使用

- 在下例中，將定義一個沒有回傳值的程序。在第 9-2 節中，曾經定義了結構 **node**，用以建構出一個鏈結串列。我們把該結構再一次列在下面：

```
struct node
{
        int data;
        struct node *next;
};
```

9-4 程序定義和使用

- 假設有兩個鏈結串列 **p** 和 **q**，希望將 **p** 串列的第一個 **node**，變成 **q** 串列的第一個 **node**，則對應的程式定義如下：

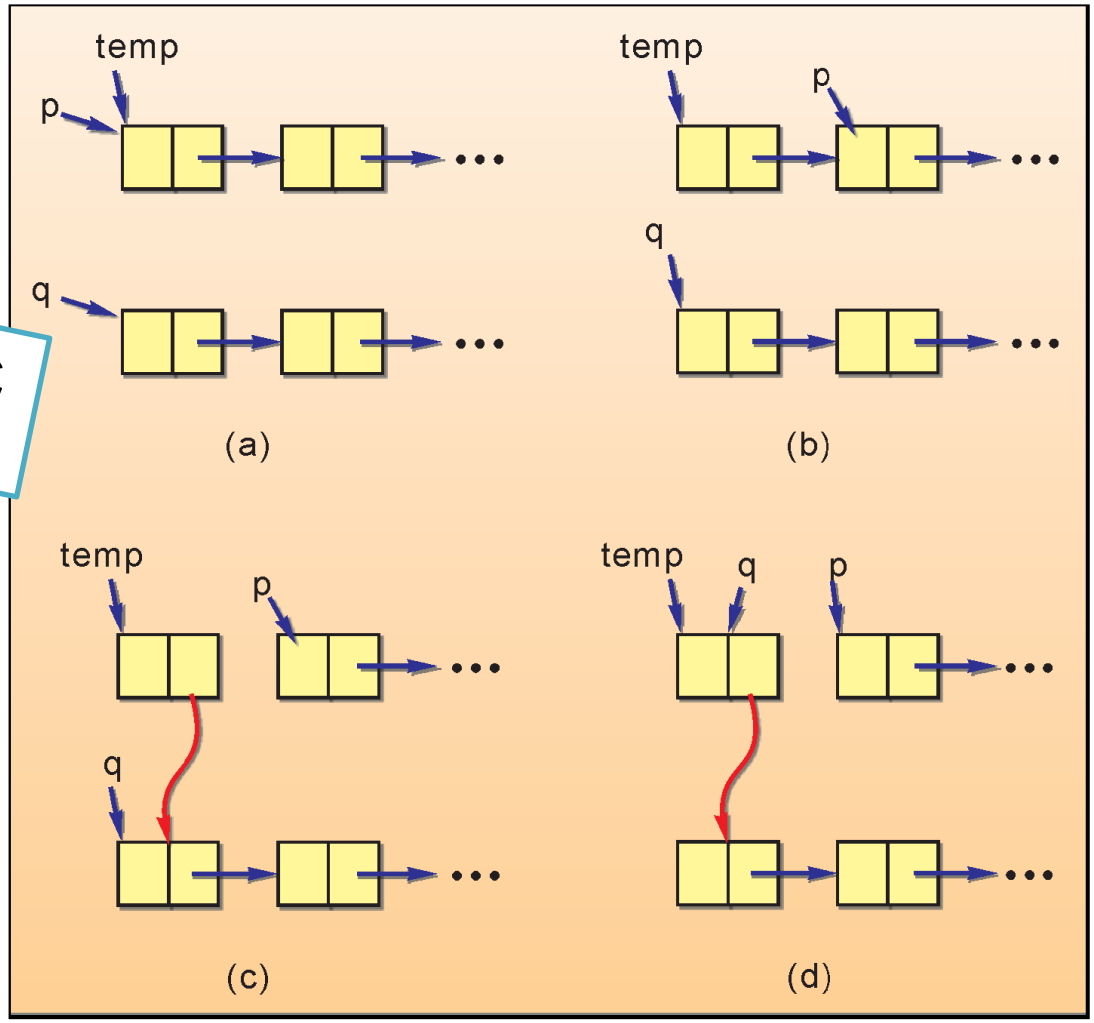
```
void changehead (struct node *p, struct node *q)
{
        struct node *temp;

        temp = p;
        p = p ->next;
        temp->next = q;
        q = temp;
}
```

9-4 程序定義和使用

程序名稱	• changehead
正式參數	• 兩個資料型態為指到結構 node 的指標參數，分別叫作 p 和 q
局部變數	• 一個資料型態為指到結構 node 的指標變數，叫作 temp
程序本體	• 將 p 串列的第一個節點移除，然後加入到 q 串列的第一個節點前
回傳值	• 並無回傳值，在 C 語言裡是以 void 表示之

程序 changehead 的執行步驟示意圖



9-4 程序定義和使用

- 程序 **square** 和程序 **changehead** 的最大差別，在於前者有回傳值，而後者沒有。
- 在 **PASCAL** 裡，有回傳值的程序稱作**函數 (function)**，為了方便起見，我們也一律稱有回傳值的 **C** 程序為函數。
- 值得注意的是，通常我們是在一個運算式裡呼叫一個函數。

9-4 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;
```

9-4 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

- 在本範例中，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**。

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;
```

- 在這裡的問題，就是我們如何把真實參數 **5**，傳給正式參數 **x**，以便進行運算?

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** 幫我們交換變數 **a** 和 **b** 的值，如下所示：

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

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

以值傳遞 vs. 以位址傳遞

- 則執行的狀況如下：

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

以值傳遞 VS. 以位址傳遞

- 在程序裡面，參數 **x** 和 **y** 的值的確被調換了，但是對真實參數 **a** 和 **b** 卻產生不了任何影響。
- 正確的寫法，應該是利用「以位址傳遞」**(passed by reference)** 的觀念，也就是把真實參數在記憶體的位址傳給正式參數，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;
}
```

以值傳遞 VS. 以位址傳遞

- 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

- 則執行的狀況如下：

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

以值傳遞 VS. 以位址傳遞

- 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。
- 第 **5** 步也是同樣的效果。如此一來，就達到了改變真實參數的目的。