創博股份有限公司

IoT 智動化解決方案

# NexMotion NRPL 程式設計手冊

版本: 1.2

日期: 2019-08-29

# 版權與免責聲明

本文件內的所有資料屬創博股份有限公司(以下簡稱本公司)專屬財產,均受智慧財產權相關法規（包括但不限於著作權法 ）所保障。任何未經本公司授權的使用均屬侵權行為。若未經本公司事先書面同意，本文件資料之全部或部份均不可被複印、銷售、散佈、修改、發表、儲存或以其他方式作不當利用。

為力求文件之正確性及完整性，本公司保留在任何時間、不另行通知之情況下，變更或修改本文件之權利。

運行中的機械或設備具有一定的危險性，使用者在做任何操作前，應特別注意並應做好安全防護措施，本公司不承擔因不當使用本文件所述設備所造成的直接或間接損失。

# 文件版本紀錄

| 版本 | 修改紀錄 |
|---|---|
| 1.0 | First released. |
| 1.1 | Add Pointer & Array chapter |
| 1.2 | Add Structure chapter |

# 目錄

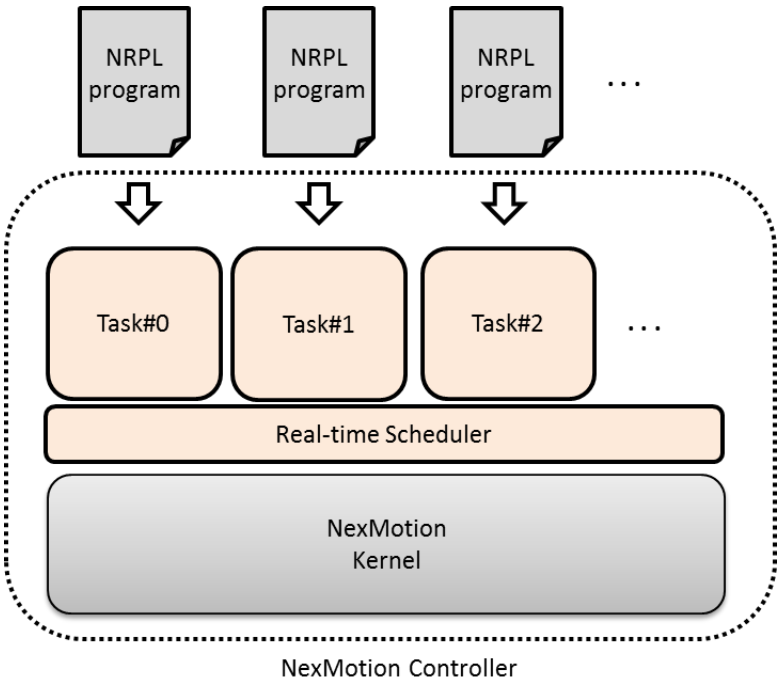
# 1. NRPL 介紹

NRPL(NexMotion Real-time Programming Language) 是一種專為工業自動化及機器人運動控制所開發的高階編程語言，其特點如下:

- 語法近似 C 語言，容易上手
- 支援多任務(Multi-tasks)
- 任務執行具有即時性(real-time)與時間可確定性(time-deterministic)
- 內建運動物件:群組(GROUP)和軸(AXIS)物件及完整運動控制指令
- 運動物件具有豐富完整的功能指令
- 運動指令具有多載(overloading)呼叫設計
- 支援基本變數、陣列、結構和指標功能
- 支援副程式撰寫與呼叫
- 支援數學(Math)函式庫

## 1.1. 系統架構

NRPL 程序為一獨立執行程序，可載入至 NexMotion 控制器的任務(Task)中運行。而 NexMotion 控制器支援多任務(Multi-Tasks)同時執行，所有任務由即時排程器(real-time scheduler)進行排程。基本上，每個任務具有相同的優先次序 (Priority)由排程器視任務中的邏輯運算量分配 CPU 時間，且同一 NRPL 程序每次的執行時間是即時且固定的，因此 NRPL 程序具有時間可確定性(time-deterministic)。




## 1.2. Motion 物件

NRPL 內建運動物件(Motion object)，包括群組(Group) 物件和軸(Axis)物件等，運動物件提供一系列成員函式(member functions)用來控制該物件進行運動控制應用。控制器於啟動時會根據系統實際的群組數量和單軸數量自動建立運動物件。
物件以陣列(array)形式由系統自動產生，在 NRPL 程式中可直接呼叫不必另外宣告，其呼叫方式如下:

```
GROUP[0].PTP( G0P1 ); //Group#0 do PTP motion
GROUP[1].PTP( G1P1 ); //Group#1 do PTP motion
```

詳細的內建指令說明可參考「*NexMotion NRPL 指令手冊*」

## 1.3. 多載函式

NRPL 內建指令具有「多載(overloading)」特性，方便使用者根據應用需求彈性使用。

舉例來說一多載函式 foo()，其原型定義如下:

```
void foo( para1, para2 [, para3] [, para4 ] );
```

此例 foo() 具有四個參數分別為 para1、para2、 para3 和 para4，其中 para1 和 para2 為必要參數，如同一般函數呼叫，必須依順序給定值或變數。而帶有中括號的[, para3] 和 [, para4]表示為非必要參數，呼叫時視應用需求可忽略不寫。

下面幾個例子為 foo() 函式合法的呼叫方式:

```
方式 1:  foo( para1,  para2 );
方式 2:  foo( para1,  para2,  para3=5  );
方式 3:  foo( para1,  para2,  para4=10 );
方式 4:  foo( para1,  para2,  para3=5,  para4=10 );
方式 5:  foo( para1,  para2,  para4=10,  para3=5 );
```

下面是 GROUP 物件一個實際的指令範例，GROUP 有一運動指令 PTP 原型如下:

```
PTP( TargetPos, [VP], [TL], [BS], [Z], [ABORT], [CONT] ], [OW] )
```

其中 TargetPos 目標位置為必要輸入，而速度百分比[VP], Tool 編號[TL]和 Base 編號 [BS]…等參數為非必要參數，下面範例為 PTP() 指令合法的呼叫方式

```
GROUP[0].PTP( P0 ); // PTP 移動至 P0, 其他參數依照系統內定參數
GROUP[0].PTP( P1, VP=50 ); //PTP 移動至 P1, 速度(VP)設定為系統速度之 50%
GROUP[0].PTP( P2, VP=50, Z=30 ); //PTP 移動至 P2, 速度=50%, 平滑化=30 mm
```

詳細 NRPL 指令參數說明可參考「*NexMotion NRPL 指令手冊*」

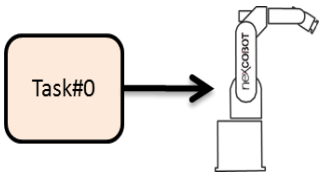
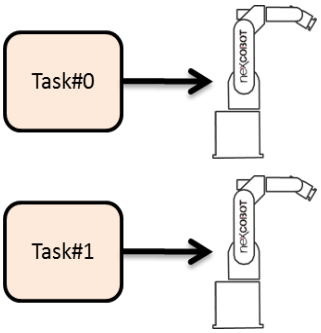
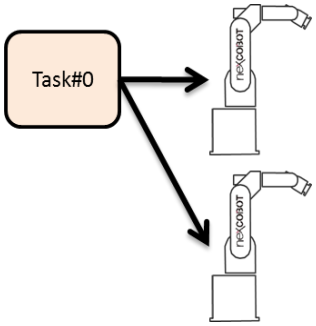
須注意使用者於 NRPL 程序中自訂函數(Functions)並不支援多載呼叫。

## 1.4. 任務與物件

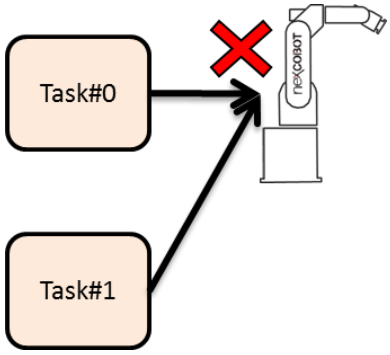
NRPL 任務與物件之關係具有下列規則:

- 一個任務可控制一個或多個運動物件
- 一個運動物件只能被一個任務所控制
- 一個任務可設定一個運動物件為預設物件

下列的應用情境是可被允許的:

| | | |
|---|---|---|
|  |  |  |
| 1 個任務控制 1 個物件 | 2 個任務分別控制 2 物件 | 1 個任務控制 2 個物件 |

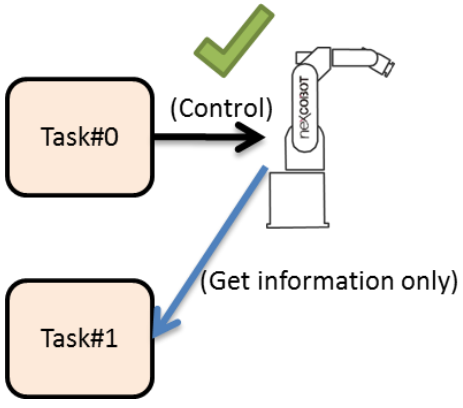
下面這個應用情境是不合法的:



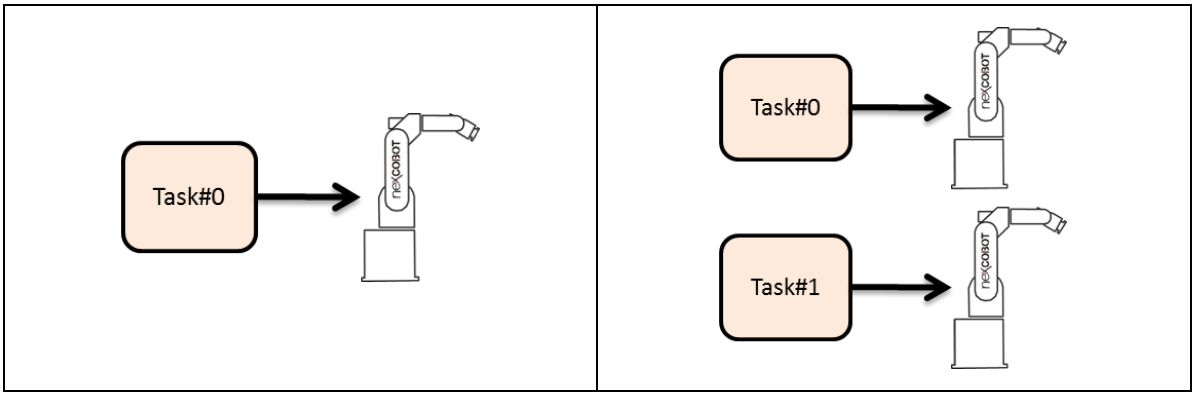

2 個任務對 1 個 GROUP 物件

若兩個任務同時控制同一個運動物件容易造成競爭條件(race condition),容易造成危險誤動作,若有此情況控制器會進入錯誤狀態。

若一個任務已控制一個物件,但有另一個任務試圖讀取同一個物件(例如取得位置),如下示意圖,則此情況是合法的:




一般建議最佳的任務分配為一個任務負責控制一個運動物件,如下圖:




## 1.5. 預設物件

一個 NRPL 程序可設定一個 GROUP 物件為預設物件。當一個 GROUP 設定為 NRPL 程序的預設物件後，可簡化物件指令呼叫，舉例如下:

原程序:

```
void main()
{
  GROUP[0].PTP( P0 );
  GROUP[0].PTP( P1 );
  GROUP[0].PTP( P2 );
  GROUP[0].LIN( P3, TL=1, BS=1 );
  GROUP[0].LIN( P4, TL=1, BS=1 );
}
```

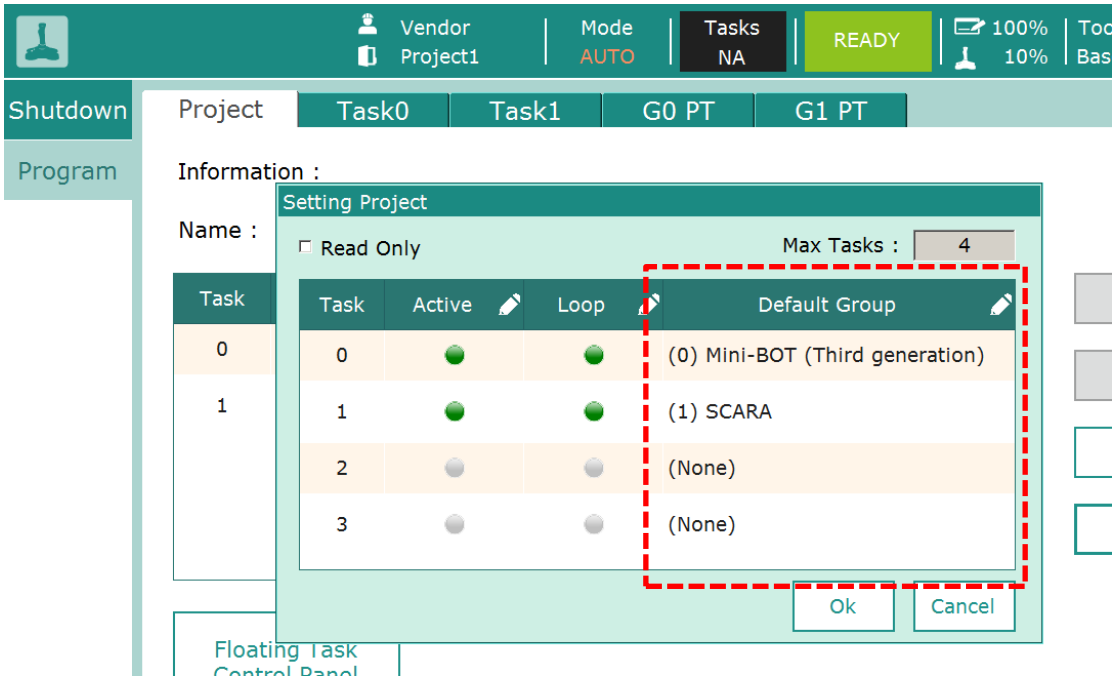
若設定 GROUP[0]為預設物件後，上述 NRPL 程式可簡化如下:

```
// Default using GROUP[0]
void main()
{
  PTP( P0 );
  PTP( P1 );
  PTP( P2 );
  LIN( P3, TL=1, BS=1 );
  LIN( P4, TL=1, BS=1 );
}
```

當控制器編譯 NRPL 程式時，會自動將預設物件之代碼 GROUP[n]加入程式中，因此當任務執行 NRPL 指令時，例如執行 PTP( P0)，能以正確物件執行指令，因此當系統只控制一具機械手臂(GROUP)時(大部分典型工業機器人系統應用)，程式可大幅度簡化。

典型工業機器人系統

設定預設物件的方式可於 NexMotion 控制器使用者介面軟體- NexTPUI 中進行設定或檢視，一般情況下，使用 NexTPUI 建立新專案時會自動將系統中 GROUP 物件設定至任務中，因此使用者無需再特別進行設定。




TPUI Default Group 設定畫面

詳細設定方式可參考「NexTPUI 用戶手冊」。

# 2. 程式結構

NRPL 程式本身不管多大或多小，均是由函數(function)和變數(variable)所組成的。函數中的敘述句(statements)是用來描述應用流程與計算工作。變數則是在記憶體中用來儲存計算的值或流程的控制。下例為 NRPL 程式基本框架:

```
void main() //NRPL Program Entry Point
{
    // Variables deceleration in beginning...

    // main program...
}
```

## 2.1. 進入點

NRPL 程式運行時會有一函式進入點，語法中的進入點的函數格式如下:

```
void main()
{
    [ 敘述句(statements )]
}
```

函數名稱必須為全小寫的 main，且無帶任何參數，而 void 代表為無回傳值。

## 2.2. 敘述句

敘述句(Statements) 事由運算式(expression )所組成，定義如下:
運算式(expression ): 假設兩個變數相加，如 a + b，則這種運算式成為表達式。
敘述句(Statements): 將運算式後面加上分號( ; )，如 a+b;　，則稱為敘述句(statement)

## 2.3. 運算子

在 NRPL 語法中有多種算術符號來表示，諸如乘號運算子為 * ，減號運算子為 - ，這類的運算子可以在第三章節有說明。

## 2.4. 註解

可在 NRPL 程式中標註文字註解增加程式的可讀性，註解之形式可分為兩種:

1. 單行註解 : 利用雙斜線 //， 來表示//後面的文字直到單行結束

2. 區塊註解 : 利用/* 開始， 用*/ 結束

下面是一個範例:

```
void main()
{

    // This is a single line comment

    /*
        Multi-line comments
        Block comments...
    */
}
```

# 3. 變數及運算式

## 3.1. 變數名稱

名稱必須用字母(A~Z，a~z)開頭，然後跟著字母(A~Z)或數字(0~9)。另外底線 _(underline)也算是字母，這在當變數名稱很長的時候，用來增加可讀性(注意:不可含有空白!)。而字母的大寫和小寫在編譯器中是完全不同名稱，所以 NEXCOBOT 和 nexcobot 是兩個不同的名稱。此外，**關鍵字(keyword)**不能當作變數名稱，如 if, else, switch, I32_T… 等均已規定為保留字。

## 3.2. 資料型別

基本資料型別如下:

| 關鍵字 | 大小 (byte) | 數值範圍 |
|---|---|---|
| F64_T | 8 | 1.7E +/- 308，精度 15 位 |
| U32_T | 4 | 0 ~ 4294967295 |
| I32_T | 4 | -2147483648 ~ 2147483647 |
| U16_T | 2 | 0 ~ 65535 |
| I16_T | 2 | -32768 ~ 32767 |
| U8_T | 1 | 0 ~ 255 |
| I8_T | 1 | -128 ~ 127 |

## 3.3. 常數

當輸入整數(e.g. 1234)時，會視作為 I32_T 型態，而輸入浮點數(e.g. 12.34)時，會視作為 F64_T 型態。

## 3.4. 變數宣告

所有變數在使用前均需先宣告，寫法是先指名資料型態，在接著一個或多個該型態的變數名稱，如下:

```
I32_T lower;
I32_T upper;
F64_T value;
```

上述範例為資料型態接著一個變數名稱，接著多個變數名稱寫法如下:

```
I32_T lower, upper, step;
```

變數宣告時可順便給予初始值，做法是宣告之後補上一個等號再跟著初始值或是運算式，如下 :

```
I32_T lower = 0 ;
```

```
I32_T upper = 1 + 2;
```

當執行宣告變數的程式碼時，系統就會配置型態大小的記憶體區塊，以提供之後的程式碼存取使用。

變數的生命週期(區域、全域)是用來規範變數何時可以使用。

區域變數 :

在函數內宣告變數，這變數就屬於區域變數，且只能在函數內使用，當函數執行完後，區域變數的記憶體配置就會釋放。另外，**注意區域變數要在運算式之前，不得在運算式之間穿插宣告變數的行為**。

合法的宣告，如下 :

```
void foo()
{
   I32_T value_1 = 1;    // Deceleration
   I32_T value_2 = 2;    // Deceleration
   I32_T result;          // Deceleration

   result = value_1 + value_2 ;  //Expression
}
```

不合法的宣告，如下 :

```
void foo()
{
    I32_T result;
    result = 0;  //Expression

    I32_T value_1 = 1;  // Illegal Deceleration
    I32_T value_2 = 2;  // Illegal Deceleration
    Result = value_1 + value_2;
}
```

全域變數 :

在函數之外宣告的變數為全域變數，能在每個函數內使用，且只有當程式結束時，才會將記憶體釋放，即結束生命週期。另外，請注意全域變數的宣告，要在宣告函數之前，不能在函數之後。合法的全域變數的宣告，如下:

```
I32_T lower;     //宣告
void foo() {…}   //函數
```

不合法的全域變數宣告，如下 ：

```
I32_T lower;    //宣告
void foo() {…}   //函數
I32_T upper;    //宣告不合法，因為在函數之後
void bar() {…}  //函數
```

### 3.5. 算數運算符號

算數運算子(operator ： 又稱算術符號)包括了五個二元運算子(binary operator) ： +, -, *, /, % 和兩個單元運算子(unary operator) ： +, -。

二元運算子 ：

| Operator | 說明 | 運算元型態 |
|---|---|---|
| + | 加 | 整數, 浮點 |
| - | 減 | 整數, 浮點 |
| * | 乘 | 整數, 浮點 |
| / | 除 | 整數, 浮點 |
| % | 取餘數 | 整數 |

注意整數除法時會將全部小數捨棄，例如 5/2 答案是 2。因為 5 和 2 為整數, 因此答案也會成為整數型態，雖然正確答案為 2.5，但整數型態會捨棄小數點，所以得到的結果為 2。另外，operator % 不可用於浮點數。

單元運算子 ：

| Operator | 說明 | 運算元型態 |
|---|---|---|
| + | 正值 | 整數, 浮點 |
| - | 負值 | 整數, 浮點 |

單元運算子 - 和二元運算子 - 不一樣，例如 8-3 這裡的 - 代表減法，而 -3 這裡表示為有負號的整數。

### 3.6. 關係和邏輯運算子

二元關係運算子 ：

| Operator | 說明 | 運算元型態 |
|---|---|---|

| | | |
|---|---|---|
| > | 大於 | 整數, 浮點 |
| >= | 大於等於 | 整數, 浮點 |
| < | 小於 | 整數, 浮點 |
| <= | 小於等於 | 整數, 浮點 |
| == | 等於 | 整數, 浮點 |
| != | 不等於 | 整數, 浮點 |

單元關係運算子

| Operator | 說明 | 運算元型態 |
|---|---|---|
| ! | 否定 | 整數, 浮點 |

運算式 8 > 3 當成立時，則會回傳 1 ，否則回傳 0。而單元關係運算子 ！ 是將非 0 的運算元轉為 0，將 0 轉為 1。

邏輯關係運算子

| Operator | 說明 | 運算元型態 |
|---|---|---|
| && | 左右運算元皆大於 1，才成立 | 整數, 浮點 |
| \|\| | 左右運算元其中一個大於 1，就可以成立 | 整數, 浮點 |

請注意，關係和邏輯運算子得到的結果之型態皆為 I32_T。

### 3.7. 型別轉換

運算式中，若運算符號兩邊的運算元素型別不同時，運算前會先依一些規則轉換成**同一型別**。一般而言，是將定義域較狹小的轉成較寬大的。很多運算子都會使運算元產生轉換，並且用下述規則來決定運算結果，稱之為一般算術轉型 (usual arithmetic conversions):

Step 1. 如果其中運算元為 F64_T，則另一個會強制轉成 F64_T，否則進入 step2
Step 2. 如果其中運算元為 U32_T，則另一個會強制轉成 U32_T，否則進入 step3
Step 3. 如果其中運算元為 I32_T，則另一個會強制轉成 I32_T，否則進入 step4
Step 4. 如果其中運算元為 U16_T，則另一個會強制轉成 U16_T，否則進入 step5
Step 5. 如果其中運算元為 I16_T，則另一個會強制轉成 I16_T，否則進入 step6
Step 6. 如果其中運算元為 U8_T，則另一個會強制轉成 U8_T，否則進入 step7
Step 7. 如果其中運算元為 I8_T，則另一個會強制轉成 I8_T。

運算式 5 / 2.0，其中 2.0 屬於 F64_T 型別，而 5 為 I32_T，套用上述規則，就會把 5 轉換成 F64_T，其結果為 2.5 的 F64_T 型別。其中請注意如果浮點數轉整

數時，會忽略小數，產生數值不一致的情況。

型別轉換也可用轉型運算子(cast operator)明顯地寫於運算式中強迫轉型，格式如下:

(type-name) expression

假設運算式(F64_T)5 + (F64_T)2,其中 5 為 I32_T 型別,套用強制轉型為(F64_T),因此結果為 5.0，而此運算式的結果為 7.0 的(F64_T)型別。

### 3.8. 遞增及遞減運算子

遞增運算子 ++,用來將運算元加 1,而遞減運算子 --,用來將運算元減 1。運算子 ++ 和 -- 均可寫在變數前(如 n++)或是變數後(如++n)，兩者均會將 n 值加 1,但++n 表示先將 n 值加 1 在取用其值，而 n++則表示先取用其值在做加 1 的工作。例如，設 n 值為 5，則

X = n++ ;

將使 X 值得到 5，但是

X = ++n ;

則使 X 值得到 6，

這兩個運算子只能用於變數，所以像(i+j)++這種寫法是不合規定的。

### 3.9. 位元運算子

提供 6 種位元運算子，**請注意只能用於整數的運算元**。

位元運算子

| Operator | 說明 | 運算元型態 |
|---|---|---|
| & | 逐位元做 AND | 整數 |
| \| | 逐位元做 OR | 整數 |
| ^ | 逐位元做 exclusive or | 整數 |
| << | 往左移位元 | 整數 |
| >> | 往右移位元 | 整數 |
| ~ | 取 1 補數，為單元運算子 | 整數 |

AND 位元運算規則 : 只有兩個位元皆為 1 時，其結果才會為 1

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

設 n 為 U32_T 型別，運算子& 用來做位元遮罩，如下:

```
n = n & 255
```

OR 位元運算規則 : 只要其中一個位元為 1，其結果就會為 1

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

將 n 的 bit8 ~ bit31 清為 0。反之，運算子 | 則用來將位元設為 1，例如:

```
n = n | 255;
```

將 n 的 bit0 ~ bit7 設為 1，其餘的 bit 保持不變。

XOR 位元運算規則 : 兩個位元不同時設為 1，相同時設為 0

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

運算子<< 和 >> 分別將左邊運算元向左或向右移位，右邊運算元則是來決定移位幾個 bits,且右邊運算元的值須為正。例如 n<<2 將 n 向左移位兩個 bit，右邊補 0，這相當於將 x 乘以 4。

運算子 ~ 將其運算元各位元 1 變 0,0 變 1,亦即取 1 補數。例如,設 n 為 U8_T 型別，且值為 0，

```
n = ~n;
```

其結果 n 為 255；

### 3.10. 設定運算子和運算式

像以下這種運算式:

```
n = n + 2;
```

等號左邊的變數 n 馬上在右邊出現，可寫成簡潔的形式:

```
n += 2;
```

此處的+=便是設定運算子(assignment operator)的一種。

| Operator | 說明 | 運算元型態 |
|---|---|---|

| | | |
|---|---|---|
| = | 指派 | 整數, 浮點 |
| += | 加 | 整數, 浮點 |
| -= | 減 | 整數, 浮點 |
| *= | 乘 | 整數, 浮點 |
| /= | 除 | 整數, 浮點 |
| %= | 取餘數 | 整數 |
| &= | AND | 整數 |
| ^= | XOR | 整數 |
| \|= | OR | 整數 |
| <<= | 左位移 | 整數 |
| >>= | 右位移 | 整數 |

如果 expr1 和 expr2 均為運算式，則

```
expr1 op= expr2
```

就完全等於

```
expr1 = (expr1) op (expr2)
```

只是前者 expr1 只計算一次。注意括弧夾住 expr2 表示它是自成運算式，例如:

```
x *= y+1
```

是等於

```
x = x * (y+1)
```

而不是

```
x = x * y + 1
```

### 3.11. 條件運算式

條件運算式(conditional expressions)使用問號與冒號聯合組成的三元運算子，格式如下:

```
expr1 ? expr2 : expr3
```

首先計算 expr1，如其值是 1(代表 true)，便接著計算 expr2，而 expr2 的值就當作該條件運算式的值。否則(expr1 為 0)就計算 expr3 當作條件運算式之值。注意 expr2 和 expr3 只會計算其中一個。範例如下 :

```
z = (a > b) ? a : b ;      /* z=max(a, b) */
```

可將 a 或 b 較大的值放入 z。如果 expr2 和 expr3 的型別不同，則結果就依照一

般算術轉型(2.2.7 節)的規則決定。假如 a 為 F64_T，而 b 為 I32_T，其結果的型別均為 F64_T。

## 3.12. 運算符號優先順序

| 運算符號 | 計算次序 |
|---|---|
| ( ) , [ ], ->, . | left to right |
| !, ~, ++, --, +(正值), -(負值), *(indirection), &(取 address), | left to right |
| *(乘), /(除), %(取餘數) | left to right |
| +(加), -(減) | left to right |
| >>, << | left to right |
| <, <=, >, >= | left to right |
| ==, != | left to right |
| & (bitwise AND) | left to right |
| ^ (bitwise XOR) | left to right |
| \| (bitwise OR) | left to right |
| && | left to right |
| \|\| | left to right |
| ? : | right to left |
| =, +=, -=, /=, %=, &=, ^=, \|=, <<=, >>== | right to left |

# 4. 控制流程

## 4.1. Statement and block

運算式如 x = 0 或 x++等在其後面補上分號（ ; ）便成為 statement。在語法中分號是 Statement 的終止符號。

大括弧 { } 是用來將一群 Statement 夾住而成為 compound statement，或叫做區段 (block)，在語法上可視為一個單一 statement。例如大括弧將整個函數所有 statement 括起來便是個明顯的例子。在區段結束後，也就是又大括弧的後面不用加分號。

## 4.2. if - else

利用 if-else 可以用來表達決策，語法如下

```
If ( expression )
   statement_1
else
   statement_2
```

而 else 部分可有可無。執行時先將 expression 算出，若是非 0(代表 true)就執行 statement_1，否則執行 statement_2。雖然 statement_1 與 statement_2 均為只能一道 statement，但前述 compound statement 可當作但一 statement 使用，所以使用大括弧便可寫一群 statement，如下:

```
If (expression)
{
     statement
     statement
     statement
} else
{
     statement
     statement
}
```

另外 else 部分可有可無，所以巢狀 if statement(即 if statement 包含另一個 if statement)中如果省掉一個 else 部分時,便會使閱讀程式碼造成混淆，

如下範例:

```
If (n > 0)
   If (a > b)
     z=a;
   else
     z=b;
```

此例中有兩個 if 但只有一個 else，所以 else 當然是屬於較接近他的 if，因此我們將 else 與第二個 if 對齊，來表示 else 屬於第二個 if，即使故意讓 else 與第一個 if 對齊也不能改變其意義。如果要讓 else 屬於第一個 if，那就用大括弧，如下:

```
If (n > 0)
{
    If (a > b)
        z=a;
} else
    z=b;
```

所以寫程式時須養成有巢狀 if 時就用大括弧的習慣。

if-else 也可寫成多種決策的寫法，一般的 if-else 寫法如下:

```
if (expression)
  statement_1
else
  statement_2
```

而多種決策寫法(else-if)，就是將 statement_2 換成 if-else 寫法，如下:

```
If(expression)
  Statement_1
else
  if(expression)
    statement_2
  else
    statement_3
```

依此類推，如果還需要就把 statement_3 換成 if-else 寫法，而上述程式碼風格會讓閱讀程式碼產生混淆，因此可整理成如下:

```
if (expression)
  Statement_1
else if (expression)
  statement_2
else
  statement_3
```

各種運算式是依由上而下的順序，只要有一運算式為 true，就執行對應的 statement，同時結束整個結構，如果全部運算式都不成立，則執行 else 部分的 statement。

### 4.3. switch

另一個處裡多重決策的 statement 是 switch，他測試運算式的值是否與許多個常數中的一個相同，依此執行相對應的 statement，寫法如下:

```
switch (expression)
{
     case const-expr : statement
     case const-expr : statement
     default : statement
}
```

其間可以有很多 case，每個 case 之後須有**一個常數且為整數**，此外還有一個叫 default:，但只能一個 default。執行 switch 的運算式之值，然後由上而下依序比較 case 之後的常數值，若相等就從該 case 之後的 statement 執行下去，若全部沒有符合 case，就執行 default 的 statement。請注意，若 default 拼錯了則會被當作普通的可 goto 的位標。

執行 switch 時,若找到符合 case 就會執行下去,且不會跳出 switch 區塊，如下寫法:

```
switch (expression)
{
case 1 :
    statement_1
```

```
case 2 :
    statement_2
case 3 :
    statement_3
default:
    statement_4
}
```

假設 expression 計算出來的值為 2，按照上述的程式碼，則會執行 3 個 statement，分別為 statement_2, statement_3, statement_4。若要執行完 statement_2 就結束 switch 區塊，則要加上 break 關鍵字，來表示結束 switch 區塊，如下:

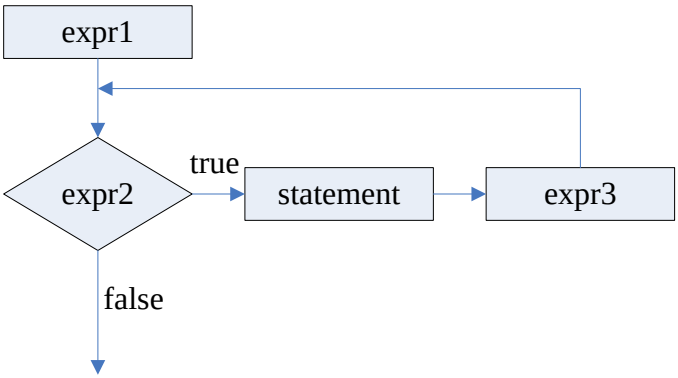
```
switch (expression)
{
case 1 :
    statement_1
    break;

case 2 :
    statement_2
    break;

case 3 :
    statement_3
    break;

default:
    statement_4
}
```

## 4.4. for

for 迴圈格式語法如下:

```
 for (expr1; expr2; expr3)
    statement
```

語法上 for 括弧中的 expr1, expr2, expr3 均為運算式(注意其間用分號隔開)。
運作流程如下:




當 expr2 為 false 時,則結束 for 迴圈。另外,for 中的 expr1, expr2, expr3,皆可省略,但其中 expr2 省略,則會被認為測試結果為 true,形成無窮迴圈,如下:

```
for ( ; ; )
{
    statement
}
```

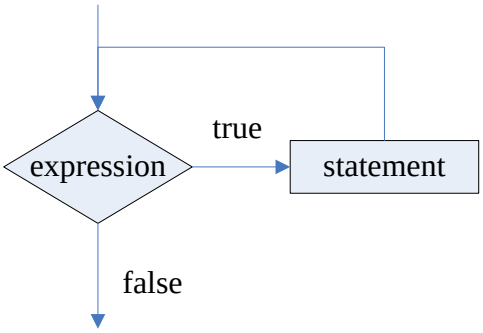
若要跳離迴圈,可使用 break。

## 4.5. while

while 迴圈格式如下:

```
while (expression)
      statement
```

運作流程如下:




只有當 expression 為 false 時，才會結束迴圈。也可以使用 break，來結束 while 迴圈。while 的無窮迴圈寫法如下:

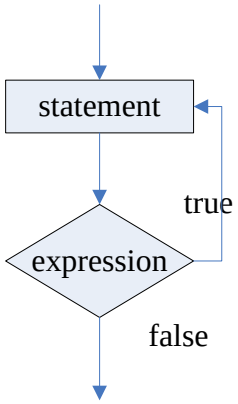
```
while(1)
    statement
```

### 4.6. do - while

do-while 迴圈格式如下:

```
do
    statement
while (expression)
```

do-while 與 while 行為剛好相反，while 是先判斷 expression 才做 statement，而 do-while 是先做 statement 才判斷 expression，其 do-while 運作流程如下:




可以使用 break，來結束 while 迴圈。

### 4.7. break and continue

有時候如果能在迴圈中離開是很方便的，break 便是提供 for, while,

do-while 中提早離開的方法，或是逃出 switch 區塊。但須注意 break 只離開最內層迴圈或是 switch(即 break 所在的那一層)。例如使用 break 在雙層 for 中

```
I32_T m, n;
I32_T result = 0;

for (m=0 ; m < 10 ; m++)
{
    for( n=0; n<10 ; n++)
    {
        if(n==5)
        {
            break;      //逃出第二個 for 迴圈
        }
        result += 1;
    }
}
```

第二個 for 迴圈裡面，執行到 n==5 時，會就跳出，因此雙層迴圈會讓 result+=1 執行 50 次。

與 break 相對應的是 continue，它並不會離開迴路，只是立即停止這一次迴路而跳到測試迴圈的地方，以決定是否進行下一次迴圈。在 for 迴圈，就是跳到 expr3 的地方。continue 只能用於迴圈中，並不適用 switch。範例如下:

```
I32_T m;

for( m=0 ; m <50 ; m++ )
{
    if ( ( m % 2 ) == 0 )
    {
        continues;   //忽略 m 為 2 的倍數
    }
    Statement
}
```

上述範例中，statement 只有當 m 不是 2 的倍數才會執行。

## 4.8. goto and label

goto 和 label 只能在函數內使用，格式如下

```
void main()
{
goto label;
    statement


label :
    statement
}
```

goto 後面要緊接著 label 名稱，label 後面接著冒號，範例如下

```
    goto ERROR;
    statement_1
    statement_2
ERROR:
    statement_3
```

當執行到 goto ERROR 時，會馬上跳到 ERROR 的地方開始執行 statement_3，而不執行 statement_1, statement_2。

理論上 goto 是不必要的，但某些情況適合使用 goto。最常見的就是在相當深的巢狀迴圈中要放棄處理，例如馬上結束兩層或更更多層的迴圈時，此時 break 就相當不好用，因此 break 只能跳離一層迴圈，所以寫法如下:

```
for( expr1 ; expr2; expr3 )
{
    for( expr3; expr4; expr5 )
    {
        if (disaster)
            goto ERROR;
    }
```

```
}

ERROR:
    statement
```

label 可以附於任何 statement 之前，函數中 goto 可以跳到函數內的任何地方。通常使用 goto 的程式不但較不易閱讀也較難維護，但必要時可適當使用。

# 5. 函數

函數(function)是一個獨立的程式單元。利用函數可將大型計算工作分解成若干較小型的工作，也很我們很容易利用別人或自己以前寫好的函數而不必一切重寫。恰當的函數會將運算細節隱藏起來，使用此函數的人不必知道其細節，使整個程式更為清晰，也易於修改。

## 5.1. 使用者自訂函數

函數的格式如下:

```
return-type function-name ( parameter declarations )
{
  statements
}
```

回傳值(return type)可以是 F64_T, I32_T…..等，而另外 void 型態是指說，此函數沒有回傳值。除了 void 之外，其餘的型態，皆要在函數內回傳相對應的型態，回傳格式如下:

```
  return expression;
```

其中 expression 的型態需與 return type 相同否則視為不合法。Parameter declarations 為使用函數傳入的參數，格式如下:

```
   type variable-name
```

傳入參數可以多個，利用逗號 ‘, ‘ 隔開，如下範例:

```
I32_T add (I32_T a, I32_T b)
{
    return a + b;
}
```

另外 parameter declarations 也可以不寫，如下:

```
I32_T get_value()
{
```

```
    return 10;
}
```

程式就是一群變數與函數的集合。函數間的溝通可以透過 parameter, 全域變數, 函數值來完成。下列範例程式為費式序列(Fibonacci sequence)，

```
I32_T fib(I32_T n)
{
  If( n==0 ) return 0;
  if( n==1 ) return 1;
  return (fib(n-1) + fib(n-2));
}

void main()
{
    I32_T  value;
    value = fib(10);    //費式序列 第 10 個值
}
```

上述 fib 函數裡面呼叫自身的函數，這稱之為遞迴(recursion)，在一些演算法中，如快速排序程式(quick sort)，可以使程式碼簡潔，也較容易了解。

## 5.2. NRPL 內建函數

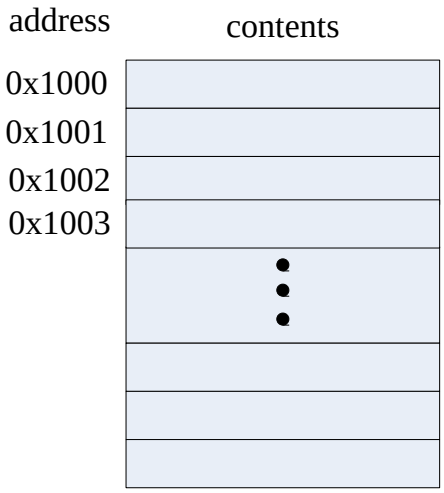
NRPL 已內建運動物件、物件成員變數和相關的資料結構，用來進行設備或機器人之運動控制，使用者無需設定或宣告可直接呼叫使用，詳細的內建物件參數說明可參考「*NexMotion NRPL 指令手冊*」。

# 6. 指標與陣列

指標(pointer)也是一種變數，存著另一個變數的位址(address)，使用指標可使程式更精簡而有效率，但若不當使用指標也可能導致程式崩潰(Crash)。

## 6.1. 指標(pointer)與位址(address)

指標(pointer)簡單的說就是一個專門存放記憶體位置(memory address)的變數。

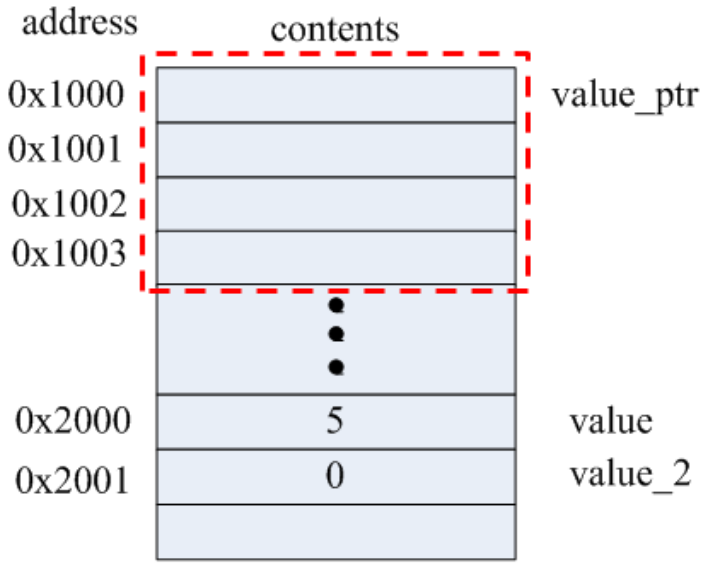



每個記憶體單元(memory cell)都有個連續編號或稱位址(address)，可被單獨存取或將連續的幾個單元看作一組資料。每個單元大小為一位元組(byte)可放一個 U8_T 大小的變數，而四個連續單元(4 bytes)可放 U32_T 或 I32_T。而 NRPL 的指標型態是一個連續 4 個 bytes 的大小型態，主要用來儲存一個 address。

下面例子說明如何宣告指標變數:

```
U8_T * value_ptr;   //宣告指標型態，利用 type(U8_T)之後加上 * 來表示
U8_T value = 5;     //宣告 U8_T 型態，並初始化為 5
U8_T value_2 = 0;   //宣告 U8_T 型態，並初始化為 0
```

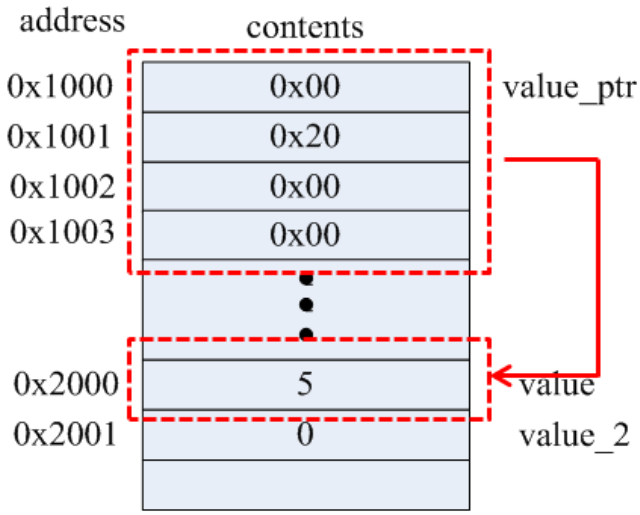
上述的變數宣告，可用下圖說明，當宣告 U8_T *value_ptr，其中 U8_T * 是表示此變數為一個 U8_T 指標型態，且記憶體位是在 0x1000。而宣告 U8_T value = 5，表示系統分配 value 的記憶體位址在 0x2000，且內容指為 5。




利用單元運算子&可取得標的物之位址(通常為變數的位址)，稱作取址運算，如下:

```
value_ptr = &value;  //將 value 的位址(0x2000)存到 value_ptr
```

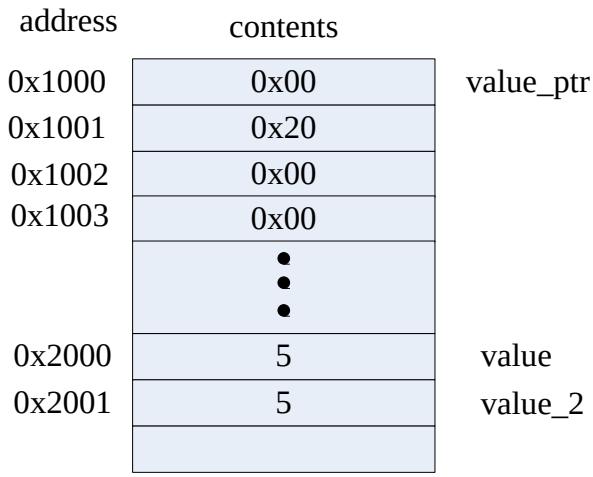
執行完後,如下圖所示,將 value 的位址設定給指標變數 value_ptr,稱 value_ptr 指向 value。這個**單元運算子&只能用於記憶體內的標的物**，即變數或陣列的元素。常數與運算式都無法取得位址。




單元運算子 * 為間接參考(indirection)或稱作解參考(dereferencing)的運算符號。利用間接取址如下:

```
value_2 = *value_ptr;  //到 value_ptr 存的位址，並取 U8_T 的內容值
```

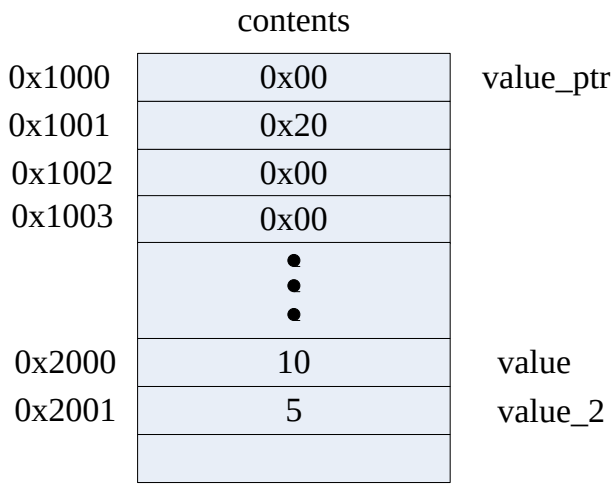
這裡的 *value_ptr，而 value_ptr 的內容為 0x2000，因此可視為 * 0x2000，其運作運作流程是到位址 0x2000 取 U8_T 型態(因 value_ptr 為 U8_T 的指標)的內容值，所以值為 5。執行完上述運算式，如下圖:




指標除了取值之外，也可以指定位址設定內容指，如下 :

```
*value_ptr = 10 ;
```

因*value_ptr 是寫在等號左邊,所以視為到位址 0x2000 設定內容值,執行完後，如下圖:




## 6.2. 指標與函數的引數

函數的引數(argument)是用傳值法(call by value)，則被叫用的函數就沒有辦法直接改變叫用函數傳來的變數。假設要將兩個變數做對調，利用 swap 函

數做這件事，寫法如下:

```
swap(a, b);
```

是無法達到目的，此時的 swap 函數實作如下 :

```
void swap(I32_T x, I32_T y) {
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;
}
```

由於使用 call by value，swap 無法改變 a 和 b 的內容，它只是將 a 和 b 的複製版本對調，離開函數後就等於什麼事也沒做過。

正確的做法是要將 a 和 b 的內容值對調，也就是將記憶體的內容值做對調，因此利用指標的做法，將 a 和 b 的記憶體位址傳入，寫法如下

```
swap( &a, &b);
```

利用單元運算子&取得變數位址，所以&a 就是指向 a 的指標。但光是這樣是不夠的，swap 函數的參數型態也要宣告成相對應的指標型態，並用間接參考方式來存取，寫法如下 :

```
void swap(I32_T *px, I32_T *py)
{
    *x = *x ^ *y;
    *y = *x ^ *y;
    *x = *x ^ *y;
}
```

## 6.3. 陣列

宣告陣列(Array)變數就是規劃一連續記憶體空間中的同一型態變數的集合。陣列提供簡便的方法來表示相類似的資料項，它提供以同樣的變數名稱加上註標

(subscript)表示其中個別的資料項。

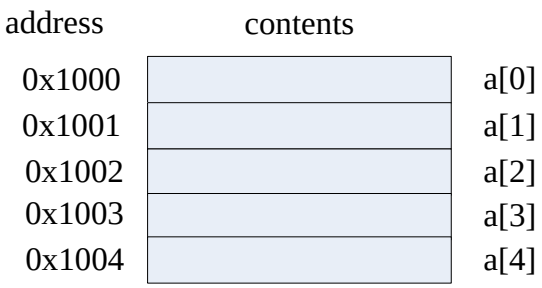
宣告一個陣列，可以使用下面的語法：

```
變數型態  陣列名稱[元素個數];
```

下面是一個例子:

```
U8_T  a[5];  //宣告 a 為一連續 5 個 U8_T 大小的空間
```

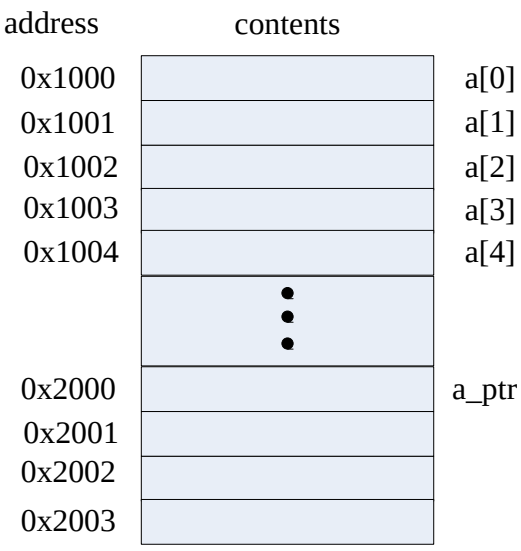
定義a為大小有5個U8_T大小連續記憶體空間的陣列,分別為a[0], a[1]~a[4]，如下圖:




若宣告a_ptr為U8_T的指標，如下:

```
U8_T *a_ptr;
```

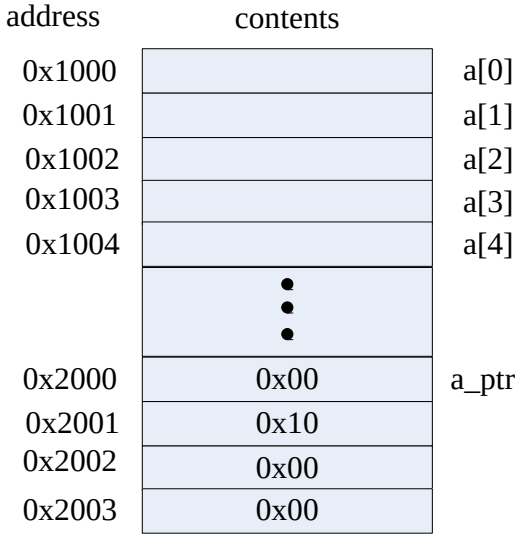
記憶體配置如下:




設定 a_ptr ：

```
a_ptr = &a[0]; //將 a[0]的位址(0x1000)給 a_ptr
```

記憶體配置如下:




此時 a_ptr 就會指向 a[0]的空間，因此 a[0]就可以利用*a_ptr 來存取，如下

```
x = *a_ptr; // *0x1000 同等於 a[0];
```

如果 a_ptr 指向陣列的某個元素，則 a_ptr + 1 就是指向陣列下一個元素，其算法過程如下

a_ptr + (1 * sizeof(U8_T) ) => 0x1000 + ( 1 * 1) => 0x1001

因為 a_ptr 是 U8_T 的指標，而 U8_T 大小為 1 btye。如要取第三個陣列元素，寫法如下:

```
x = *(a_ptr + 2) ; //a_ptr+2 算出來的 address 會是 0x1002，等同 a[2]
```

指標的加或減運算的通則可以如下 :

ptr + expression 或 ptr - expression

這裡的 expression 運算式必須是一個**整數**，指標的加法之位址算法如下:

ptr + ( expression * sizeof(type) )

ptr + expression 為 ptr 往後算整數個元素的位址，也就是說自動將 expression 乘上元素的大小(以 byte 為單位)。另外，**陣列名稱的值同等於就是第 0 個元素的位址**，如果執行 :

a_ptr = & a[0];

同等於

a_ptr = a;

也就是說，寫 a 與 &a[0]是相同的，反之 a[1] 也可以寫成 *(a+1)，因此可以把 a 視為指向 U8_T 的指標，因此只要是指標型態的變數皆可以用陣列註標 (subscript) [ i ] 來存取，如下範例 :

```
a_ptr[0] = 10 ;  // 同等於 a[0]=10
*(a + 1) = 20 ;   // 同等於 a[1]=20;
*(a_ptr + 2) = 30; // 同等於 a[2]=30;
a[3] = a_ptr[0];  // 同等於 a[3] = a[0]
```

如果將陣列名稱 a 傳給函數，事實上傳的是 &a[0]，也是就 a[0]的位址 (0x1000)，下列範例是將利用函數初始化陣列:

```
void init_a_array ( U8_T *ptr )
{
    ptr[0]=0;
    ptr[1]=1;
    ptr[2]=2;
    ptr[3]=3;
    ptr[4]=4;
}
```

## 6.4. 位址計算

如果指標 p 指向某個陣列元素，則 p++會使 p 指向下一個元素，而 p += i 會使 p 指向往下算 i 個元素。這些形式是最簡單的指標計算式或稱位址計算式。

指標只能在某些情況下做比較，如果 p 與 q 都指向同一個陣列，關係運算(==, !=, <, <=, >, >=.. 等)均可使用，例如:

```
p < q
```

任何指標均可做 == 和 != 。但如果指向不同陣列，除了 == 和 != 之外的比較是無意義的。指標也可以加上或減去一個整數(6.3 有說明指標的加減運算)，如下 :

```
P + n
```

代表 p 所指向的元素往後算 n 個元素的位址，也是就是自動乘上每個元素大小。最後合法的指標操作，總結如下 :

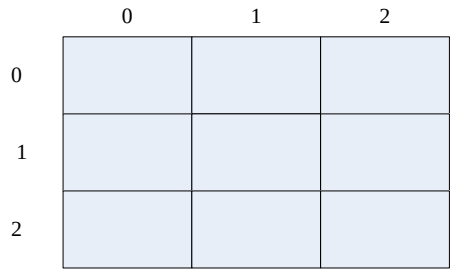
- 指向同型別之指標的相互設定(assign)
- 將指標加上或減去一個整數
- 指向同一陣列各元素的指標可以互相比較
- 指向不同陣列的指標可做相等或不相等的比較

## 6.5. 多維陣列

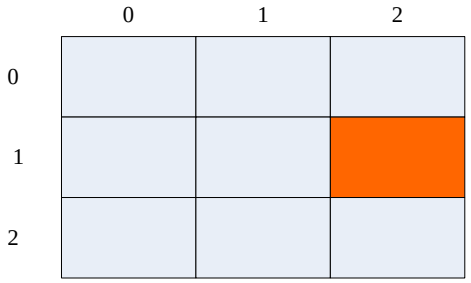
多微陣列可以看做是陣列中的陣列，例如宣告 3x3 的 2 維陣列，如下

```
U8_T var[3][3];
```

用圖表達，如下:




如果存取 var[1][2]代表，存取下圖的位置




當宣告 U8_T var[3][3]時，就會配置連續 3x3 的記憶體空間, 如下:

| Address | Contents | |
|---|---|---|
| 0x100 | | var[0][0] |
| 0x101 | | var[0][1] |
| 0x102 | | var[0][2] |
| 0x103 | | var[1][0] |
| 0x104 | | var[1][1] |
| 0x105 | | var[1][2] |
| 0x106 | | var[2][0] |
| 0x107 | | var[2][1] |
| 0x108 | | var[2][2] |

當取 var[1][2]，其 address 算法為 var+ 1*sizeof(U8_T)*3+ 2*sizeof(U8_T) 等於 0x100+1*1*3+2*1=0x105。另外也可以利用指標使用一維表示法來存取，

```
U8_T *var_ptr = &var[0][0];
```

此時 var_ptr 的內容值為 0x100，如要利用 var_ptr 取 var[1][2]，其表示為 var_ptr[1*3+2] => var_ptr[5]，其位址算法為 var_ptr + 5 * sizeof(U8_T) =

0x100 + 5*1 = 0x105。

## 6.6. 陣列初始化

一維陣列宣告時給予初始化範例，如下:

```
U8_T var[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
```

利用大括弧寫入數值，並用逗號隔開即可完成初始化。另外二維初始化，可以如下:

```
U8_T  var[2][5] =
{
    {  1,   2,   3,   4,   5 } , //var[0][0] ~ var[0][4]
    { 11, 22, 33, 44, 55 }   //var[1] [0] ~ var[1][4]
};
```

利用括弧中的大括弧來分維度，並用逗號隔開，並且會依照記憶體排序填入，例如下列的初始化:

```
U8_T  var[2][5] =
{
    {  1,       } ,  //var[0][0]
    { 11, 22,  }    //var[1][0] , var[1][1]
};
```

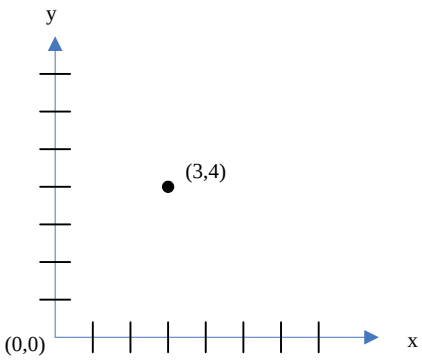
上述初始化的結過會是只有初始化 var[0][0], var[1][0], var[1][1]，其餘的會是未知數。

# 7. 結構

結構(structure)是由一個或多個變數組成，各變數可為不同型態，集合一起透過一個名稱可以方便處理。結構可以用來組織複雜的資料，在大型程式中有顯著益處，因它可使一群有相關聯的資料可被看做一個資料單位來處理。

## 7.1. 結構定義

我們舉一個例子來說明如何定義結構。在圖形處理上我們使用兩個整數 x 座標和 y 座標來描述一個點(point):




我們可將點(point)和其元素 x 和 y 定義如下結構 :

```
struct point
{
    I32_T x;
    I32_T y;
} ;
```

結構的宣告由關鍵字 struct 開頭，其後跟著結構命名，接下去是用一對大括弧夾住宣告。在結構內的變數成為該結構的成員。結構成員名稱可以與一般變數(非結構成員)同名而不相衝突，因為他們可以由上下文來區別。另外，請注意結構成員變數的型態不能是結構本身的型態，如下:

```
struct point
{
    I32_T x;
    I32_T y;
```

```
    point p;  //不合法，因為型態為結構本身
};
```

建立好 struct 型別之後，就進行可以一般變數的宣告，如下

```
point pt;
```

這表示pt成為具有point結構的變數,結構變數在宣告時也可以給定初值，方法是在該變數後跟著等號及一對大括弧夾住其對應的值，如下:

```
point pt = {100, 200};
```

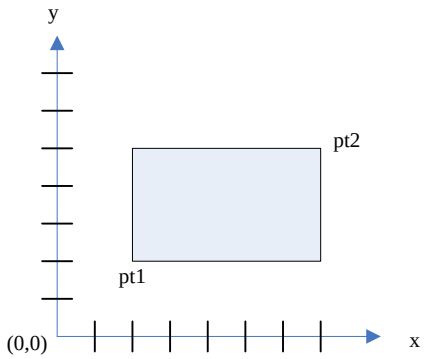
在運算式中要使用結構中的成員時，規則如下:

結構變數名.成員名

利用原點來存取成員，如 pt.x 或 pt.y。

結構可以是巢狀結構，亦即結構可以再包含結構，下面是一個例子:

例如矩形可以由對角的兩點來表示，如下:




定義矩形結構 rect 如下:

```
struct rect
{
    point pt1;
    point pt2;
};
```

結構 rect 包含兩個成員:pt1, pt2 均屬 point 結構，宣告如下

```
rect screen;
```

則以下寫法:

```
screen.pt1.x
```

代表 screen 的點 pt1 之 x 座標

## 7.2. 結構與函數

結構可以使用的運算包括:

- 複製整個結構或設定值給整個結構
- 利用&運算子取得結構位址
- 使用結構成員(取值)

複製與設定包括函數間的引數傳送及由函數傳回結構。

下面是一個函數回傳結構的範例:

```
point make_point( I32_T x, I32_T y )
{
   point temp;
   temp.x = x;
   temp.y = y;
   return temp;
}
```

make_point() 函數傳入兩個整數，並回傳一個 point 結構，利用 make_point() 函數來動態給予任一點結構初值，範例如下:

```
rect screen;
point middle;
```

```
screen.pt1 = make_point(0, 0);
screen.pt2 = make_point(100, 100);
middle = make_point ( (screen.pt1.x + screen.pt2.x)/2,
                      (screen.pt1.y + screen.pt2.y)/2 );
```

也可以將結構當作函數的輸入參數，下面是一個例子:

```
point add_point(point p1, point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

此例中，參數及函數回傳值均為結構，由於函數的引數以「傳值呼叫」(call by value)的方式傳入，所以不會改變呼叫者所傳入的變數。

如果要將大型的結構傳給函數，可利用指向該結構的指標(pointer)進行傳送而不要複製整個結構，來提升程式執行效率。下面是一個宣告結構指標的範例:

```
point  pt;
point *pp = &pt;

(*pp).x = 10;
(*pp).y = 20;
```

宣告 pp 為一指標，指向一個 point 結構 pt。而(*pp).x 和(*pp).y 即為其成員取值。在(*pp).x 中的括號是必要的，因為點(.)運算子的優先權高於取值的星號(*)運算子。

結構指標成員取值也可以使用(->)符號，例如 (*pp).x 可以用 pp -> x 表示，注意->符號是減號緊接著大於的符號。

```
pp->x = 10;
pp->y = 20;
```

下面範例說明函數的輸入參數使用結構指標: 利用指標結構來計算中間點範例

```
point middle_point( point *p1, point *p2 )
{
    point temp;
    temp.x= (p1->x + p2->x)/2;
    temp.y=(p1->y + p2->y)/2;
    return temp;
}
```

## 8. 關鍵字

下列的識別字都保留作為關鍵字之用，不可用來作其他用途。

| | | | | | |
|---|---|---|---|---|---|
| F64_T | U32_T | I32_T | U16_T | I16_T | U8_T |
| I8_T | break | case | continue | default | do |
| else | for | goto | if | return | struct |
| switch | void | while | | | |