C 程式語言-位元運算的介紹 (richwang)

(*) 位元的介紹

位元(binary digit: bit) 表示兩個值：0 與 1。

以一個 1 byte = 8 bits 爲例，一個位元可以表示兩個值 (0,1) (或者更廣義地說兩種狀態)，因此整個 byte (位元組)的表示的範圍就是 2^8=256。

通常電腦在表示有正負號數值時會將最高位元(msb: most significant bit 或最左位元)當成符號位元，而符號位元：0 表正，1 爲負。

正數：msb=0，所以剩餘的 8-1=7 個位元用來表示此數的大小，數值範圍 0 ~ 2^7-1，

即 **0** 0000000 ~ **0** 1111111。

負數：msb=1，但爲了避免 0 在負數的範圍中再重複出現，因此負數的大小是透過取 2 補數(將所有位元做 0 變 1、1 變 0 轉換後，再加上 1)的方式來表示，因此負數的數值範圍將介於：–(2^7 ~ 1) = -2^7 ~ -1 = -128 ~ -1。

解讀負數的規則如下：

1) 首先符號位元爲 1 就說明這是一個負數；
2) 接著來求其大小(相當於計算其絕對值)，將此負數全體取 2 補數後，此時補數在不分正負的值就是此負數的絕對值大小。

例如：(少了-0，因而多了-128)

負數	2 補數	負數的大小	負數值
1 0000000	=> 0 1111111 + 1	=> 1 0000000	=> (128) => -128
1 0000001	=> 0 1111110 + 1	=> 0 1111111	=> (127) => -127
……			
1 1111110	=> 0 0000001 + 1	=> 0 0000010	=> (2) => -2
1 1111111	=> 0 0000000 + 1	=> 0 0000001	=> (1) => -1

(*) 負數的 2 進位數表示

歸納上述的討論，我們可以得到寫出任意負數二進位形式的表示規則。

例如：-38 首先以二進位寫出其大小 38 的表示式：0010-0110 (0x26)，

接著取此數的 2 補數即爲所求：11011001 + 1 = 11011010。

(*) 具有 n 個位元資料可以表示的數值範圍

根據上述的討論，我們可以將結果推廣到有 n 個位元的有號數(signed 有正負區分的數)，其所能表示的數值範圍爲：正數 => 0 ~ 2^{n-1}-1； 負數 => -2^{n-1} ~ -1。

若純粹只是拿來當作無號數(unsigned)也就是只管數值的大小，通通當成正數這時可以表示的數值範圍爲：0 ~ 2^n-1。

(*) 以 typedef 自定新資料型態的說明

1) unsigned char *byte*; // *byte* 是一個變數，這是原來變數宣告的語法。

2) **typedef** unsigned char byte; // 在最前方加上 typedef 就成了型態宣告的語法。

現在我們多了一個自定的(unsigned char)資料型態：byte。

因此可以這樣用：byte *value*; 等同於 unsigned char *value*;

C 程式語言-位元運算的介紹 (richwang)

(*) 位元的運算

C 語言中與位元相關的運算子有：**&, |, ~, ^, >>, <<**。

&	|	~	^	>>	<<
and	or	not	xor	右移	左移

&	0	1
0	0	0
1	0	1

A & 0 = 0
A & 1 = A
任一數 A 與 0 的 & 運算等於 0。
任一數 A 與 1 的 & 運算等於 A。
利用與 0 的 & 運算可以將某位元設成 0。

|	0	1
0	0	1
1	1	1

A | 0 = A
A | 1 = 1
任一數 A 與 0 的 | 運算等於 A。
任一數 A 與 1 的 | 運算等於 1。
利用與 1 的 | 運算可以將某位元設成 1。

^	0	1
0	0	1
1	1	0

A ^ 0 = A
A ^ 1 = not A = A'
任一數 A 與 0 的 ^ 運算等於 A。
任一數 A 與 1 的 ^ 運算等於 not A = A'。
利用與 1 的 ^ 運算可以將某位元反相，0⇔1。

	0	1
~	1	0

~A = not A = A'
同時將所有的位元反相，0⇔1。

(*) 位元的稱呼

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
bit 7 *bit* 6 *bit* 5 *bit* 4 *bit* 3 *bit* 2 *bit* 1 *bit* 0 (*以次方數表示位元位置)

其中的 bit 7 稱爲 MSB；bit 0 則是 LSB (least significant bit)。

(*) 在做位元運算時，請記得將變數資料型態加上 **unsigned**。

(*) 熟記下列表示法可以很快地建構出數值的二進位形式：

2^3 2^2 2^1 2^0
8 4 2 1 => 16 進位的顯示數字：**0123456789ABCDEF**。

例如：$(1011)_2$= 8+2+1 = 11 = 0xB (*轉成 16 進位時，可以記住 13=>D, 3D)

C 程式語言-位元運算的介紹 (richwang)

位元的運算的示範例子	
1101 0011 : original a = 0xD3 1100 0011 : reset bit4 to 0 1101 1011 : set bit3 to 1 0010 0011 : inverse high nibble 0010 1100 : inverse a 1101 0011 : original a = 0xD3	1100 1101 & 1101 0011 and operation ----------- 1100 0001 1100 1101 | 1101 0011 or operation ----------- 1101 1111 1100 1101 ^ 1101 0011 xor operation ----------- 0001 1110 ~ 1100 1101 not operation ------------ 0011 0010

(*) 位元的設定 (設定爲 1 or 0)

先把數值以 2 進位的形式寫出來，再根據題意去設定相關位元的值後，接著祇要將結果以 16 進位來表示即可完成。

例 1：請完成將某數的第 5, 4, 3 位元設成 0 的運算。(使用**&**做位元運算)

unsigned char value = 0xAB;
unsigned char mask;
7654-3210 (善用性質 A&0 = 0, A&1 = A)
1100-0111 => mask = 0xC7, 則 value = value & mask;

例 2：請完成將某數的第 7, 4, 1 位元設成 1 的運算。(使用**|**做位元運算)

unsigned char value = 0xAB;
unsigned char mask;
7654-3210 (善用性質 A|0 = A, A|1 = 1)
1001-0010 => mask = 0x92, 則 value = value | mask;

例 3：請完成將某數的高四位元反相的運算。(使用^做位元運算)

unsigned char value = 0xAB;
unsigned char mask;
7654-3210 (善用性質 A^0 = A, A^1 = 1)
1111-0000 => mask = 0xF0, 則 value = value ^ mask;

C 程式語言-位元運算的介紹 (richwang)

例 4：請保留某數反相後的低四位元，其餘位元設爲 0。(使用**~**與**&**做位元運算)

unsigned char value = 0xAB;

unsigned char mask;

7654-3210　　(善用性質 A^0 = A,　A^1 = 1)

0000-1111 =>　 mask = 0x0F,

則 value = ~ value;　 value = value & mask;

(*) 位元的位移

位元的位移運算子：<< (左移) 或 >> (右移)。

移動後產生的位元空位將會被補上 0。

```
/* a function for showing binary value */
void binary(unsigned char A)
{
   unsigned char index=0x80; // (1000 0000)b
   while (index)
   {  if ( index & A )
         printf("1");
      else
         printf("0");

      index >>= 1;  // 所有位元右移一位，最左邊空缺位元補零。
   }
}
```

位元的右移示範 (to right)	位元的左移示範 (to left)
7654 3210　 => 0x80 >> k	7654 3210　 => 0x01 << k
1000 0000 = 0x80　// (1000 0000)	0000 0001 = 0x01 // (0000 0001)
0100 0000 // >> 1	0000 0010 // << 1
0010 0000 // >> 2	0000 0100 // << 2
0001 0000 // >> 3	0000 1000 // << 3
0000 1000 // >> 4	0001 0000 // << 4
0000 0100 // >> 5	0010 0000 // << 5
0000 0010 // >> 6	0100 0000 // << 6
0000 0001 // >> 7	1000 0000 // << 7
0000 0000 // >> 8	0000 0000 // << 8

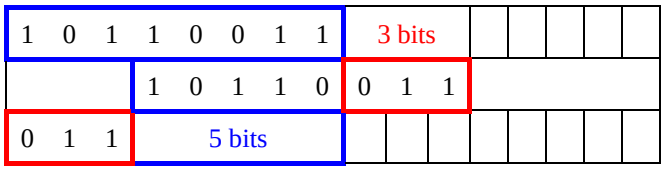
C 程式語言-位元運算的介紹 (richwang)

(*) 循環位移 (Cyclic shift)

假設 value 資料長度爲 n 位元，(可藉由 sizeof(.)來求其所佔的位元數)

1. 將 value 循環右移 k 位元: **`value = (value>>k) | (value<<(n-k));`**
2. 將 value 循環左移 k 位元: **`value = (value<<k) | (value>>(n-k));`**

例題：將 value 循環右移 3 個位元，**`value=(value>>3)|(value<<5);`**



值得特別說明的是，相加與或("|")運算的差別在：

相加運算有可能因爲進位出現數值正負變號的情形，導致產生非預期的結果。

另外，value 所佔的 byte 數可透過 sizeof(value) 求出，再將此求出值乘上 8 就能夠算出 value 總共佔的 bits 數。

例題:請寫出依序將 unsigned char A, R, G, B;合成 int color = (ARGB) 的程式碼。

參考寫法： **`color = (A<<24) | (R<<16) | (G<<8) | B;`**

A	R	G	B

例題：計算某數以二進位表示時共有多少個位元爲 1？

(sizeof(.)的使用：由所得的位元組數(bytes)來推算位元數(bits))

```
int bitNumber(int value)
{
   int k, bit1 = 0;
   int bits = sizeof(value) * 8;  // 1 byte = 8 bits

   for (k=0; k<bits; ++k)
   {
      if ( value & 1 )
        ++bit1;

      value >>= 1;  // 右移一個位元
   }
   return bit1;
}
```

C 程式語言-位元運算的介紹 (richwang)

(*) 簡易的資料加密與解密

利用 xor 的運算特性： $A \wedge B = A \oplus B = A'B + AB'$　其中 $A' = not\ A = \sim A$ ，

兩數相同者其結果爲 0，否則爲 1。

(a) $A \wedge 1 = A'$　$A \wedge 0 = A$

(b) $A \wedge A = 0$

(c) $B \wedge A \wedge A = B \wedge 0 = B$ ，連續 xor 相同的數兩次，具有讓原數值還原的效果。

```
void showSomethingHidden(unsigned char code)
{
    char msg[] = "東南科大資通系";
    int k, len = strlen(msg);

    printf("加密前：\"%s\"\n", msg);

    for (k=0; k<len; ++k)
    {
       msg[k] ^= code;
    }

    printf("加密後：\"%s\"\n", msg);
}
```

(*) 將中文的位元組反相

資料要能恢復必須確認在轉換的過程中沒有資料漏失的現象。

例如：要幫中文做簡單的資料加密時，如果採用爲每個 byte 增與減一數值來當成加密及解密的規則，那麼有可能在加上數值時造成溢位，此時由於加法進位而漏失的資訊便消失了，於是接下來的解密自然就會失敗。

而像 xor，not 與 cyclic shift 的方式，所有的位元資訊均被保留著，只是形式或位置變了，所以將資訊完全恢復是沒有問題的。

在 C 語言中做位元運算時，將資料宣告爲 **unsigned** 可以避免因數值正負關係所導致的解讀錯誤。

例如：$(1000\ 0000)_2$ 到底是被當成 128 還是 -128？

C 程式語言-位元運算的介紹 (richwang)

(*) 究竟是整數還是字串？

在 C 語言中，所有的東西都可以看成是數值！一旦取得該資料的位址時，透過指標的型態轉換就會讓它呈現出多種的形式，其間的變換端看你想如何去詮釋它了！所以，資料可能是整數，但不是你想要的形式；它也可以是字串，但看不出其真正的內容。

```
void intOrString()
{
   char str[] = "資通"; //"中文";
   char *ptr, *toPtr;
   int k, value, len;

   ptr   = (char*) str;  // 對字串而言，變數的值也是其所在的位址！
   toPtr = (char*) &value;
   len = sizeof(int);
   printf("  str= %s or %d or %u\n", str, *str, *str);
   printf("value= %s or %d or %u\n", (char*)&value, value, len);

   for (k=0; k<len; ++k) // 將字串的內容逐一填入 int 中
   {
      toPtr[k] = ptr[k];  // toPtr[k] = *(toPtr+k);    ptr[k] = *(ptr+k);
   }

   printf("  str= %s or %d or %u\n", str, *str, *str);
   printf("value= %s or %d or %u\n", (char*)&value, value, value);
}
```

(*) 練習題

1) 利用二進位的方式寫出你學號末兩碼的負值，並寫程式印出驗證。
2) 計算一數中有位元是 1 的個數。
3) 將指定的某些位元設定成 0，1 或反相。
4) 完成 2 進位的換算與顯示程式，binary(n)。
5) 交換一個 byte 的前後四個位元(nibble)。
   (* unsigned char value = (value<<4) | (value>>4);)
6) 請模擬一個會循環的位元位移效果，即由右邊移出的位元會轉而成爲最左邊的位元，反之亦然。因此在經過 8 次的向左或往右循環位移後，將會回到原來的結果。
7) 給一個無正負號整數 unsigned int value = 0x1A2B3C4D; 請取出從最左方數來的第 2 個位元組(byte)的資料，並將所得結果以 2 進位的形式印出。
8) 簡易的資料加密與解密。