

你所不知道的 C 語言：數值系統篇

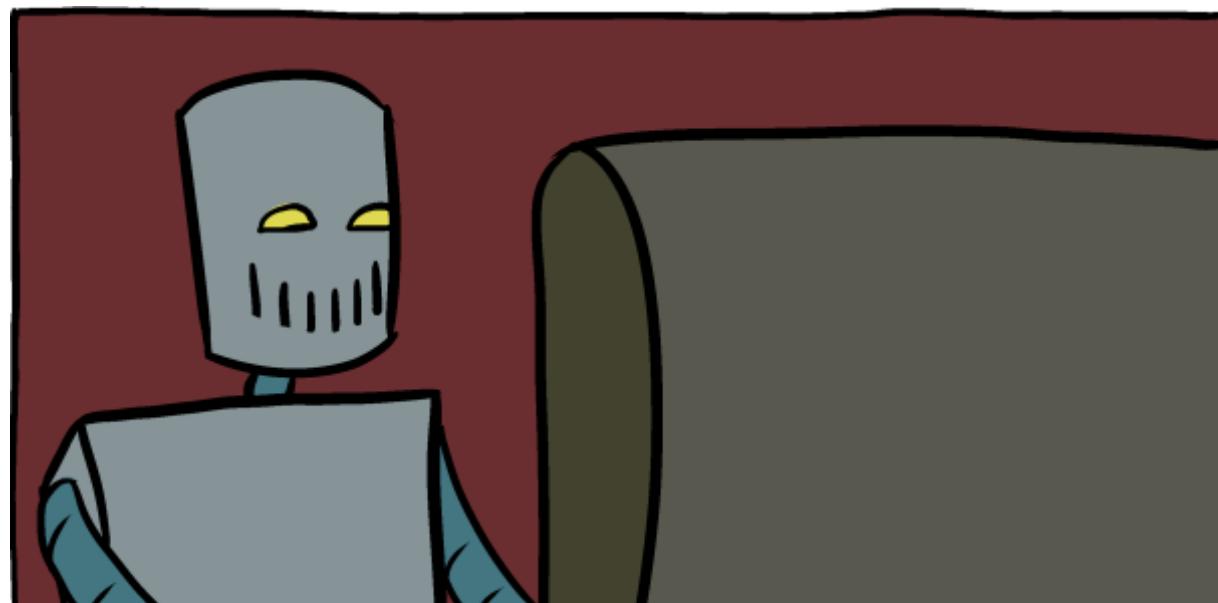
Copyright (憲C) 2017, 2019 宅色夫

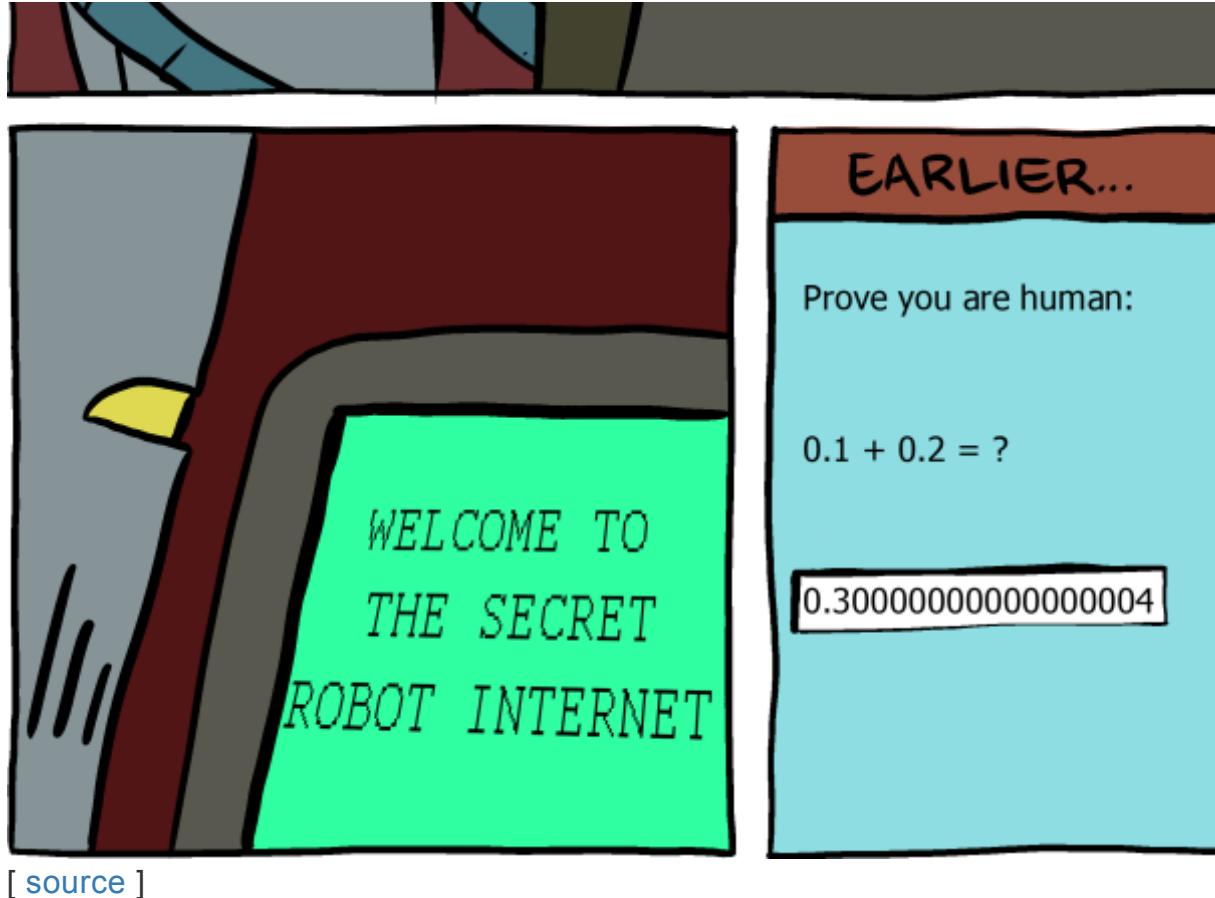
[直播錄影\(上\)](#)

[直播錄影\(下\)](#)

從一則新聞、動畫，和漫畫談起

- $1/2 + 1/3$, 要孩子怎麼討論？





[[source](#)]

不同程式語言給出相似的執行結果: Floating Point Math

Python 2.7 在 GNU/Linux 的執行:

```
>>> 1 - 0.1  
0.9  
>>> 0.1 - 0.01  
0.0900000000000001
```

後者顯然比預期數值 0.09 略大

```
>>> 0.1 - 0.01 - 0.1  
-0.00999999999999995
```

而 `0.1 - 0.01 - 0.1` 又會得到比預期數值 `-0.01` 略大的結果，有辦法讓電腦精準地表達和運算數值嗎？

電腦不是只有二進位

電腦科學家 Donald E. Knuth 在《The Art of Computer Programming》第 2 卷說：

“Perhaps the prettiest number system of all is the balanced ternary notation”

這裡的 *ternary* 意思是三個的、三個一組的、三重的，也稱為 base-3，顧名思義，不是只有 0 或 1，而是將可能的狀態擴充為 `0`, `1`, `2`，在 balanced ternary 中，就是 `-1`, `0`, `+1` 等三個可能狀態，又可以簡寫為 `-`, `0`, `+`。

the ternary values as being “balanced” around the mid-point of 0. The same rules apply to ternary as to any other numeral system: The right-most symbol, R, has its own value and each successive symbol has its value multiplied by the base, B, raised to the power of its distance, D from R.

考慮以下 balanced ternary:

$$\begin{aligned} + + + - 0 &= (1 * 3^4) + (1 * 3^3) + (1 * 3^2) + (-1 * 3^1) + 0 \\ &= 81 + 27 + 9 + -3 \\ &= 114 \end{aligned}$$

乍看沒什麼特別的，但當我們考慮 -114 的表示法時，就有趣了：

$$\begin{aligned} ---+0 &= (-1 * 3^4) + (-1 * 3^3) + (-1 * 3^2) + (1 * 3^1) + 0 \\ &= -81 + -27 + -9 + 3 \\ &= -114 \end{aligned}$$

也就是把所有的 $+$ 和 $-$ 對調，就不用像在 2 進位表示法中，需要特別考慮 signed 和 unsigned。

balanced ternary 的作用不僅在一致的方式去表達數值，還可用於浮點數。以下是 10 進位的 0.2 對應的 balanced ternary 表示法：

$$\begin{aligned} 0.++-- &= 0 + (1 * (3^{-1})) + (-1 * (3^{-2})) + (-1 * (3^{-3})) + (1 * (3^{-4})) \\ &= 0.33 + -0.11 + -0.03 + 0.01 \\ &= 0.2 \end{aligned}$$

如何表達 10 進位的 0.8 呢？既然 $0.8 = 1 - 0.2$ ，我們做以下表示：

$$\begin{aligned} +.-++- &= 1 + (-1 * (3^{-1})) + (1 * (3^{-2})) + (1 * (3^{-3})) + (-1 * (3^{-4})) \\ &= 1 + -0.33 + 0.11 + 0.03 + -0.01 \\ &= 0.8 \end{aligned}$$

把最開頭的 0 換成 $+1$ ，然後小數點後的 $+$ 和 $-$ 對調即可。

參考資料：

- [The Balanced Ternary Machines of Soviet Russia](#)
- [The Tech Behind IOTA Explained](#)

These 3 states perform transaction very balanced, which is quite helpful to build a self-organizing and self-sustaining network like the tangle.

數值表達方式和阿貝爾群

數學中的「群」是個由我們定義的二元運算的集合，這裡的二元運算稱為「加法」，表示為符號 $+$ 。為了讓一個集合 G 成為群，必須定義加法運算並使之具有以下 4 個特性：

1. 封閉性：若 a 和 b 是集合 G 中的元素，於是 $(a + b)$ 也是集合 G 中的元素；
2. 結合律： $(a + b) + c = a + (b + c)$ ；
3. 存在單位元素 0 ，使得 $a + 0 = 0 + a = a$ ；
4. 每個元素都有反元素（或稱「逆元」），也就是說：對於任意 a ，存在 b ，使得 $a + b = 0$ ；

倘若我們追加下述條件：

5. 交換律： $a + b = b + a$ ；

那麼，稱這個群為阿貝爾群 (Abelian group)。

嚴格定義後，我們再回顧通常概念的「加法」時，就可發現，整數的集合 \mathbb{Z} 就是一個群（同時也是個阿貝爾群），但是，自然數的集合 (\mathbb{N}) 就不是群，因為 \mathbb{N} 不滿足上述第 4 個特性。

為何我們要大費周章去表達「群」的特性呢？一旦我們證明它具備上述 4 個特性，那麼就可自由地獲取到一些其他特性。像是：

- 單位元素是唯一的；
- 反元素也是唯一的，即：對於每一個 a ，存在唯一的一個 b ，使得 $a + b = 0$ （我們可以將 b 寫成 $-a$ ）。

以電腦的數值系統來說，整數（包含 sign 和 2's complement）加法形成阿貝爾群，實數 (\mathbb{R}) 的加法也形成阿貝爾群，但我們必須考慮四捨五入（或無條件捨入）對這些屬性的影響。更甚者，由於 overflow 的考慮，導致儘管 x 和 y 都是實數，結果可能截然不同。

回到電腦的資料表示法，假設我們用 4 個 bits 來表示，像是 0000 表示 0，我們可以額外引入一個 bit 來表示 +/- (sign bit)，但事實上我們可將上述特性考慮進去，引入反元素，讓每個正整數都可有一個對應的反元素，也是負數，這也是為何對應的正整數 bit-wise not 後 +1。1000 是唯一沒有對應正整數的數值，因此有號數的負整數會比正整數多一個。

在 IEEE 754 的單精度運算符點數中 ([好看的解說影片](#)，我說板書)，表達式 $(3.14 + 1e10) - 1e10$ 求值會得到 0.0 —— 因為捨入，數值 3.14 會丟失。另一方面，表達式 $3.14 + (1e10 - 1e10)$ 會得到數值 3.14。

- 延伸閱讀: [浮點數的美麗與哀愁](#)

作為阿貝爾群，大多數值的浮點數加法都有反元素，但是 INF (無窮) 和 NaN 是例外情況，因為對任何 x ，都有 $\text{NaN} + fx = \text{NaN}$ ；

浮點數加法不具有結合性，這是缺乏的最重要「群」特性。知道這些後，對我們寫程式有什麼影響呢？

衝擊可大了！

假設 C 語言編譯器即將處理以下程式碼：

```
x = a + b + c;  
y = b + c + d;
```

編譯器可能為了省下一道浮點數運算，而產生以下中間程式碼：(code motion 技巧，詳見 [編譯器和最佳化原理篇](#))

```
t = b + c;  
x = a + t;  
y = t + d;
```

但對於 x 來說，這樣的計算方式可能會導致和原始數值截然不同的結果，因為它運用了加法運算的不同的結合方式！

單精度浮點數運算中：

- $(1e20 * 1e20) * 1e20$ 為 +INF
- $1e20 * (1e20 * 1e-20)$ 為 1e20
- $1e20 * (1e20 - 1e20)$ 為 0.0
- $1e20 * 1e20 - 1e20 * 1e20$ 為 NaN

Integer Overflow

- [神一樣的進度條](#)
- [波音 787 不再「夢幻」](#)
 - 波音 787 的電力控制系統在 248 天電力沒中斷的狀況下，會自動關機，為此 FAA (美國聯邦航空管理局) 告知應每 120 天重開機，看來「重開機治百病」放諸四海都通用？這當然是飛安的治標辦法，我們工程人員當然要探究治本議題。
 - 任教於美國 [Carnegie Mellon University](#) (CMU) 的 Phil Koopman 教授指出，這其實就是 integer overflow，再次驗證「失之毫釐，差之千里」的道理。
 - 我們先將 248 天換成秒數：
 - $248 \text{ days} * 24 \text{ hours/day} * 60 \text{ minute/hour} * 60 \text{ seconds/minute} = 21,427,200$
 - 這個數字若乘上 100，繼續觀察：
 - $0x7FFFFFFF$ (32-bit 有號數最大值) $= 2147483647 / (24 * 60 * 60) = 24855 / 100 = 248.55 \text{ days.}$
 - 看出來了嗎？每 1/100 秒紀錄在 32-bit signed integer，然後遇到 overflow
 - [Counter Rollover Bites Boeing 787](#)

- Deep Impact (2005)
- Ariane 5 (1996)
 - [detail report](#) : a data conversion from 64-bit floating point to 16-bit signed integer value

其他 integer overflow 案例:

- OpenSSH 2002 security hole
- Year 2038 problem
- Youtube Gangnam Style overflows
- Diablo III Real Money Action House integer overflow
- Lempel-Ziv-Oberhumer (LZO) algorithm
- OpenSSL integer underflow leading to buffer overflow in base64 decoding
- Trend Micro Discovers Vulnerability That Renders Android Devices Silent
- IPv4 address Lexhaustion , A bug and a crash — The explosion of Ariane 5 rocket
- Integer overflow in Mozilla Firefox 3.5.x before 3.5.11 and 3.6.x before 3.6.7
- CVE-2015-1593 - Linux ASLR integer overflow: Reducing stack entropy by four
- Integer overflow in Bitcoin software, Bitcoinwiki - Value overflow incident
- SSH CRC32 attack detection code contains remote integer overflow
- .NET Framework EncoderParameter integer overflow vulnerability
- The classic videogame Donkey Kong has an infamous 'kill screen', where the game stops working. But why? =>integer overflow
- Adobe Flash Player casi32 Integer Overflow
- ngx_http_close_connection integer overflow
- PHP Integer Overflow Affects Tenable's Security Center
- Therac-25 radiation overdose

- CVE-2014-3669: Integer overflow in unserialize() PHP function
- MS15-034 – Range Header Integer Overflow
- Python Integer Overflow in 'bufferobject.c' Lets Users Obtain Potentially Sensitive Information
- Super Mario Bros life

Integer Overflow 案例分析

- 2002 年 FreeBSD [53]

```
#define KSIZE 1024
char kbuf[KSIZE];
int copy_from_kernel(void *user_dest, int maxlen) {
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

假設懷有惡意的程式設計師將「負」的數值作為 maxlen 帶入 `copy_from_kernel`，會有什麼問題？

- 2002 年 External data representation (XDR) [62]

```
void *copy_elements(void *ele_src[], int ele_cnt, int ele_size) {
    void *result = malloc(ele_cnt * ele_size);
    if (result==NULL) return NULL;
    void *next = result;
    for (int i = 0; i < ele_cnt; i++) {
```

```
        memcpy(next, ele_src[i], ele_size);
        next += ele_size;
    }
    return result;
}
```

假設懷有惡意的程式設計師將 `ele_cnt = 220 + 1`, `ele_size = 212` 帶入，會有什麼問題？

二進位

搭配觀看影片 [How to count to 1000 on two hands](#), 記得開啟 YouTube 字幕

- 萊布尼茲在 1678 年發明二進位表示法
 - [[source](#)] 萊布尼茲研究 Pascal 在 1642 年設計製造的十進位數字計算機，並在 1671 年設計出能作加減乘除的分級計算機設計。藉由多次的加減來實現乘除，還可以求平方根。這過程中，他發現平時用起來很方便的十進位計數法，搬到機械上去實在太麻煩。
 - 為了解答「能否用較少的數碼來表示一個數呢？」這問題，萊布尼茲在 1678 年發明二進位計數法，也就是二進位。如此一來，用 0 和 1 兩個數碼就可以表示出一切數。比如用 `10` 表示 2, `11` 表示 3, `100` 表示 4, `101` 表示 5, 以此類推。
 - 大清國康熙時期，派遣傳教士白晉 (法語: Joachim Bouvet) 回到法國，白晉在 1701 年寄了一封附上兩張易經六十四卦圖的信給萊布尼茲，萊布尼茲受到啟發，稱讚八卦是「世上流傳下來的科學中最古老的紀念物」。
- George Boolean 在 1800 年介紹「邏輯代數」，後來成為「布林代數」(Boolean Algebra)

- Claude E. Shannon 於 1938 年發表布林代數對於二進制函數的應用
- 世界上只有10種人，一種是懂二進位的
- 解讀計算機編碼

運用 bit-wise operator

- 實作二進位加法器
- C 語言中，`x & (x - 1) == 0` 的數學意義
 - power of two
 - signed v.s. unsigned
- 將字元轉成小寫: 免除使用分支

```
('a' | '_') // 得到 'a'  
('A' | '_') // 得到 'a'
```

- 將字元轉為大寫: 免除使用分支

```
('a' & '_') // 得到 'A'  
('A' & '_') // 得到 'A'
```

- 大小寫互轉: 避免使用分支

```
('a' ^ ' ') // 得到 'A'  
('A' ^ ' ') // 得到 'a'
```

- XOR swap

- 交換兩個記憶體空間內的數值，可完全不用額外的記憶體來實作

```
void xorSwap(int *x, int *y) {  
    *x ^= *y;  
    *y ^= *x;  
    *x ^= *y;  
}
```

- 需要這種手法的情境：

1. 指令集允許 XOR swap 產生較短的編碼 (某些 DSP);
2. 考慮到暫存器數量在某些硬體架構 (如 ARM) 非常有限，register allocation 就變得非常棘手，這時透過 XOR swap 可降低這方面的衝擊；
3. 在微處理器中，記憶體是非常珍貴的資源，此舉可降低記憶體的使用量；
4. 在加解密的實作中，需要常數時間的執行時間，因此保證 swap 兩個數值的執行成本要固定 (取決於指令週期數量)；

- 避免 overflow

- 比方說 $(x + y) / 2$ 這樣的運算，有個致命問題在於 $(x + y)$ 可能會導致 overflow (考慮到 x 和 y 都接近 `UINT32_MAX`，亦即 32-bit 表示範圍的上限之際)
 - Binary search 的演算法提出之後十年才被驗證
- 於是我們可以改寫為以下：

```
(x & y) + ((x ^ y) >> 1)
```

- 用加法器來思考: `x & y` 是進位, `x ^ y` 是位元和, `/ 2` 是向右移一位
- 位元相加不進位的總和: `x ^ y`; 位元相加產生的進位值: `(x & y) << 1`
- $$\begin{aligned} x + y &= x ^ y + (x \& y) << 1 \\ \text{所以 } (x + y) / 2 &= (x + y) >> 1 \\ &= (x ^ y + (x \& y) << 1) >> 1 \\ &= (x \& y) + ((x ^ y) >> 1) \end{aligned}$$

- 以下 C 語言程式的 DETECT 巨集能做什麼？

```
#if LONG_MAX == 2147483647L
#define DETECT(X) \
    (((X) - 0x01010101) & ~(X) & 0x80808080)
#else
#if LONG_MAX == 9223372036854775807L
#define DETECT(X) \
    (((X) - 0x0101010101010101) & ~(X) & 0x8080808080808080)
#else
#error long int is not a 32bit or 64bit type.
#endif
#endif
```

- 巨集 `DETECT` 在偵測什麼？
 - Detect NULL

測試這程式時，要注意到由於 `LONG_MAX` 定義在 `<limits.h>` 裡面，因此要記得作 `#include <limits.h>`。這個巨集的用途是在偵測是否為 0 或者說是否為 NULL char '\0'，也因此，我們可以在 iOS 的原始程式碼 `strlen` 的實作中看到這一段。那，為什麼這一段程式碼可以用來偵測 NULL char ?

我們先思考 `strlen()` 該怎麼實做，以下實作一個簡單的版本

```
unsigned int strlen(const char *s) {
    char *p = s;
    while (*p != '\0') p++;
    return (p - s);
}
```

這樣的版本有什麼問題？雖然看起來精簡，但是因為他一次只檢查 1byte，所以一旦字串很長，他就會處理很久。另外一個問題是，假設是在 32-bit 的 CPU 上，一次是處理 4-byte (32-bit) 大小的資訊，不覺得這樣很浪費嗎？

為了可以思考這樣的程式，我們由已知的計算方式來逆推原作者可能的思考流程，首先先將計算再簡化一點點，將他從 $((X) - 0x01010101) \& \sim(X) \& 0x80808080$ 變成

```
((X) - 0x01) \& \sim(X) \& 0x80
```

還是看不懂，將以前學過的笛摩根定理套用上去，於是這個式子就變成了

```
\sim(\sim(X - 0x01) | X) \& 0x80
```

再稍微調整一下順序

```
\sim(X | \sim(X - 0x01)) \& 0x80
```

所以我們就可進行分析

- $X | \sim(X - 0x01)$ => 取得最低位元是否為 0，並將其他位元設為 1

- $X = 0000\ 0011 \Rightarrow 1111\ 1111$
- $X = 0000\ 0010 \Rightarrow 1111\ 1110$
- 想想 $0x80$ 是什麼? $0x80$ 是 $1000\ 0000$ ，也就是 1-byte 的最高位元

上面這兩組組合起來，我們可以得到以下結果

- $X = 0 \Rightarrow 1000\ 0000 \Rightarrow 0x80$
- $X = 1 \Rightarrow 0000\ 0000 \Rightarrow 0$
- $X = 2 \Rightarrow 0000\ 0000 \Rightarrow 0$
- ...
- $X = 255 \Rightarrow 0000\ 0000 \Rightarrow 0$

於是我們知道，原來這樣的運算，如果一個 byte 是 0，那經由這個運算得到的結果會是 $0x80$ ，反之為 0。

再將這個想法擴展到 32-bit，是不是可以想到說在 32bit 的情況下，0 會得到 $0x80808080$ 這樣的答案？我們只要判斷這個數值是不是存在，就可以找到 '\0' 在哪了！

參考資料：

- [Hacker's Delight](#)
- <http://www.hackersdelight.org/corres.txt>
- FreeBSD 的 `strlen(3)`
- [Bug 60538 - \[SH\] improve support for cmp/str insn](#)
- [Bit Twiddling Hacks](#)

應用：

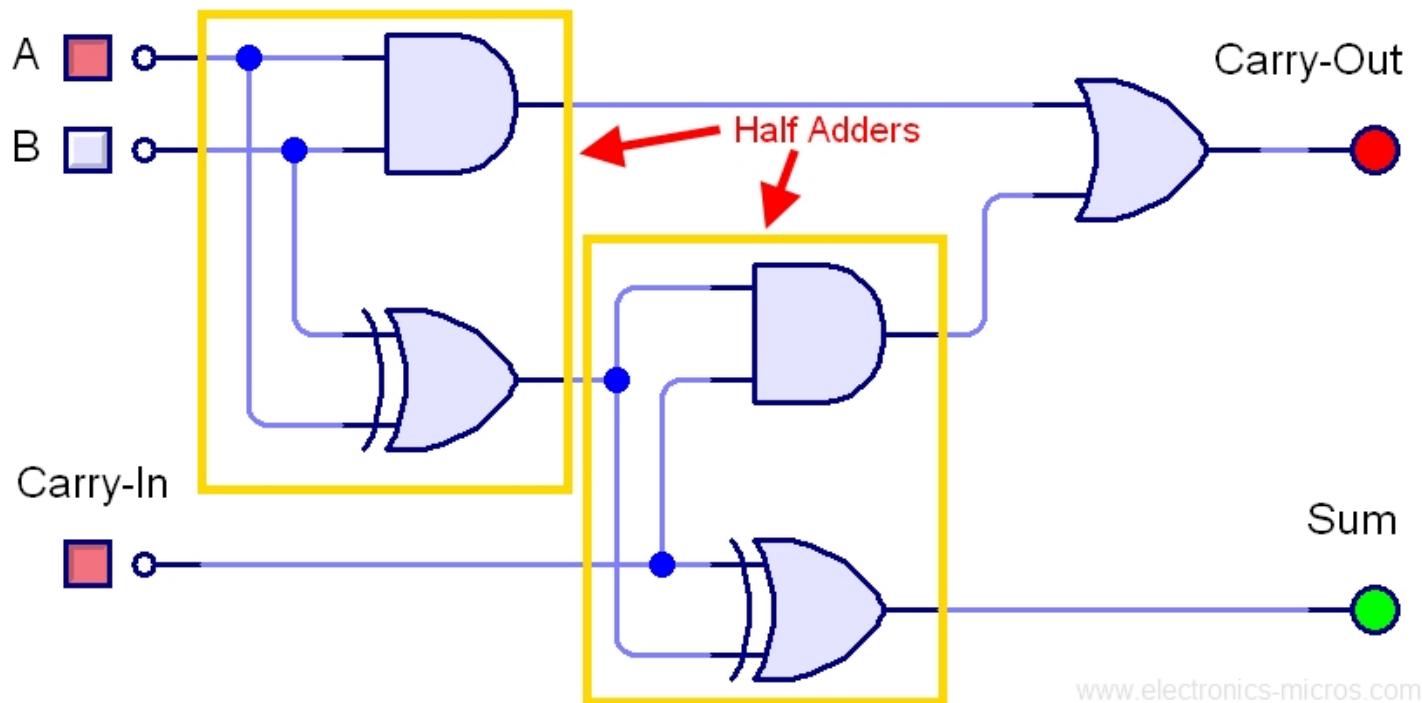
- [newlib 的 `strlen`](#)

- newlib 的 strcpy
- SSE 4.2 最佳化版本: [Implementing strcmp, strlen, and strstr using SSE 4.2 instructions](#)

算術完全可用數位邏輯實做

只能使用位元運算子和遞迴，在 C 程式中實做兩個整數的加法，可行嗎？

回顧 [加法器](#) 的實做：



思考以下程式碼：

```
int add(int a, int b) {
    if (b == 0) return a;
    int sum = a ^ b; /* 相加但不進位 */
    int carry = (a & b) << 1; /* 進位但不相加 */
    return add(sum, carry);
}
```

延伸閱讀: [How to simulate a 4-bit binary adder in C](#)

Count Leading Zero

當我們計算 $\log_2 N$ (以 2 為底的對數) 時, 實際只要算高位有幾個 0's bits. 再用 31 減掉即可。

```
int BITS = 31;
for (i < 32; --BITS) {
    if (N & 0x80000000) break;
    N <<= 1;
}
```

$\log_2(N)$ is BITS

當要算 $\log_{10} N$ 時, 因為 32-bit unsigned integer 最大只能顯示 4294967295U, 所以 32-bit LOG10() 的值只可能是 0 ~ 9.

這時可透過查表法, 以省去除法的成本。

```
unsigned int vals[] = {
    1UL,
    10UL,
    100UL,
```

```

        1000UL,
        10000UL,
        100000UL,
        1000000UL,
        10000000UL,
        100000000UL,
        1000000000UL,
    } ;
    for (i = 0; i < (nr - 1); ++i) { // 9
        if (N >= vals[i] && N < vals[i + 1]) { // 8
            break; // 1
        }
    }
}

```

換句話說，計算 $\log_2 N$ 時，知道「高位開頭有幾個 0」就成為計算的關鍵操作。

延伸閱讀: [Fast computing of log2 for 64-bit integers](#)

- 類似 De Bruijn 演算法

- 64-bit version

```

const int tab64[64] = {
    63, 0, 58, 1, 59, 47, 53, 2,
    60, 39, 48, 27, 54, 33, 42, 3,
    61, 51, 37, 40, 49, 18, 28, 20,
    55, 30, 34, 11, 43, 14, 22, 4,
    62, 57, 46, 52, 38, 26, 32, 41,
    50, 36, 17, 19, 29, 10, 13, 21,
    56, 45, 25, 31, 35, 16, 9, 12,
    44, 24, 15, 8, 23, 7, 6, 5
};

```

```
int log2_64 (uint64_t value)
{
    value |= value >> 1;
    value |= value >> 2;
    value |= value >> 4;
    value |= value >> 8;
    value |= value >> 16;
    value |= value >> 32;
    return tab64[((uint64_t)((value - (value >> 1)) * 0x07EDD5E59A4E28C2)) >> 58];
}
```

- 32-bit version

```
const int tab32[32] = {
    0, 9, 1, 10, 13, 21, 2, 29,
    11, 14, 16, 18, 22, 25, 3, 30,
    8, 12, 20, 28, 15, 17, 24, 7,
    19, 27, 23, 6, 26, 5, 4, 31
};
int log2_32 (uint32_t value)
{
    value |= value >> 1;
    value |= value >> 2;
    value |= value >> 4;
    value |= value >> 8;
    value |= value >> 16;
    return tab32[(uint32_t)(value * 0x07C4ACDD) >> 27];
}
```

gcc 提供 built-in Function:

- `int __builtin_clz (unsigned int x)`

- Returns the number of leading 0-bits in x, starting at the most significant bit position.
- If x is 0, the result is undefined.

可用來實做 log2:

```
#define LOG2(X) ((unsigned) \
(8 * sizeof (unsigned long long) - \
__builtin_clzll(X) - 1))
```

那該如何實做 clz 呢？

iteration version

```
int __clz(uint32_t x) {
    int n = 32, c = 16;
    do {
        uint32_t y = x >> c;
        if (y) { n -= c; x = y; }
        c >>= 1;
    } while (c);
    return (n - x);
}
```

binary search technique

```
int __clz(uint32_t x) {
    if (x == 0) return 32;
    int n = 0;
    if (x <= 0x0000FFFF) { n += 16; x <<= 16; }
    if (x <= 0x00FFFFFF) { n += 8; x <<= 8; }
```

```
    if (x <= 0xFFFFFFFF) { n += 4; x <= 4; }
    if (x <= 0x3FFFFFFF) { n += 2; x <= 2; }
    if (x <= 0x7FFFFFFF) { n += 1; x <= 1; }
    return n;
}
```

byte-shift version

```
int __clz(uint32_t x) {
    if (x == 0) return 32;
    int n = 1;
    if ((x >> 16) == 0) { n += 16; x <= 16; }
    if ((x >> 24) == 0) { n += 8; x <= 8; }
    if ((x >> 28) == 0) { n += 4; x <= 4; }
    if ((x >> 30) == 0) { n += 2; x <= 2; }
    n = n - (x >> 31);
    return n;
}
```

- `ffs()` 會回傳給定數值的 first bit set 的位置
 - 例如 128 在 32-bit 表示為 0b10000000，`ffs(128)` 會回傳 8
 - 129 在 32bit 表示為 0b10000001，`ffs(129)` 會回傳 1

延伸閱讀: [Bit scanning equivalencies](#)

省去迴圈

考慮以下 C 程式,解說在 32-bit 架構下具體作用(不是逐行註解), 以及能否避開用迴圈？

```
int func(unsigned int x) {
    int val = 0; int i = 0;
    for (i = 0; i < 32; i++) {
        val = (val << 1) | (x & 0x1);
        x >>= 1;
    }
    return val;
}
```

這段程式的作用是逐位元反轉順序，如下面測試所示，顛倒後位元不足 32bit 者，全部補 0

```
-----input number 99-----
2bit= 1100011
val = 11000110000000000000000000000000
-----output number -973078528-----

-----input number 198-----
2bit= 11000110
val = 11000110000000000000000000000000
-----output number 1660944384-----

-----input number 297-----
2bit= 100101001
val = 10010100100000000000000000000000
-----output number -1803550720-----

-----input number 396-----
2bit= 110001100
val = 11000110000000000000000000000000
-----output number 830472192-----

-----input number 4294967281-----
2-bit= 1111111111111111111111111111111111001
```

```
val  = 10001111111111111111111111111111  
-----output number -1879048193-----
```

參考 [Reverse integer bitwise without using loop](#), 將原本的 for 迴圈變更為 bit-wise 操作:

```
new = num;  
new = ((new & 0xffff0000) >> 16) | ((new & 0x0000ffff) << 16);  
new = ((new & 0xff00ff00) >> 8) | ((new & 0x00ff00ff) << 8);  
new = ((new & 0xf0f0f0f0) >> 4) | ((new & 0x0f0f0f0f) << 4);  
new = ((new & 0xcccccccc) >> 2) | ((new & 0x33333333) << 2);  
new = ((new & 0xaaaaaaaa) >> 1) | ((new & 0x55555555) << 1);
```

在不使用迴圈的情況下，可以做到一樣的功能。

延伸閱讀:

- [你所不知道的 C 語言: 浮點數運算](#)
- [CS:APP 第 2 章重點提示和練習](#)

Bits Twiddling Hacks 解析: ([一](#)), ([二](#)), ([三](#))

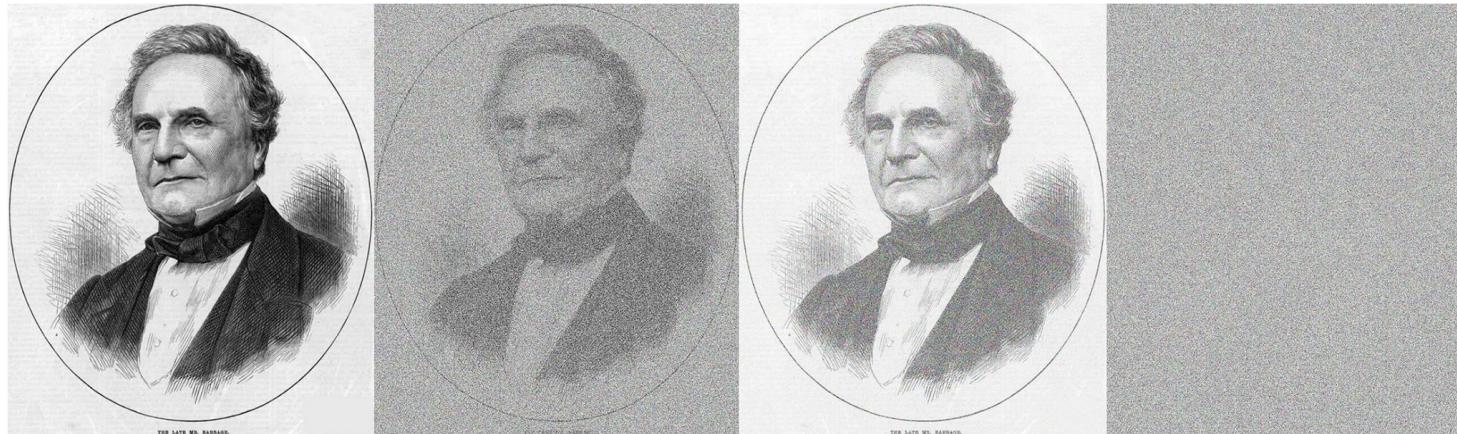
加解密的應用

Caesar shift cipher

- 把 A-Z 這 26 個字母表示成 A=0, B=1, ..., Z=25，然後給任意一個 KEY，把訊息的字母加上 KEY 之後 mod 26 就會得到加密之後的訊息。假設 KEY=19，那麼原本的訊息例如 HELLO (7 4 11 11 14) 經過 cipher 後 (26 23 30 30 33) mod 26 => (0 23 4 4 7) 會變成 AXEEH 的加密訊息。

XOR

- 假設有一張黑白的相片是由很多個0 ~255 的 pixel 組成 (0 是黑色, 255 是白色), 這時候可以用任意的 KEY (00000000_2 - 11111111_2) 跟原本的每個 pixel 做運算, 如果使用 AND (每個 bit 有 75% 機率會變成 0), 所以圖會變暗。如果使用 OR (每個 bit 有 75% 機率會變 1), 圖就會變亮。這兩種幾乎都還是看的出原本的圖片, 但若是用 XOR 的話, 每個 bit 變成 0 或 1 的機率都是 50%, 所以圖片就會變成看不出東西的雜訊。



上圖左 1 是原圖，左 2 是用 AND 做運算之後，右 2 是用 OR 做運算之後，右 1 是用 XOR，可見使用 XOR 的加密效果最好。

已知 X, Y 是 random variable over $\{0,1\}^n$, X 是 independent uniform distribution, 則 $Z = X \text{ xor } Y$ 也會是 uniform distribution。附圖是用 truth table 列舉證明, $n = 2$ 的真值表:

An important property of XOR

Thm: Y a rand. var. over $\{0,1\}^n$, X an indep. uniform var. on $\{0,1\}^n$

Then Z := Y \oplus X is uniform var. on $\{0,1\}^n$

Proof: (for $n=1$)

$$\Pr[Z=0] =$$

Y Pr		Y X Pr		
0	p_0	0	0	$p_0/2$
1	p_1	0	1	$p_0/2$
		1	0	$p_1/2$
		1	1	$p_1/2$

於是我們可以對 X 作 xor, 將任意分佈的 random number 轉為 uniform distribution

完整證明: [How to prove uniform distribution of \$m \oplus k\$ if \$k\$ is uniformly distributed?](#)

參考資料：[Ciphers vs. codes](#)

待整理

- [awesome-bits](#): A curated list of awesome bitwise operations and tricks

Published on  HackMD

 53343

 6

