1997: Fuga da Bitwise

Bob Hauk: *Mi ucciderai ora, Jena?*
Jena (Snake) Plissken: *Sono troppo stanco... Forse più tardi...*

Nel capolavoro **1997: Fuga da New York** il grande **John Carpenter** ci mostrava come scappare da una tristissima New York trasformata in prigione federale. Ecco, l'argomento di questo articolo è uno di quegli argomenti che inducono alla fuga molti valorosi Programmatori **C/C++…** "*ehm… oggi non ho tempo, puoi occupartene tu delle operazioni di bitwise? Poi domani mi farai vedere…*"



…a me Bitwise non me l'ha mai detto nessuno…
Cominciamo con un piccolo gioco: alzi la mano chi ha scritto recentemente codice che usa le **operazioni bit a bit**. Oppure, alzi la mano, senza prima andare a rileggersi un manuale del **C/C++**, chi di voi sa usare e/o descrivere perfettamente le **operazioni bit a bit**. Uhm… vedo poche mani alzate. Il fatto è che le **bitwise operations** sono una di quelle parti del **C/C++** un po' misconosciute, di uso dubbio e infrequente, insomma una di quelle parti di cui, per mancanza di pratica ci si scorda.

E, oltretutto, devo dire che sulle **operazioni di bitwise** il fantastico **K&R** non è particolarmente chiaro e dettagliato, le tratta (*come sono*) come un argomento di nicchia, e non ti coinvolge con decine di divertenti esempi che ti aiutano a memorizzare definitivamente l'argomento, anzi, per quel che ricordo dell'ultima volta che lo lessi, sono un paio di pagine che scivolano via e che hai già dimenticato quando passi al prossimo capitolo. Beh anche la bibbia **K&R** (che io considero sacra) ha alcuni punti non proprio coinvolgenti.

Rinfreschiamo: gli operatori di **bitwise** (che operano sui singoli bit) sono:

"&" AND "|" OR "^" XOR "~" NOT (complemento a 1) "<<" SHIFT a sinistra ">>" SHIFT a destra N.B.:
- il NOT e' un operatore unario: opera su un solo argomento indicato sulla destra.
- gli shift sono operatori unari: operano su un solo argomento indicato sulla sinistra.

Adesso, senza dilungarci in noiosi sproloqui, passiamo a una piccola tabella e ad alcuni semplici esempi pratici. Ecco la tabella:

"&": il risultato è 1 se i due operandi valgono 1. Altrimenti 0. "|": il risultato è 0 se i due operandi valgono 0. Altrimenti 1. "^": il risultato è 1 se i due operandi sono diversi. Altrimenti 0. "~": il risultato è 1 se l'operando vale 0. Altrimenti 0. "<< n": il risultato è l'operando con tutti i bit spostati a sinistra di n posizioni. ">> n": il risultato è l'operando con tutti i bit spostati a destra di n posizioni.

Ed ecco i semplici esempi pratici:

AND
```
int a = 74; // 0 1 0 0 1 0 1 0
int b = 174; // 1 0 1 0 1 1 1 0
int c = a & b; // 0 0 0 0 1 0 1 0 risultato c=10 OR
int a = 74; // 0 1 0 0 1 0 1 0
int b = 174; // 1 0 1 0 1 1 1 0
```

```
int c = a | b; // 1 1 1 0 1 1 1 0 risultato c=238 XOR
int a = 74; // 0 1 0 0 1 0 1 0
int b = 174; // 1 0 1 0 1 1 1 0
int c = a & b; // 1 1 1 0 0 1 0 0 risultato c=228 NOT
int a = 74; // 0 1 0 0 1 0 1 0
int b = ~b; // 1 0 1 1 0 1 0 1 risultato c=181 SHIFT a sinistra
int a = 74; // 0 1 0 0 1 0 1 0
int b = a << 2; // 0 0 1 0 1 0 0 0 risultato c=296 SHIFT a destra
int a = 74; // 0 1 0 0 1 0 1 0
int b = a >> 2; // 0 0 0 1 0 0 1 0 risultato c=18
```

Notare che nelle operazioni di **shift** i bit nuovi che **entrano a destra** (nello **shift** a sinistra) valgono 0, e i bit nuovi che **entrano a sinistra** (nello **shift** a destra) valgono 0.

Notare anche che lo **shift** a destra equivale a una divisione per multipli di 2 (**>>1** è una divisione per 2, **>>2** è una divisione per 4, ecc.), mentre lo **shift** a sinistra equivale a una moltiplicazione per multipli di 2 (**<<1** è una moltiplicazione per 2, **<<2** è una moltiplicazione per 4, ecc.). Queste operazioni di moltiplicazione e divisione sono molto veloci, e si potrebbe **essere tentati a usarle per velocizzare il codice**: beh, prima di farlo rileggetevi (o leggetevi) **questo**.

E aggiungo un avvertimento: in base alla dimensione del tipo del operando e alla presenza o meno del bit di segno, le moltiplicazioni e divisioni con shift possono dare **risultati inaspettati**. Anzi, andiamoci subito alle ***dolenti note***, togliamoci il pensiero: vediamo un esempio:

```
#include <stdio.h> void main()
{ // 1) SHIFT a sinistra con unsigned int unsigned int a, b; a = 74; // 0 0 1 0 0 1 0 1 0 b = a << 2; // 1 0 0 1 0 1 0 0 0 risultato b=296 printf("var = %d; var << 2 = %d\n", a, b); // 2) SHIFT a sinistra con unsigned char unsigned char c, d; c = 74; // 0 1 0 0 1 0 1 0 d = c << 2; // 0 0 1 0 1 0 0 0 risultato d=40 printf("var = %d; var << 2 = %d\n", c, d); //
```

3) SHIFT a destra con signed char (o int) char e, f; e = -4; // 1 1 1 1 1 1 0 0 f = e >> 2; // 1 1 1 1 1 1 1 1 risultato f=-1 printf("var = %d: var >> 2 = %d\n", e, f); // N.B.: senza estensione di segno sarebbe: // e = -4; // 1 1 1 1 1 1 0 0 // f = e >> 2; // 0 0 1 1 1 1 1 1 risultato f=63
}

Il **caso 1** è, evidentemente il caso funzionante: un **int** è molto più grande degli 8 bit usati per rappresentare il numero di partenza (74), per cui lo **shift** non perde nessun 1 sulla sinistra (è questo il possibile problema) e il risultato è corretto (74×4=296). Se usiamo però (**caso 2**) un **char** (8 bit) durante lo **shift** perdiamo un 1 e il risultato va in overflow (74×4=40 ??). **Quindi attenzione!**

Il **caso 3**, poi, è ancora più subdolo: facendo operazioni con segno (nei casi 1 e 2 ho usato **variabili unsigned** proprio in preparazione al punto 3) e usando valori negativi, possono succedere cose strane: per la rappresentazione stessa dei numeri negativi in binario (**complemento a 2**) il bit più a sinistra (MSB) è il bit di segno, e, in questo caso l'operazione di **shift** è **machine-dependent**: in base al tipo di CPU possiamo disporre o no dell'**estensione di segno** (di default, ad esempio, su macchine Intel), per cui lo **shift** dell'esempio può dare il risultato aspettato (-4/4=-1) o un risultato completamente diverso (-4/4=63 ??). **Di nuovo: attenzione!**

E adesso è il momento di mostrare alcuni esempi pratici di uso di quanto esposto, che altrimenti, sarebbe *know-how* fine a se stesso: vediamo come leggere lo stato dei singoli bit di una **word** usando una maschera:

#include <stdio.h> void main()
{ // uso di una maschera per leggere i bit di una word unsigned char mask = 1; // 0 0 0 0 0 0 0 1 unsigned char word = 74; // 0 1 0 0 1 0 1 0 // loop di lettura int i; for (i = 0; i < 8; i++) printf("il bit %d della word è %s\n", i, (word & mask<<i) ? "ON" : "OFF");
}

semplice no? E la stessa operazione si può fare con una macro:

```
#include <stdio.h> #define INPUT(w, i) (w & 0x01<<i) void main()
{ // uso di una macro per leggere i bit di una word unsigned char i_word = 74; // 0 1 0 0 1 0 1 0 // loop di lettura int i; for (i = 0; i < 8; i++) printf("il bit %d della word è %s\n", i, INPUT(i_word, i) ? "ON" : "OFF");
}
```

E poi, visto che lo stile è sempre molto importante, vediamo un modo con una una buona estetica per leggere degli input di un dispositivo, per esempio i **fine corsa di un sistema elettromeccanico** che dobbiamo controllare col nostro amato **C**:

```
#include <stdio.h> #define INPUT_FC1 (in_word & 0x01<<0)
#define INPUT_FC2 (in_word & 0x01<<1) void main()
{ // uso di una define per ogni bit da leggere di una word unsigned char in_word = 74; // 0 1 0 0 1 0 1 0 // lettura printf("il bit FC1 della word è %s\n", INPUT_FC1 ? "ON" : "OFF"); printf("il bit FC2 della word è %s\n", INPUT_FC2 ? "ON" : "OFF");
}
```

Ecco, l'esempio appena mostrato indica una maniera, **semplice ed elegante,** per descrivere degli **input** (usando dei mnemonici auto-esplicativi) che può risultare utile per scrivere del Software di controllo di dispositivi Hardware, facile da leggere e da manutenere.

Facile da leggere e da manutenere: **questa l'ho già sentita…** ah si: è come dovrebbe essere tutto il **S/W** che scrive un **Analista Programmatore** (*…ma questa è un'altra storia…*).

Ciao e al prossimo post!

https://italiancoders.it/1997-fuga-da-bitwise/