Manejo de Bits en Lenguaje C

Diego Feroldi
feroldi@fceia.unr.edu.ar

Arquitectura del Computador*
Departamento de Ciencias de la Computación
FCEIA-UNR

*Actualizado 22 de junio de 2025 (D. Feroldi, feroldi@fceia.unr.edu.ar)

Índice

1. Introducción

En el lenguaje C, los operadores de bits permiten manipular directamente los bits que conforman los datos de tipo entero. Estas operaciones son fundamentales en programación de bajo nivel, ya que proporcionan un control preciso sobre la representación binaria de los valores, lo que resulta útil en optimización de código, manipulación de registros, criptografía y control de hardware, entre otras aplicaciones.

El lenguaje C no define un tipo de dato específico para trabajar con bits, pero permite operar a nivel de bits utilizando variables de tipo entero. Para ello, ofrece un conjunto de operadores bit a bit que permiten realizar desplazamientos, máscaras y combinaciones lógicas entre bits. Estos operadores incluyen el AND bit a bit (`&`), OR bit a bit (`|`), XOR bit a bit (`^`), NOT bit a bit (`~`), así como los desplazamientos a la izquierda (`<<`) y a la derecha (`>>`).

Comprender el uso de estos operadores es clave para desarrollar programas eficientes y aprovechar al máximo los recursos del hardware. A lo largo de este material, exploraremos sus características, aplicaciones y ejemplos prácticos que ilustran su utilidad en distintos contextos.

2. Operadores de bits en C

La siguiente tabla presenta los principales operadores de bits (*bitwise operators*) en el lenguaje C:

Operador	**Acción**
`&`	Operación AND bit a bit
`|`	Operación OR bit a bit
`^`	Operación XOR bit a bit
`~`	Operación NOT bit a bit
`>>`	Desplazamiento a la derecha
`<<`	Desplazamiento a la izquierda

Importante: Los operadores de bits pueden ser aplicados a variables de tipo entero ya sea con signo o sin signo (`char`, `short`, `int`, `long`, etc.) y NO pueden ser aplicados a variables tipo flotante (`float`, `double`, etc.).

2.1. Operador AND

El operador AND bit a bit está definido por la siguiente tabla de verdad:

a_i	b_i	a_i `&` b_i
0	0	0
0	1	0
1	0	0
1	1	1

Supongamos que tenemos dos variables, `A=56` y `B=72`, entonces la operación AND entre estas dos variables resulta:

$$\begin{array}{rcccccccc|c} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & (56) \\ \& & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & (72) \\ \hline & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & (8) \end{array}$$

Es decir, `A&B=8`.

Notar que el resultado hubiera sido diferente si se hubiera aplicado el operador lógico `&&`. En este caso habría resultado `A&&B=1`. El operador AND lógico (`&&`) devuelve el valor booleano *true* si ambos operandos son *true*. En caso contrario, devuelve *false*. Los operandos se convierten implícitamente al tipo *bool* antes de su evaluación y el resultado de la operación es de tipo *bool*.

Una aplicación útil para el operador `&` consiste en determinar si un determinado bit de cierto número es 1 o 0.

Ejemplo 1. Si se tiene el número `A=72` y se quiere averiguar si el cuarto bit de dicho número es 1 o 0, podemos aplicar el operador `&` realizando la operación `A&A`, donde `B` es un número cuya representación binaria es $00001000b$, es decir un número cuyo cuarto bit es 1 y todos los demás son ceros. A este último número se lo denomina "máscara". Las "máscaras" son secuencias de bits que combinadas en una operación binaria con cualquier otra secuencia de bits permite modificar esta última u obtener alguna información sobre ella. Entonces:

	0	1	0	0	1	0	0	0	A=72
&	0	0	0	0	1	0	0	0	Máscara
	0	0	0	0	1	0	0	0	

Por lo tanto, al realizar `A&B` vemos que efectivamente el cuarto bit del número es 1. Ya veremos que este ejemplo se puede mejorar utilizando el operador desplazamiento.

Ejemplo 2. Un número es potencia de 2 si tiene exactamente un bit en 1, lo que se puede verificar con `n & (n - 1)`:

```c
#include <stdio.h>

int esPotenciaDeDos(int num) {
    return (num > 0) && ((num & (num - 1)) == 0);
}

int main() {
    int num = 16;

    if (esPotenciaDeDos(num))
        printf("%d es una potencia de 2\n", num);
    else
        printf("%d no es una potencia de 2\n", num);
    return 0;
}
```

2.2. Operador OR

El operador OR bit a bit está definido por la siguiente tabla de verdad:

a_i	b_i	a_i `|` b_i
0	0	0
0	1	1
1	0	1
1	1	1

Supongamos que tenemos dos variables, `A=56` y `B=72`. Entonces la operación OR entre estas dos variables resulta:

$$\begin{array}{rcccccccc} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \quad (56) \\ | & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \quad (72) \\ \hline & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \quad (120) \end{array}$$

Es decir, `A|B=120`. Notar que el resultado es distinto al obtenido si se utiliza el operador OR lógico (`||`).

El operador `|` se puede utilizar para "encender" determinados bits de un número.

Ejemplo 3. Sea el número `A=28`, si queremos poner en uno el séptimo bit podemos realizar `A=A|B`, donde `b=01000000b`:

$$\begin{array}{rcccccccl} & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \quad \text{A=28} \\ | & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \quad \text{Máscara} \\ \hline & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \end{array}$$

Podemos observar que el séptimo bit de `A` ahora es uno.

2.3. Operador XOR

El operador XOR bit a bit está definida por la siguiente tabla de verdad:

a_i	b_i	a_i `^` b_i
0	0	0
0	1	1
1	0	1
1	1	0

Supongamos que tenemos dos variables, `A=56` y `B=72`, entonces la operación XOR entre estas dos variables resulta:

$$\begin{array}{rcccccccc} & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \quad (56) \\ \hat{} & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \quad (72) \\ \hline & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \quad (112) \end{array}$$

Es decir, `A^B=112`.

El operador XOR se puede utilizar para invertir determinados bits de un número.

Ejemplo 4. Sea el número `A=28`, si queremos invertir el cuarto bit podemos realizar `A=A^B`, donde `B=00001000b`:

$$\begin{array}{rcccccccc} & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ \hat{} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

Vemos que el cuarto bit de `A` pasó de ser uno a ser cero.

2.3.1. Propiedades del operador XOR

A partir de la tabla de la verdad del operador XOR se pueden determinar las siguientes propiedades:

- Conmutativa: `A^B = B^A`
- Asociativa: `A^(B^C) = (A^B)^C`
- `A^A = 0`
- `A^0 = A`

A partir de las propiedades anteriores resulta:

- `(B^A)^A = B^0 = B`

Esto resulta útil para hacer cifrado de datos. Una cadena de texto puede ser cifrada aplicando el operador de bit XOR sobre cada uno de los caracteres utilizando una clave. Para descifrar la salida, solo hay que volver a aplicar el operador XOR con la misma clave.

Ejemplo 5. La cadena de caracteres "Hola" se puede representar utilizando código ASCII como 01001000 01101111 01101100 01100001. Esta cadena puede ser cifrada con la clave 11110011 de la siguiente manera:

$$\begin{array}{rc} & 01001000\ 01101111\ 01101100\ 01100001 \\ \hat{} & 11110011\ 11110011\ 11110011\ 11110011 \\ \hline & 10111011\ 10011100\ 10011111\ 10010010 \end{array}$$

Si a este resultado se le vuelve a aplicar el operador XOR con la misma clave volvemos a tener la cadena original:

$$\begin{array}{rc} & 10111011\ 10011100\ 10011111\ 10010010 \\ \hat{} & 11110011\ 11110011\ 11110011\ 11110011 \\ \hline & 01001000\ 01101111\ 01101100\ 01100001 \end{array}$$

2.4. Operador complemento a uno

El operador binario `~` se conoce como operador complemento a uno o también como operador NOT. Es un operador unario, es decir solo necesita un operando. La tabla de la verdad de este operador es la siguiente:

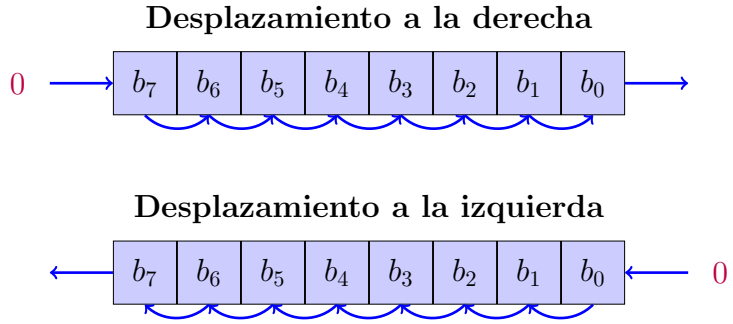
a_i	$\sim a_i$
0	1
1	0

Por lo tanto, si tenemos la variable `A=56`, resulta `~A=-57` dado que la secuencia de bits que representa a la variable `A` es $00111000b$ y entonces la secuencia que resulta de aplicarle el operador `~` resulta $11000111b$.

Ejemplo 6. La expresión `x=x&~0xff` pone los últimos 8 bits de `x` en cero. Notar que esto es independiente de la longitud del tipo de dato a diferencia de la expresión `x=x&0xffffff00` que asume que `x` tiene 32 bits.

2.5. Operadores desplazamiento de bits

Se pueden realizar desplazamientos de N bits, con $N \in \mathbb{N}$, es decir $N = 1, 2, \ldots$. Los desplazamientos de bits pueden ser en dos direcciones: desplazamiento hacia la derecha o desplazamiento hacia la izquierda. Cada vez que se hace un desplazamiento se completa con ceros en el otro lado (no se trata de una rotación).



Supongamos que tenemos el número `A=56` ($56_{10} = 00111000b$) y le realizamos un corrimiento de un bit hacia la derecha: `A>>1`. El resultado es $00011100b = 28_{10}$. Es decir, se ha dividido al número `a` por dos. Por lo tanto, desplazando el número n bits hacia la derecha se realiza la división entera del número:

$$A\texttt{>>}n = \frac{A}{2^n}$$

Análogamente, el desplazamiento a la izquierda es equivalente a multiplicar por potencias de dos:

$$A\texttt{<<}n = A \times 2^n \tag{1}$$

La ventaja de los desplazamientos es que son menos costosos que hacer las operaciones de multiplicación y división.

Importante:

- Hay que tener cuidado al realizar desplazamientos hacia la izquierda dado que se obtener resultados erróneos (con respecto a la Ecuación (1)) debido al límite en el rango de valores representables.
- También hay que tener precauciones al desplazar hacia la derecha.
- Existen dos tipos de desplazamientos hacia la derecha:
  - El "*arithmetic shift*" se completa de acuerdo al bit de signo.
  - El "*logical shift*" se completa con ceros.

Ejemplo 7. En el Ejemplo 1 vimos cómo determinar si un bit determinado de un número es 0 o 1. Dicho ejemplo se puede mejorar realizando un desplazamiento hacia la derecha luego de aplicar el operador `&`:

```
(A&B)>>3
```

El resultado es 1, indicando que el cuarto bit es efectivamente 1.

Ejemplo 8. El siguiente código cuenta cuántos bits están en 1 en un número:

```
#include <stdio.h>

int contarBits(int num) {
    int count = 0;
    while (num) {
        count += num & 1;  // Sumar si el bit menos
           significativo es 1
        num >>= 1;         // Desplazar a la derecha
    }
    return count;
}

int main() {
    int num = 10;
    printf("El número %d tiene %d bits en 1\n", num,
       contarBits(num));
    return 0;
}
```

Ejemplo 9. El desplazamiento de bits es una alternativa eficiente para multiplicar o dividir por potencias de 2:

```c
#include <stdio.h>

int main() {
    int num = 6;
    printf("Multiplicar por 2: %d<<1 = %d\n", num, num
       <<1);
    printf("Dividir por 2: %d>>1 = %d\n", num, num>>1);
    return 0;
}
```

Sin embargo, hay que tener algunas precauciones:

- Asegurar que el resultado no supere el tamaño del tipo de dato.
- Verificar si el compilador usa desplazamiento aritmético o lógico para números negativos.
- Tener cuidado con el truncamiento en la división cuando el número no es par.
- No desplazar más bits de los que tiene el tipo de dato.

Apéndices

A. Operador ternario

El operador ternario (`? \,:`)[1] puede interpretarse como una forma compacta de un `if/else`. Opera con tres expresiones: $E1$, $E2$ y $E3$. Así, la expresión `E1?E2:E3` es equivalente al siguiente código:

```c
if (E1)
   E2
else
   E3
```

Ejemplo 10. Determinar si un número es par o impar.

```c
#include <stdio.h>

int main(){
    char a;
    printf ("Ingrese un número\n");
    scanf ("%hhd", &a);
    (a&1)?printf("Es impar\n"):printf("Es par\n");
    return 0;
}
```

[1] Si bien no es un operador bit a bit, se incluye en este apunte debido a su utilidad al trabajar con operadores de bits, como se muestra en el Ejemplo 10.

B. Campos de bits

Cuando el espacio de almacenamiento es escaso, puede ser necesario empaquetar varios objetos en una sola palabra de máquina. El método que se utiliza en C para operar con campos de bits, está basado en las estructuras (*structs*). La forma general de definición de un campo de bits es la siguiente:

```
typedef struct bits
{
    <tipo de dato> nombre1: <cantidad de bits>;
    <tipo de dato> nombre2: <cantidad de bits>;
    . . .
    <tipo de dato> nombre3: <cantidad de bits >;
}campo_bits;
```

Cada campo de bits puede declararse como `char`, `unsigned char`, `int`, `unsigned int`, etc. y su longitud puede variar entre 1 y el máximo número de bits disponible de acuerdo a la arquitectura del microprocesador en que se esté trabajando.

Ejemplo 11. Uso de campos de bits.

```
#include<stdio.h>

typedef struct s_bits{
    unsigned int mostrar: 1;
    unsigned int rojo: 8;
    unsigned int azul: 8;
    unsigned int verde: 8;
    unsigned int transparencia: 1;
    }campo_bits;

int main(){
campo_bits unColor;

    /* Se crea un color */
    unColor.rojo = 51;
    unColor.azul = 55;
    unColor.verde = 255;
    unColor.transparencia = 1;

    /* Se verifica si el color es transparente */
    if (unColor.transparencia == 1){
       printf("El color es transparente\n");
    }
    return 0;
}
```

Referencias

[KR17] Brian Kernighan and Dennis M Ritchie. *The C programming language*. Prentice Hall, 2017.

[War13] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.