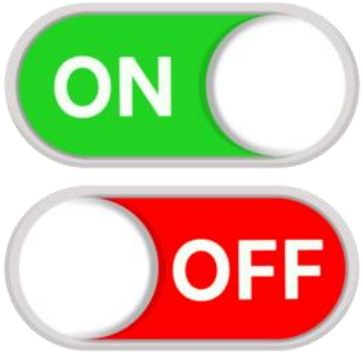
Manipulación de Bits

Bit (b)

- Unidad mínima de información.
- Es un dígito en el sistema binario, puede tomar dos valores: 0 (apagado) y 1 (prendido)
- La computadora solo “entiende” a nivel de bits.



Representación de enteros positivos

Como las computadores, en el más bajo nivel, solo entienden de bits, resulta natural que los números sean guardados en su representación binaria.

Los números positivos serán fácilmente convertidos a base 2. Por ejemplo asumiendo que tenemos 7 bits para usar, el número 13 sería guardado como 0001101.

En la realidad, C++ utiliza 16, 32 o 64 bits dependiendo del tipo de dato, pero para nuestros ejemplos vamos a utilizar una menor cantidad de bits para que sea más fácil visualizarlos.

Podemos enumerar las posiciones de cada bit. Por convención vamos a indexar las posiciones desde 0 y tomaremos al primer bit como al de más a la derecha.

El bit de más a la izquierda se le conoce como el **bit más significativo** (por ser el de mayor potencia) y el bit de más a la derecha como el **bit menos significativo**.

Bit	0	0	0	1	1	0	1
Posición	6	5	4	3	2	1	0

Representación de enteros negativos

- ¿Cómo haríamos para representar números negativos como una secuencia de bits?

Opción 1: Uso el bit más significativo para representar el signo.

Ejemplo: Asumiendo que tengo 4 bits disponibles.
Si tengo el número $3 \rightarrow 0011$, entonces el número $-3 \rightarrow 1011$
Si tengo el número $4 \rightarrow 0100$, entonces el número $-4 \rightarrow 1100$

Ventajas:

- Simple de representar para un humano

Desventajas:

- El 0 tiene dos formas de representarse: $0000 \rightarrow 1000$
- Hacer operaciones de suma y resta no es tan simple. Por ejemplo $3 + (-3) = 0$ pero si sumamos sus representaciones binarias $0011 + 1011 = 1110 \rightarrow -6$

Representación de enteros negativos

- ¿Cómo haríamos para representar números negativos como una secuencia de bits?

Opción 2: “Invierto” todos los bits. Esto es llamado **Complemento a 1**

Ejemplo: Asumiendo que tengo 4 bits disponibles.
Si tengo el número $3 \rightarrow 0011$, entonces el número $-3 \rightarrow 1100$
Si tengo el número $5 \rightarrow 0101$, entonces el número $-5 \rightarrow 1010$

Nota que el negativo de un número X con n bits es $2^n - X - 1$.

Ventajas:

- Hacer operaciones de suma y resta es un poco más simple. Ejemplo: $5 + (-3) = 2$. En complemento a 1, sería $0101 + 1100 = 10001$. Como solo tengo 4 bits, el 5to bit lo llevo a las unidades (+1) y tendría el número $0010 = 2$

Desventajas:

- El 0 tiene dos formas de representarse: $0000 \rightarrow 1111$

Representación de enteros negativos

- ¿Cómo haríamos para representar números negativos como una secuencia de bits?

Opción 3: “Invierto” todos los bits y le sumo 1. Esto es llamado **Complemento a 2**

Ejemplo: Asumiendo que tengo 4 bits disponibles.
Si tengo el número $3 \rightarrow 0011$, entonces el número $-3 \rightarrow 1101$
Si tengo el número $5 \rightarrow 0101$, entonces el número $-5 \rightarrow 1011$

Nota que el negativo de un número X con n bits es $2^n - X$.

Ventajas:

- Hacer operaciones de suma y resta es mucho más simple. Ejemplo: $5 + (-3) = 2$. En complemento a 2, sería $0101 + 1101 = 10010$. Como solo tengo 4 bits, el 5to bit ignoro y tendría el número $0010 = 2$
- El 0 solo tiene una representación
- Tenemos un espacio extra que lo usamos para un negativo más. Es decir podemos representar valores en el rango $[-2^{n-1}, 2^{n-1} - 1]$

Desventajas:

- No tan simple para un humano

Debido a sus ventajas, es la más usada en los computadores actuales.

Operadores Bitwise

- Realizan operaciones bit a bit.
- Son similares a los operadores booleanos
- Estos son AND (&), OR (|), XOR (^), negación (~), left shift (<<), right shift (>>)
- En C++ funciona en $O(1)$

Veamos los más simples para 1 solo bit

a	b	~ a (not)	a & b (and)	a | b (or)	a ^ b (xor)
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Nota: En notación matemática al xor también se le representa como $\oplus$

Operadores Bitwise

Veamos ejemplos con números con más de 1 bit. En este caso tienes que hacer la operación bit por bit, de manera independiente.

$A = 1010$
$B = 1001$

- $\sim A = 0101$
- $A \,\&\, B = 1000$
- $A \mid B = 1011$
- $A \oplus B = 0011$

Propiedades:

- $A \,\&\, A = A$
- $A \,\&\, 0 = 0$
- $A \mid A = A$
- $A \mid 0 = A$
- $A \oplus A = 0$
- $A \oplus 0 = A$

Otras propiedades del XOR

- **Commutativo:** $A \oplus B = B \oplus A$
- **Asociativo:** $(A \oplus B) \oplus C = A \oplus (B \oplus C)$
- Si $A \oplus B = C \implies A = B \oplus C$

Demostración:

$A \oplus B \oplus B = C \oplus B \implies A = B \oplus C$ ■

- $A + B = (A \oplus B) + 2\,(A \& B)$

Demostración:

Si analizamos bit por bit, el $A \oplus B$ tiene casi el mismo comportamiento que $A + B$ excepto en el caso que se operan los bits 1 y 1, para lo cual el resultado en ambos casos es un bit 0; pero en el caso de la suma, adicionalmente, se "lleva" un 1 a la siguiente potencia. Hasta ahí podemos concluir que $A + B = A \oplus B + carry$, donde $carry$ es toda la cantidad que se llevó.

Finalmente, podemos notar que $A \& B$ representa, con sus bits encendidos, las posiciones en donde se tuvo que "llevar" 1 a la siguiente potencia. Y $2(A \& B)$ mueve todos los bits de $A \& B$ a la izquierda, obteniendo exactamente lo que se llevó ($carry$), por lo tanto $carry = 2(A \& B)$. ■

Operadores Bitwise

- **Left shift ($\ll$)**

Realizar $x \ll i$ hace que todos los bits de x corran i posiciones a la izquierda, completando esos i espacios de la derecha con 0. Esta operación es equivalente a hacer $x \times 2^i$.

Nota: Los primeros i bits van a excederse del tamaño máximo al correr a la izquierda. Estos bits se ignorarían.
Advertencia: Si x es negativo, la operación tiene **undefined behavior.**

Ejemplos:
Asumamos que tenemos 6 bits

Si tenemos $x = 5 = 000101$
Entonces
$x \ll 1 = 001010 = 10$
$x \ll 2 = 010100 = 20$
$x \ll 3 = 101000 = 40$
$x \ll 4 = 010000 = 16$ (overflow)

Operadores Bitwise

- **Right shift (≫)**

Realizar $x \gg i$ hace que todos los bits de x corran i posiciones a la derecha, completando esos i espacios de la izquierda con el bit del signo (normalmente 0 ya que recomendamos trabajar solo con positivos). Esta operación es equivalente a hacer $\lfloor x \div 2^i \rfloor$.

Nota: Los primeros i bits van a excederse del tamaño máximo al correr a la izquierda. Estos bits se ignorarían.
Advertencia: Si x es negativo, la operación tiene **implementation defined** (depeden del compilador), aunque la mayoría de implementaciones preservará el signo de x.

Ejemplos:
Asumamos que tenemos 6 bits
Si tenemos $x = 13 = 001101$
Entonces
$x \gg 1 = 0001101 = 6$
$x \gg 2 = 000011\cancel{01} = 3$
$x \gg 3 = 000001\cancel{101} = 1$
$x \gg 4 = 000000\cancel{1101} = 0$

Operadores Bitwise

Hay que tener cuidado con los overflows.

En C++, el resultado de los operadores shift tendrán el mismo tipo que el primer operando.

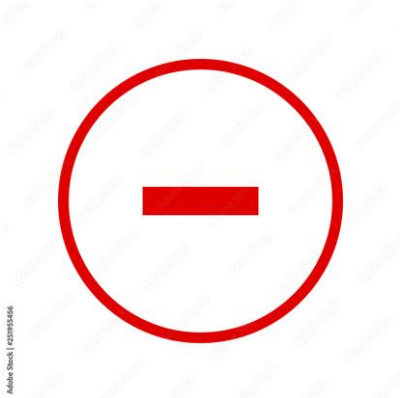
Ejemplo, en estas 3 sentencias, las 2 primeras tendrán overflow ya que el operando a es un int. Solo la 3ra será calculado en un long long por el cast explícito.

```
int a = 1;
int b = 40;
cout << (a << b) << endl;
cout << (a << Long(b)) << endl;
cout << (Long(a) << b) << endl;
```

Negativos con operadores bitwise

Sabemos que en C++ números son guardados en complemento a 2.

Por lo tanto, algo útil de saber es que hallar el negativo de un número es equivalente a negar sus bits y sumarle 1, es decir: $-x = \sim x + 1$



Máscara de bits (bitmask)

- Podemos usar un entero para representar un arreglo booleano o subconjuntos de un conjunto de hasta 32 elementos (o 64 si usamos long long).
- El i-ésimo bit del entero (máscara) es 1 si el i-ésimo elemento del conjunto está presente y será 0 si está ausente.

Supongamos que tenemos el conjunto {8, 10, 11}

Número	Bitmask	Subconjunto
0	000	{}
1	001	{8}
2	010	{10}
3	011	{8, 10}
4	100	{11}
5	101	{8, 11}
6	110	{10, 11}
7	111	{8, 10, 11}

Tareas frecuentes

- Unión de conjuntos : $A \mid B$
- Intersección de conjuntos : $A \,\&\, B$
- Diferencia de conjuntos : $A \,\&\, {\sim}B$
- Obtener bit de posición i : $(\text{mask} \gg \text{i}) \,\&\, 1$
- Encender el bit de posición i : $\text{mask} = \text{mask} \mid (1 \ll \text{i})$
- Apagar el bit de posición i : $\text{mask} = \text{mask} \,\&\, (\,{\sim}(1 \ll i)\,)$
- Cambiar estado del bit i : $\text{mask} = \text{mask} \,\hat{}\, (1 \ll \text{i})$

Tareas frecuentes — Funciones útiles

Tarea	Función (int)	Función (long long)	Comment
Número de bits encendidos de x	$\_\_builtin\_popcount(x)$	$\_\_builtin\_popcountll(x)$	
Número de bits 0 consecutivos a la izquierda de x (leading zeroes)	$\_\_builtin\_clz(x)$	$\_\_builtin\_clzll(x)$	Undefined behavior cuando x es 0
Número de bits 0 consecutivos a la derecha de x (trailing zeroes)	$\_\_builtin\_ctz(x)$	$\_\_builtin\_ctzll(x)$	Undefined behavior cuando x es 0

Tareas frecuentes

- **Obtener último bit encendido de un entero (en forma de potencia)**

Ejemplo: Para un $x = 14 = 1110$. El último bit encendido se ubica en la posición 1 y equivale a $2^1 = 2$
Ejemplo: Para un $x = 12 = 1100$. El último bit encendido se ubica en la posición 2 y equivale a $2^2 = 4$
Ejemplo: Para un $x = 0 = 0000$. El último bit no existe

Una primera forma de obtener esto es utilizar la función $\_\_builtin\_ctz(x)$ para obtener el número de 0 a la derecha. Esto nos dará la posición del último bit encendido. Luego podemos utilizar right shift para poder obtener la potencia.

El resultado en forma compacta sería $1 \ll \_\_builtin\_ctz(x)$

Tareas frecuentes

❖ **Obtener último bit encendido de un entero (en forma de potencia)**

Una segunda forma más compacta sería con la expresión $x \ \& \ -x$.

Demostración

Sea $x = \overline{a10 \ldots 0}$, donde a representa una secuencia de bits arbitraria.

$$\Longrightarrow -x = \overline{\sim(a10 \ldots 0)} + 1$$
$$\Longrightarrow -x = \overline{(\sim a)01 \ldots 1} + 1$$
$$\Longrightarrow -x = \overline{(\sim a)10 \ldots 0}$$

Finalmente,

$$x \ \& \ -x = \overline{a10 \ldots 0} \ \& \ \overline{(\sim a)10 \ldots 0} = \mathbf{0} \ldots \mathbf{010} \ldots \mathbf{0}$$

■

Tareas frecuentes

- **Saber si un número es potencia de 2**

Quitamos el último bit encendido y lo que queda deber ser igual a 0

$Si \;\; x - (x \;\&\; -x) = 0 \Longrightarrow es\ potencia\ de\ 2$
$Si \;\; \_\_builtin\_popcount(x) = 1 \;\; \Longrightarrow es\ potencia\ de\ 2$

Tareas frecuentes — Funciones útiles C++20

Con la introducción de C++20, llegaron algunas nuevas funciones útiles que pueden reemplazar a las $builtin$ vistas anteriormente e incluso también otras con nuevas funcionalidades. Sin embargo, algo a tomar en cuenta es que estas funciones generalmente solo reciben un entero de tipo T **unsigned** en sus parámetros, lo cual puede ser algo molesto.

- $int\ popcount(T\ x)$: Cuenta el número de bits prendidos. Reemplaza a $\_builtin\_popcount$
- $int\ countl\_zero(T\ x)$: Cuenta el número de bits 0 consecutivos a la izquierda. Reemplaza a $\_builtin\_clz(x)$
- $int\ countr\_zero(T\ x)$: Cuenta el número de bits 0 consecutivos a la derecha. Reemplaza a $\_builtin\_ctz(x)$
- $bool\ has\_single\_bit(T\ x)$: Retorna true si y solo si el número es potencia de 2
- $int\ countl\_one(T\ x)$: Cuenta el número de bits 1 consecutivos a la izquierda.
- $int\ countr\_one(T\ x)$: Cuenta el número de bits 1 consecutivos a la derecha.
- $T\ bit\_ceil(T\ x)$: Retorna la mínima potencia 2^p que sea mayor o igual que x
- $T\ bit\_floor(T\ x)$: Retorna la máxima potencia 2^p que sea menor o igual que x
- $int\ bit\_width(T\ x)$: Retorna el mínimo número de bits necesarios para representar x. Es decir $1 + \lfloor \log_2 x \rfloor$
- $T\ rotl(T\ x, int\ s)$: Realiza una rotación circular de los bits hacia la izquierda s posiciones ($1011 \rightarrow 0111$)
- $T\ rotr(T\ x, int\ s)$: Realiza una rotación circular de los bits hacia la derecha s posiciones ($0111 \rightarrow 1011$)

¡Gracias por su atención!

Referencias

- Wikipedia. Method of Complements. Recuperado de https://en.wikipedia.org/wiki/Method_of_complements