Capítulo 17 — El mundo binario: bits, bytes y máscaras

> *Capítulo avanzado. Se apoya en variables y tipos (`02`), operadores (`03`), `if` y bucles (`04`), funciones (`05`), punteros (`07`), `struct` (`08`) y tipos de ancho exacto (`16`).*
>
> *Acá bajás un piso: dejás de mirar **números** y empezás a mirar los **bits** que los componen. Con eso vas a poder meter ocho banderas en un solo byte, extraer campos de un archivo binario, hacer multiplicaciones y divisiones gratis, y entender de una vez qué son `BYTE` , `WORD` , `DWORD` y `QWORD` .*

Cómo leer este capítulo

Este es el capítulo que convierte "sé que existen los bits" en "los uso todos los días". No es teórico: cada idea viene con **código para mirar**, y cada bloque cierra con **práctica** para que lo manejes con las manos.

Está dividido en **seis partes**. Después de cada concepto importante hay uno o varios ejercicios cortos. **No los saltees**: son la mitad del capítulo.

Parte	De qué habla	La idea en una frase
1	El sustrato	Un bit es un interruptor; un byte son ocho interruptores en fila.
2	Los operadores `&` , `|` , `^` , `~`	Cuatro formas de combinar dos filas de interruptores.
3	Máscaras	Una plantilla que deja pasar solo los bits que te interesan.
4	`<<` y `>>`	Correr los interruptores y, de yapa, multiplicar y dividir.
5	`BYTE` , `WORD` , `DWORD` , `QWORD`	Cajas de 8, 16, 32 y 64 bits, y cómo leer campos dentro de ellas.
6	Conceptos extra	Contar bits, potencia de dos, rotar, endianness, `bitfields` y más.

Los **ejercicios** se numeran en orden: `17.1` , `17.2` , `17.3` , y así. Cada uno trae su **escenario de prueba** (qué entra y qué tiene que salir, nunca el algoritmo) para que **compiles y compruebes**.

> ***Regla de oro del libro:*** *el capítulo explica y muestra* ***cómo funciona*** *cada cosa (con código ilustrativo), pero* ***los ejercicios no traen la solución****. El enunciado te da el problema; la respuesta la escribís vos.*

PARTE 1 — El sustrato: bit, byte y bases

Antes de operar con bits, hay que verlos. Esta parte es corta y es la base de todo lo demás.

1.1 El bit: un interruptor

Pensá en un **interruptor de luz**. Tiene dos posiciones: **apagado** o **encendido**. No hay matices. Un **bit** es exactamente eso: un interruptor que puede valer **0** (apagado) o **1** (encendido).

Con **un** interruptor solo podés representar dos cosas. Con **dos**, cuatro combinaciones (`00` , `01` , `10` , `11`). Con **ocho**, doscientas cincuenta y seis (2^8). Esa regla es la única que tenés que recordar:

```
n bits  →  2ⁿ combinaciones distintas
```

1.2 El byte: ocho interruptores en una caja

Un **byte** es, en la práctica de hoy, **8 bits**. Los lenguajes modernos garantizan que `sizeof(char) == 1` , pero **no** que un carácter tenga 8 bits: eso lo dice otra constante, `CHAR_BIT` , que en todas las máquinas que vas a tocar vale **8**.

```c
#include <stdio.h>
#include <limits.h>   /* define CHAR_BIT */

int main(void) {
    /* sizeof(char) SIEMPRE es 1: un char ocupa "un byte".
       Pero "un byte" tiene CHAR_BIT bits. En x86/x64/ARM, CHAR_BIT == 8. */
    printf("sizeof(char) = %zu\n", sizeof(char));
    printf("CHAR_BIT     = %d\n", CHAR_BIT);   /* casi siempre: 8 */
    return 0;
}
```

Los bits de un byte se numeran **de derecha a izquierda**, empezando en **0**. Es una convención, no un capricho: el bit 0 es el **menos significativo** (el que vale `2⁰ = 1`) y el bit 7, el **más significativo** (el que vale `2⁷ = 128`).

```
bit:   7    6    5    4    3    2    1    0
     ┌────┬────┬────┬────┬────┬────┬────┬────┐
byte │ 0  │ 1  │ 0  │ 1  │ 1  │ 0  │ 0  │ 1  │
     └────┴────┴────┴────┴────┴────┴────┴────┘
valor: 128   64   32   16    8    4    2    1
             (los que están en 1 se suman)
     = 0+64+0+16+8+0+0+1 = 89
```

1.3 Bases: por qué escribimos `0x` y `0b`

Tu computadora piensa en **binario**, pero nadie escribe `0b01011001` a mano todo el tiempo: es larguísimo y un dígito de más te arruina todo. Por eso usamos **atajos**:

Base	Prefijo	Ejemplo	Cuánto vale
Decimal	(ninguno)	`89`	89
Hexadecimal	`0x`	`0x59`	89
Octal	`0`	`0131`	89
Binario	`0b`	`0b01011001`	89

El **hexadecimal** es el atajo estrella porque **cada dígito hex son exactamente 4 bits** (un *nibble*, del inglés *nibble*, "mordisco"). Entonces un byte entra justo en **dos dígitos hex**. Pasás de binario a hex **de a cuatro bits**, sin cuentas:

```
0101 1001        ← el byte partido en dos nibbles
  5    9         ← cada nibble, su dígito hex
= 0x59
```

> ***Momento ajá.*** *Los dígitos hex van del `0` al `9` y después de la `A` a la `F` (`A=10`, `B=11`, … `F=15`). ¿Por qué letras? Porque con un solo símbolo hay que representar **16** valores y solo tenemos **10** dígitos. Así que se usan seis letras. Nada más.*

> ***El prefijo binario `0b` es de C23.*** *Se estandarizó recién ahí (y con él, el separador `'` para agrupar dígitos: `0b0101'1001`). En compiladores viejos, `0b` era una* ***extensión*** *de GCC/Clang que quizá no estaba activada. Si te sale un error raro con `0b`, revisá el estándar: compilá con `-std=c23` (o usá `0x`).*

1.4 El byte también guarda negativos: complemento a dos

Un byte sin signo (`unsigned char`) va de `0` a `255`. Pero un byte **con signo** (`signed char`, y `char` en la mayoría de las plataformas) va de **`-128` a `127`**. ¿Cómo entra un número negativo en una caja de bits que no tiene signo?

C dijo "el bit **más alto** decide": si el bit 7 vale **1**, el número es **negativo**, y su valor se lee con la regla del **complemento a dos**: para saber cuánto vale, **invertís todos los bits y sumás 1**.

```
0b11111111  →  invertir: 00000000  →  +1  →  -1
0b11111110  →  invertir: 00000001  →  +1  →  -2
0b10000000  →  invertir: 01111111  →  +1  →  -128   (el más negativo)
```

Esa es la razón de que el rango sea asimétrico: el `0` ocupa una sola combinación (`00000000`), así que del lado negativo sobra un lugar: hay un `-128` pero no un `+128`.

> ***Ojo con esto.*** *El* ***mismo byte*** *`0xFF` es* ***255*** *si lo tratás como `unsigned char` y* ***-1*** *si lo tratás como `signed char`. Los bits no cambian: cambia* ***cómo los interpretás****. Por eso importa tanto el* ***tipo*** *(capítulo `16`): es el que le dice a C cómo leer esos bits.*

1.5 Extensión de signo: el detalle que muerde

Cuando un número "chico" se convierte en uno "grande", C tiene que rellenar los bits nuevos. Y ahí se comporta distinto según el signo:

- **Sin signo**: rellena con **0**. `(unsigned)0x9` → `0x00000009`.
- **Con signo**: **copia el bit de signo** (extensión de signo). `(int)-1` → `0xFFFFFFFF`.

```c
#include <stdio.h>
```

```c
int main(void) {
    signed char   sc = -1;      /* 0xFF en 8 bits */
    unsigned char uc = 255;     /* 0xFF en 8 bits */

    /* %x imprime en hexadecimal (sin signo). Mirá los ceros de relleno:
       sc (con signo) propaga el 1 del signo; uc (sin signo) rellena con 0. */
    printf("sc = %08x\n", (unsigned)sc);   /* ffffffff */
    printf("uc = %08x\n", (unsigned)uc);   /* 000000ff */

    return 0;
}
```

> ***Para guardar.*** *Antes de operar con bits, preguntate:* ***¿este valor es un número o un montón de bits?*** *Si es un montón de bits (una máscara, un byte de imagen), usá* ***tipos sin signo*** *(`uint8_t`, `uint32_t`). Evitás sorpresas como las de arriba. La regla corta:* ***bits se manipulan en `unsigned`.***

Práctica de la Parte 1

Ejercicio 17.1 — Radiografía de un byte

Conceptos · bits, valor posicional, `uint8_t`, `printf` (`%x`, `%b` o `%u`)

Contexto · Para entender los bits hay que verlos. Simplificación: mostrás un byte en decimal, hexadecimal y binario, y desglosás qué bits están encendidos.

Consigna · Pedí (o fijá) un valor de `0` a `255` en un `uint8_t`. Imprimí:

1. el valor en **decimal**;
2. el mismo valor en **hexadecimal** (`%x`), con dos dígitos;
3. el mismo valor en **binario**. Podés usar el `%b` de C23 o, si compilás con un estándar anterior, construir el texto bit a bit con un bucle;
4. la lista de **posiciones** de los bits que están en `1` (de mayor a menor o al revés, a tu gusto).

Entrada · Un valor de `0` a `255`. **Salida** · El valor en las tres bases y la lista de bits encendidos.

Escenario de prueba

```
valor = 89
decimal     : 89
hexadecimal : 0x59
binario     : 01011001
bits en 1   : 6, 4, 3, 0      (64 + 16 + 8 + 1 = 89)
```

```
valor = 255
decimal     : 255
hexadecimal : 0xff
binario     : 11111111
bits en 1   : 7, 6, 5, 4, 3, 2, 1, 0
```

Identificadores sugeridos · `value`, `mask`, `bitIndex`, `printBinary` **Criterios de aceptación**

- [ ] El valor se guarda en un `uint8_t` (sin signo).
- [ ] Las tres bases coinciden para los dos escenarios.
- [ ] La lista de bits encendidos suma exactamente el valor original.
- [ ] El binario se muestra con **ocho** dígitos (con ceros a la izquierda).

Ejercicio 17.2 — ¿Cuánto vale ese bit de signo?

Conceptos · complemento a dos, `int8_t` vs `uint8_t`, `0xFF`, interpretación según el tipo

Contexto · El mismo byte significa dos cosas distintas según lo leas con o sin signo, y eso confunde a cualquiera la primera vez. Simplificación: mostrás las dos interpretaciones de los mismos bits.

Consigna · Tomá el byte `0xFF` y guardalo en un `uint8_t` y en un `int8_t`. Imprimí **ambos** valores y el **mismo** byte en hex. Después explicá en un comentario **por qué** el mismo patrón de bits da dos números distintos.

Entrada · Ninguna (el byte es fijo, `0xFF`). **Salida** · Los dos valores y el hex.

Escenario de prueba

```
bits       : 11111111  (0xff)
uint8_t    : 255
int8_t     : -1
explicación: el tipo decide si el bit 7 es "128" (sin signo) o el signo (complemento a
dos, con signo).
```

Identificadores sugeridos · `rawByte`, `asUnsigned`, `asSigned` **Criterios de aceptación**

- [ ] El mismo patrón de bits se imprime igual en hex.
- [ ] Sin signo da `255`; con signo da `-1`.
- [ ] La explicación menciona el bit de signo y el complemento a dos.
- [ ] No se cambian los bits: solo la **interpretación**.

PARTE 2 — Los operadores de bits: `&`, `|`, `^` y `~`

Acá está el corazón del capítulo. Son **cuatro** operadores (más dos de desplazamiento, que vienen en la Parte 4), y se aplican **bit por bit**, columna por columna. Se parecen a los lógicos, pero **no** son los mismos: `&` no es `&&`, y `|` no es `||`.

> ***No los confundas.***
>
> - `&` *(un solo ampersand) = AND* ***de bits****, opera sobre cada bit.*
> - `&&` *(dos) = AND* ***lógico****, devuelve* `0` *o* `1` *según si ambos son "verdaderos".*
> - `|` *(una barra) = OR* ***de bits****.* `||` *(dos) = OR* ***lógico****. La diferencia es la misma que entre "multiplicá cada dígito" y "¿es el número distinto de cero?".*

2.1 `&` — AND: "solo si los dos están encendidos"

El AND bit a bit compara **dos bits** y devuelve `1` **solo si ambos son** `1`. Si alguno es `0`, el resultado es `0`. Pensalo como **dos amigos que tienen que estar de acuerdo**: si uno dice que no, no hay trato.

`a`	`b`	`a & b`
0	0	0
0	1	0
1	0	0
1	1	**1**

```c
#include <stdio.h>

int main(void) {
    uint8_t a = 0b11001100;   /* 0xCC */
    uint8_t b = 0b10101010;   /* 0xAA */

    /* & compara cada bit con su vecino de la otra fila.
       Solo queda en 1 donde AMBAS tienen 1. */
    uint8_t r = a & b;        /* 0b10001000 = 0x88 */

    printf("a     = %08b (0x%02x)\n", a, a);   /* 11001100 (0xcc) */
    printf("b     = %08b (0x%02x)\n", b, b);   /* 10101010 (0xaa) */
    printf("a & b = %08b (0x%02x)\n", r, r);   /* 10001000 (0x88) */
    return 0;
}
```

2.2 `|` — OR: "si alguno está encendido"

El OR bit a bit devuelve `1` si **al menos uno** de los dos bits es `1`. Solo da `0` cuando **los dos** son `0`. Es el operador de **encender**: sirve para "prender" bits concretos.

`a`	`b`	`a | b`
0	0	0
0	1	**1**
1	0	**1**
1	1	**1**

```
uint8_t a = 0b11001100;
uint8_t b = 0b10101010;
uint8_t r = a | b;   /* 0b11101110 = 0xEE */
```

2.3 `^` — XOR: "encendido solo si son DISTINTOS"

El XOR (OR exclusivo) devuelve `1` **solo si los dos bits son distintos**. Si son iguales, da `0`. Es un **detector de diferencias** y, además, el único operador que **invierte** un bit cuando le aplicás un `1` (por eso es el operador de *toggle*, ver Parte 3).

`a`	`b`	`a ^ b`
0	0	0
0	1	**1**
1	0	**1**
1	1	0

```
uint8_t a = 0b11001100;
uint8_t b = 0b10101010;
uint8_t r = a ^ b;   /* 0b01100110 = 0x66 */
```

> ***Dato de color.*** *El XOR tiene una propiedad mágica que lo hace famoso:* ***aplicarlo dos veces devuelve el original*** *(`x ^ k ^ k == x`). Por eso se usa en cifrados simples de juguete y en el truco de "intercambiar dos variables sin variable temporal". Es lindo, pero* ***`^` con la misma variable dos veces da 0****: no te olvides de eso.*

2.4 `~` — NOT: "invertí todo"

El complemento (`~`) es el único que trabaja con **un** operando: agarra cada bit y lo **da vuelta** (`0` → `1`, `1` → `0`). Es el operador de la **negación**, y es la pieza clave para **apagar** bits (Parte 3).

`a`	`~a`
0	1
1	0

```c
#include <stdio.h>

int main(void) {
    uint8_t a = 0b11001100;
    /* ~ invierte cada bit. Ojo: se evalúa promoviendo a int,
       por eso conviene asignar a un uint8_t para "recortar" a 8 bits. */
    uint8_t r = (uint8_t)~a;   /* 0b00110011 = 0x33 */

    printf("a  = %08b (0x%02x)\n", a, a);   /* 11001100 (0xcc) */
    printf("~a = %08b (0x%02x)\n", r, r);   /* 00110011 (0x33) */
    return 0;
}
```

> ***La identidad que no se olvida.*** *Con números enteros, `~x == -x - 1`. Por eso `~0` es `-1` (`0xFFFFFFFF`) y `~5` es `-6`. No es una regla aparte: es el complemento a dos otra vez.*

2.5 La tabla de un solo vistazo

Operador	Nombre	Hace…	Regla corta	Uso típico
`&`	AND	deja pasar si ambos son 1	**filtrar / testear**	"¿está encendido el bit?"
`|`	OR	enciende si alguno es 1	**prender**	"encendé estos bits"
`^`	XOR	enciende si son distintos	**invertir / comparar**	"cambiá este bit"
`~`	NOT	da vuelta todo	**negar**	"apagá este bit" (`& ~`)
`<<`	shift izq	corre a la izquierda	$\times 2^k$	"movete k lugares"
`>>`	shift der	corre a la derecha	$\div 2^k$	"movete k lugares"

Práctica de la Parte 2

Ejercicio 17.3 — La calculadora de bits

Conceptos · `&` , `|` , `^` , operaciones bit a bit, entrada de dos bytes

Contexto · Antes de usar máscaras hay que ver los operadores "en crudo". Simplificación: una calculadora que combina dos bytes con cada operador.

Consigna · Leé **dos** valores de `0` a `255` (por ejemplo con `strtoul` en base 0, para aceptar `0x` y `0b` , o con `scanf`). Para ese par, mostrá el resultado de `&` , `|` , `^` y `~` **de cada uno**, en decimal, hex y binario. Mostrá también **la tabla de verdad** (los cuatro casos) para cada operador, como comprobación.

Entrada · Dos bytes (adminsitrá base `0x` , `0b` y decimal). **Salida** · Una fila por operación con los tres formatos.

Escenario de prueba

```
a = 0xCC   b = 0xAA
a & b = 0x88   (10001000)
a | b = 0xEE   (11101110)
a ^ b = 0x66   (01100110)
~a    = 0x33   (00110011)
~b    = 0x55   (01010101)
```

Identificadores sugeridos · `left`, `right`, `andResult`, `orResult`, `xorResult`
Criterios de aceptación

- [ ] Se manejan valores **sin signo**.
- [ ] Cada resultado se imprime en decimal, hex y binario.
- [ ] La tabla de verdad de cada operador se muestra y coincide con la teoría.
- [ ] Los resultados del escenario salen exactamente como se espera.

Ejercicio 17.4 — El mapa de diferencias

Conceptos · `^` como detector de diferencias, contar bits distintos

Contexto · Comparar dos versiones de un mismo byte (por ejemplo, dos fotos de un estado) y saber **qué bits cambiaron** es un caso real de XOR. Simplificación: contás y ubicás las diferencias entre dos bytes.

Consigna · Leé dos bytes. Usando `^`, mostrá:

1. una **máscara** con `1` en cada posición donde los dos bytes **difieren**;
2. **cuántos** bits difieren (la cantidad de `1` de esa máscara);
3. la lista de las **posiciones** que difieren.

Entrada · Dos bytes. **Salida** · La máscara, el conteo y las posiciones.

Escenario de prueba

```
a = 0b11001100
b = 0b11000011
a ^ b = 0b00001111  (0x0f)
bits distintos    : 4
posiciones        : 0, 1, 2, 3
```

Identificadores sugeridos · `a`, `b`, `diffMask`, `diffCount` **Criterios de aceptación**

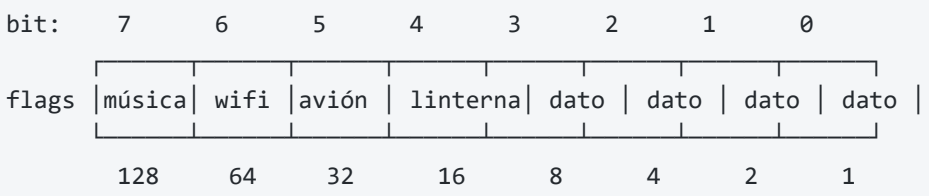
- [ ] La máscara de diferencias sale de **un** `^` entre los dos bytes.
- [ ] El conteo coincide con el número de posiciones listadas.
- [ ] Dos bytes iguales dan máscara `0` y conteo `0`.
- [ ] Se usa un tipo sin signo para los bytes.

PARTE 3 — Máscaras: un byte, ocho banderas

Esta es **la** parte que pediste, y es la aplicación más común de todo lo anterior. Una **máscara** es un valor con **un `1` en el/los bits que te interesan** y `0` en el resto. Es como un **calco** (*stencil*, la plantilla con agujeros): la ponés encima y solo pasa la luz por los agujeros.

3.1 La idea: ocho fuentes de luz, un solo interruptor general

Imaginate una sala con **ocho lámparas** y **un tablero con ocho interruptores**. No querés ocho cajas separadas: querés **una** caja de 8 bits donde cada bit sea el estado de una lámpara. Eso es exactamente un byte usado como **conjunto de banderas** (*flags*).



Con **un byte**, ocho banderas booleanas. Con un `uint32_t` (**DWORD**), **32**. Con un `uint64_t` (**QWORD**), **64**. La técnica es siempre la misma; cambia el ancho de la caja.

3.2 El truco del `1 << n`: fabricar una máscara

Para una bandera en la posición `n`, la máscara es un `1` corrido `n` lugares a la izquierda. Eso se escribe `1u << n` (el `u` la hace `unsigned`, para no meter signo en la fiesta).

```
n = 0 →  1u << 0 = 0b00000001   (0x01)
n = 3 →  1u << 3 = 0b00001000   (0x08)
n = 7 →  1u << 7 = 0b10000000   (0x80)
```

Y como cada bandera tiene su nombre, lo natural es **definir máscaras con nombre** en vez de repetir el `1u << n` por todo el código:

```
#include <stdint.h>

/* Cada bandera es un bit propio. Los nombres dicen QUÉ son, no cuánto valen. */
enum DeviceFlags {
    FLAG_ON            = 1u << 0,   /* 0x01  bit 0 */
    FLAG_CONNECTED     = 1u << 1,   /* 0x02  bit 1 */
    FLAG_ERROR         = 1u << 2,   /* 0x04  bit 2 */
    FLAG_UPDATE_READY  = 1u << 3,   /* 0x08  bit 3 */
    FLAG_LOW_BATTERY   = 1u << 4,   /* 0x10  bit 4 */
    FLAG_MUTED         = 1u << 5,   /* 0x20  bit 5 */
    FLAG_CHARGING      = 1u << 6,   /* 0x40  bit 6 */
    FLAG_SAFE_MODE     = 1u << 7    /* 0x80  bit 7 */
};
```

> ***Por qué esto es mejor que ocho `bool`.*** *Ocho `bool` ocupan **8 bytes** (cada uno, aunque valga un bit, ocupa un byte entero). Las ocho banderas en **un `uint8_t` ocupan 1 byte**. Cuando eso se repite en un millón de registros, la diferencia deja de ser un detalle. Es el mismo motivo del capítulo `16` (sección 16.9): tipos chicos ahorran **cuando hay muchos**.*

3.3 Las cuatro operaciones sobre una bandera

Con las cuatro herramientas de la Parte 2, hacés todo lo que necesitás con una bandera: **testear**, **prender**, **apagar** e **invertir**. Esta es la tabla que vas a usar el resto de tu vida.

Quiero…	Escribo…	Operador	Por qué
Testear si está encendida	`flags & MASK`	`&`	solo queda `MASK` si el bit estaba en 1
Prender (poner en 1)	`flags | MASK`	`|`	fuerza ese bit a 1, deja el resto igual
Apagar (poner en 0)	`flags & ~MASK`	`& ~`	el `~MASK` es "todos menos ese bit"
Invertir (toggle)	`flags ^ MASK`	`^`	cambia 1↔0 solo en ese bit

Mirá cómo queda cada una en código:

```c
#include <stdint.h>
#include <stdio.h>

enum { FLAG_ERROR = 1u << 2 };   /* 0x04 */

int main(void) {
    uint8_t flags = 0;

    /* PRENDER: OR con la máscara fuerza el bit 2 a 1. */
    flags |= FLAG_ERROR;                  /* flags = 0b00000100 */
    printf("prendida : %08b\n", flags);

    /* TESTEAR: AND con la máscara. Si el resultado NO es 0, el bit estaba encendido. */
    if (flags & FLAG_ERROR) {             /* el & da 0x04, distinto de 0 → verdadero */
        printf("la bandera esta encendida\n");
    }

    /* INVERTIR: XOR con la máscara da vuelta solo ese bit. */
    flags ^= FLAG_ERROR;                  /* flags = 0b00000000 */
    printf("invertida: %08b\n", flags);

    /* VOLVER A PRENDER y luego APAGAR: AND con el complemento del bit. */
    flags |= FLAG_ERROR;                  /* 0b00000100 */
    flags &= ~FLAG_ERROR;                 /* ~0x04 = 0xFB → apaga el bit 2 */
    printf("apagada  : %08b\n", flags);

    return 0;
}
```

La trampa N.º 1: testear con `==`. *No escribas `if (flags == FLAG_ERROR)`: eso pregunta si* ***todas*** *las banderas valen exactamente eso, no si* ***una*** *está encendida. Se testea con `&`, y la condición es "el resultado es distinto de cero". La forma idiomática es `if (flags & FLAG_ERROR)`, no `if ((flags & FLAG_ERROR) == FLAG_ERROR)` (aunque también funciona).*

La trampa N.º 2: el `~` y el ancho. *`~FLAG_ERROR` sobre un `int` de 32 bits da `0xFFFFFFFB`, no `0xFB`. Al hacer `flags &= ~FLAG_ERROR` con `flags` de 8 bits, los bits de arriba del `~` se recortan al asignar, así que* ***funciona igual****. Pero si operás con tipos de distinto ancho en un mismo cálculo, tenelo presente.*

3.4 Varias banderas de una: preguntar por un grupo

Un `&` te deja preguntar por **varias** banderas juntas. Combinando máscaras con `|` armás "grupos":

```c
/* ¿Están las DOS banderas encendidas a la vez? (AND con la suma de máscaras == la suma)
*/
if ((flags & (FLAG_ERROR | FLAG_LOW_BATTERY)) == (FLAG_ERROR | FLAG_LOW_BATTERY)) {
    /* ambas encendidas */
}

/* ¿Está encendida ALGUNA de las dos? (el AND no es 0) */
if (flags & (FLAG_ERROR | FLAG_LOW_BATTERY)) {
    /* al menos una encendida */
}
```

> ***Para guardar.*** *`flags & GRUPO` = "¿alguna de estas?" (intersección no vacía). `(flags & GRUPO) == GRUPO` = "¿todas estas?" (todas presentes). Son dos preguntas **distintas**, y confundirlas es de los bugs más comunes.*

Práctica de la Parte 3

Ejercicio 17.5 — El panel de control

Conceptos · máscaras, `1u << n`, `&`, `|`, `~`, `^`, banderas

Contexto · Todo panel de control (un router, un reproductor, una consola de juegos) mantiene su estado en un conjunto de banderas. Simplificación: un panel de 8 banderas en un `uint8_t`.

Consigna · Definí **ocho** banderas con nombre, una por bit, para un panel de control (por ejemplo: `POWER`, `RECORDING`, `MUTED`, `NETWORK`, `ERROR`, `PAUSED`, `OVERHEATED`, `BATTERY_LOW`). Implementá **cinco** funciones que reciban el estado (`uint8_t`) y:

1. **prender** una bandera dada;
2. **apagar** una bandera dada;
3. **invertir** una bandera dada;
4. **testear** una bandera dada (devolver un `bool`);
5. **contar** cuántas banderas están encendidas.

Demostralas encendiendo unas cuantas, invirtiendo una, apagando otra y mostrando el estado en binario después de cada paso.

Entrada · Una secuencia de operaciones (o fija, en el `main`). **Salida** · El estado del panel en binario después de cada operación, y el conteo final.

Escenario de prueba

```
inicio            : 00000000  (0 encendidas)
prender RECORDING : 00000100  (1 encendida)
prender NETWORK   : 00010100  (2 encendidas)
invertir MUTED    : 00110100  (3 encendidas)
apagar RECORDING  : 00110000  (2 encendidas)
total encendidas  : 2
```

Identificadores sugeridos · `PanelState`, `setFlag`, `clearFlag`, `toggleFlag`, `isFlagSet`, `countFlags` **Criterios de aceptación**

- [ ] Las ocho banderas se definen con `1u << n` y con nombres significativos.
- [ ] Prender usa `|`; apagar usa `& ~`; invertir usa `^`; testear usa `&`.
- [ ] El conteo recorre los bits (no cuenta a mano).
- [ ] El escenario sale igual, y dos banderas nunca comparten el mismo bit.

Ejercicio 17.6 — Permisos al estilo Unix

Conceptos · máscaras, grupos de banderas, `&` con grupo, `==` con grupo

Contexto · Los permisos de archivo de Unix (`rwx` para dueño, grupo y otros) se guardan en bits justamente para operarlos con máscaras. Simplificación: modelás **solo** los permisos de **un** grupo sobre un `uint8_t` (aunque los reales son 9 bits en total).

Consigna · Definí tres banderas: `READ` (leer), `WRITE` (escribir), `EXEC` (ejecutar). Implementá funciones que digan:

1. si se puede **leer** (`READ`);
2. si se puede **escribir** (`WRITE`);
3. si se puede **ejecutar** (`EXEC`);
4. si **todos** los permisos están dados (testear con grupo y `==`);
5. si **alguno** de leer/escribir está dado (testear con grupo).

Mostrá el resultado para varios conjuntos de permisos.

Entrada · Varios valores de permisos (por ejemplo `0b111` , `0b100` , `0b010` , `0b000`).
Salida · El veredicto de cada función para cada conjunto.

Escenario de prueba

```
permisos = rwx (111): leer=sí   escribir=sí   ejecutar=sí   todos=sí   leer-o-escribir=sí
permisos = r-- (100): leer=sí   escribir=no   ejecutar=no   todos=no   leer-o-escribir=sí
permisos = -w- (010): leer=no   escribir=sí   ejecutar=no   todos=no   leer-o-escribir=sí
permisos = --- (000): leer=no   escribir=no   ejecutar=no   todos=no   leer-o-escribir=no
```

Identificadores sugeridos · `Permissions` , `canRead` , `canWrite` , `canExec` , `canAll` , `canReadOrWrite` **Criterios de aceptación**

- [ ] "Todos" usa `(p & GRUPO) == GRUPO` ; "alguno" usa `(p & GRUPO) != 0` .
- [ ] Cada permiso es un bit distinto.
- [ ] Los cuatro escenarios dan exactamente lo esperado.
- [ ] El código distingue claramente "todos" de "alguno".

Ejercicio 17.7 — El registro de opciones de arranque

Conceptos · banderas de 32 bits (`uint32_t`), máscaras altas, `1u << n` con `n` grande

Contexto · Los programas serios reciben muchas opciones (verbose, debug, dry-run, sin color, …) y las guardan como banderas. Con 8 bits te quedás corto enseguida, así que se usa un `uint32_t` . Simplificación: **treinta y dos** opciones, aunque solo uses algunas.

Consigna · Definí al menos **diez** banderas en un `uint32_t` , incluyendo algunas en **bits altos** (por ejemplo el bit 31). Implementá una función que reciba el registro y **liste por nombre** cada opción que esté encendida. Comprobá que una bandera en el bit 31 **no** se pierde ni se confunde con las de abajo.

Entrada · Un registro con varias opciones (incluida una del bit alto). **Salida** · La lista de opciones activas, por nombre.

Escenario de prueba

```
registro = 0x80000005   (bit 31 + bit 2 + bit 0)
opciones activas:
  - VERBOSE       (bit 0)
  - DRY_RUN       (bit 2)
  - EXPERIMENTAL  (bit 31)
```

Identificadores sugeridos · `StartupOptions`, `optionMask`, `listActiveOptions` **Criterios de aceptación**

- [ ] El registro es `uint32_t` (sin signo, para que el bit 31 no sea "signo").
- [ ] La bandera del bit 31 se define como `1u << 31` (o `0x80000000u`).
- [ ] El bit 31 se detecta correctamente (no aparece como negativo).
- [ ] Las banderas altas y bajas conviven sin pisarse.

PARTE 4 — Desplazamientos: `<<` y `>>`

Los otros dos operadores. Mueven **toda la fila de bits** a la izquierda o a la derecha. Es como correr un tren: los vagones que salen por un extremo se pierden, y por el otro extremo entran vagones nuevos (que son **ceros**).

4.1 `<<` y `>>`: correr los bits

- `x << k`: cada bit se corre **k lugares a la izquierda**. Los bits que se salen por la izquierda se **pierden**, y por la derecha entran **ceros**.
- `x >> k`: cada bit se corre **k lugares a la derecha**. Los bits que se salen por la derecha se **pierden**, y por la izquierda entran... depende del signo (ver 4.4).

```
0b00001111  << 2  =  0b00111100     (se corrió a la izquierda; entran dos ceros)
└ izquierda              derecha ┘
```

```
   0b11000000  >> 3  =  0b00011000     (se corrió a la derecha; entran tres ceros)
```

```c
#include <stdio.h>

int main(void) {
    uint8_t x = 0b00001111;   /* 0x0F */

    printf("%08b << 1 = %08b\n", x, (uint8_t)(x << 1));   /* 00011110 */
    printf("%08b << 4 = %08b\n", x, (uint8_t)(x << 4));   /* 11110000 */
    printf("%08b >> 2 = %08b\n", x, (uint8_t)(x >> 2));   /* 00000011 */
    return 0;
}
```

El `<<` también es la fábrica de máscaras. *Todas las máscaras de la Parte 3 (`1u << n`) son un desplazamiento. Por eso el `<<` aparece en todos lados: es tanto "mover" como "fabricar".*

4.2 Multiplicar por 2, 4, 8… con `<<`

Correr **un** bit a la izquierda **duplica el valor**. Correr **k** lugares lo multiplica por 2^k. No es magia: es que cada bit vale el **doble** que el de su derecha.

```
   0b00000101 = 5
   5 << 1 = 0b00001010 = 10   (×2)
   5 << 2 = 0b00010100 = 20   (×4)
   5 << 3 = 0b00101000 = 40   (×8)
```

4.3 Dividir (hacia abajo) con `>>`

Igual pero al revés: `x >> k` **divide** entre 2^k, **truncando** hacia el entero de abajo (para valores sin signo).

```
0b00101000 = 40
40 >> 1 = 0b00010100 = 20   (÷2)
40 >> 2 = 0b00001010 = 10   (÷4)
40 >> 3 = 0b00000101 = 5    (÷8)
```

4.4 Las cuatro trampas del desplazamiento

Acá es donde el `<<` y el `>>` dejan de ser "gratis". Leé esta lista dos veces.

Trampa 1 — corrimiento por el ancho o más es indefinido. Correr `k >= ancho` (por ejemplo `x << 32` sobre un `uint32_t`) es **comportamiento indefinido**: el estándar no dice qué pasa y cada CPU puede dar algo distinto. Corré **siempre** con `0 <= k < ancho`.

Trampa 2 — `>>` de un valor con signo es dependiente de la implementación. Para valores **sin signo**, `>>` rellena con ceros (es "lógico"). Para valores **con signo negativos**, el estándar deja que el compilador elija: la mayoría copia el bit de signo (corrimiento **aritmético**, `-8 >> 1` da `-4`), pero **no está garantizado**. Regla: desplazá a la derecha **valores sin signo**.

Trampa 3 — multiplicar puede desbordar silenciosamente. `5 << 30` sobre un `uint32_t` no da 5×2^{30}: los bits que se salen se pierden. El desplazamiento **no avisa** del desborde.

Trampa 4 — `<<` de un valor con signo que desborda es indefinido. Mover a la izquierda un `signed` de forma que el resultado no entre en el tipo es UB. Por eso, otra vez: **bits se manipulan en `unsigned`**.

> ***¿Entonces el truco de ×2 es inútil?*** *No. Es* ***rapidísimo*** *y lo usan a fondo los compiladores. Vos, además, lo vas a necesitar para leer y escribir* ***campos*** *en formatos binarios (Parte 5), donde el desplazamiento no es una optimización sino la forma correcta de pararse sobre los bits.*

Práctica de la Parte 4

Ejercicio 17.8 — El multiplicador sin `*`

Conceptos · `<<`, `>>`, multiplicación y división por potencias de dos

Contexto · Los procesadores antiguos (y algunos chips chicos de hoy) no tenían multiplicador de hardware: multiplicaban a fuerza de sumar y correr bits. Simplificación: multiplicás y dividís **solo** con desplazamientos.

Consigna · Dado un `uint32_t x` y un exponente `k` (de `0` a `30`), calculá `x * 2^k` y `x / 2^k` **usando únicamente** `<<` y `>>` (nada de `*`, `/` ni `pow`). Mostrá el resultado en decimal y binario. Comprobá con `k = 0` (no cambia nada) y con un `k` que haga perder bits.

Entrada · Un valor `x` y un exponente `k`. **Salida** · `x * 2^k` y `x / 2^k`, en decimal y binario.

Escenario de prueba

```
x = 5, k = 3
x * 2^3 = 40    (0b00000000000000000000000000101000)
x / 2^3 = 0     (5 >> 3, trunca a 0)

x = 40, k = 2
x * 2^2 = 160
x / 2^2 = 10
```

> ***Ojo:*** *`5 / 8` "de verdad" es `0.625`, pero con enteros y `>>` da* ***0****. La división con `>>` **trunca**.*

Identificadores sugeridos · `value`, `shift`, `multiplyByShift`, `divideByShift` **Criterios de aceptación**

- [ ] No se usa `*`, `/` ni `pow` en ninguna parte.
- [ ] `k = 0` devuelve el mismo valor.
- [ ] La división trunca hacia abajo (como `/` entre enteros).
- [ ] Se usa `uint32_t` (sin signo) para que `>>` sea confiable.

Ejercicio 17.9 — Empaquetar dos valores en uno

Conceptos · `<<` , `|` , `&` , máscaras, empaquetado

Contexto · A veces dos datos chicos tienen que viajar juntos dentro de uno más grande (por ejemplo, dos coordenadas de 16 bits en un solo `uint32_t`). Simplificación: empaquetás y desempaquetás dos valores.

Consigna · Tomá dos `uint16_t` (`high` y `low`) y **empaquetálos** en **un** `uint32_t` con el `high` en los 16 bits de arriba y el `low` en los 16 de abajo, usando `<<` y `|` . Después **desempaquetálos** de vuelta con `>>` y `&` , y comprobá que recuperás los valores originales.

Entrada · Dos valores de 16 bits. **Salida** · El paquete de 32 bits (en hex y binario) y los dos valores recuperados.

Escenario de prueba

```
high = 0xABCD   low = 0x1234
paquete   = 0xABCD1234
recuperado high = 0xABCD
recuperado low  = 0x1234
```

Identificadores sugeridos · `pack` , `unpackHigh` , `unpackLow` , `packed` **Criterios de aceptación**

- [ ] El paquete se arma con `<<` y `|` (sin sumas ni multiplicaciones).
- [ ] `high` se extrae con `>>` ; `low` , con una máscara de 16 bits.
- [ ] Los valores recuperados coinciden **exactamente** con los originales.
- [ ] Se comprueba que `high` y `low` no se pisaron.

PARTE 5 — `BYTE` , `WORD` , `DWORD` , `QWORD` y los paquetes de bits

Ya viste que una caja puede tener 8, 32 o 64 bits. En el mundo de Windows (y en el de los archivos binarios, como el BMP del capítulo `15`) esas cajas tienen **nombres clásicos**. Conocerlos te permite leer APIs y formatos de archivo sin traducir mentalmente todo el tiempo.

5.1 Los nombres de los anchos

Nombre	Bits	Bytes	Equivalente portable (`<stdint.h>`)	De dónde viene
BYTE	8	1	`uint8_t`	`<windows.h>`
WORD	16	2	`uint16_t`	`<windows.h>`
DWORD	32	4	`uint32_t`	`<windows.h>` (*double word*)
QWORD	64	8	`uint64_t`	convención (*quad word*)

> ***¿Qué significa "word"?*** *Originalmente, la* ***palabra*** *(word) era el tamaño con el que la CPU trabajaba de una: en las CPUs de 16 bits, una word eran 16 bits. De ahí el nombre. Con el tiempo las CPUs crecieron, y aparecieron double word (32) y quad word (64). Hoy "word" es un nombre* ***histórico****: por eso* `DWORD` *son 32 bits aunque tu máquina sea de 64.*

> ***La versión portable existe y es mejor.*** *Si no estás en Windows, o querés código que compile en cualquier lado, usá* `uint8_t` / `uint16_t` / `uint32_t` / `uint64_t` *del capítulo* `16`*.* `DWORD` *es un* `typedef` *de 32 bits* ***atado a*** `<windows.h>`*. El* ***ancho*** *es lo importante; el nombre es cosmético.*

5.2 Un `DWORD` no es "un número": es una caja de 32 bits

La clave de esta parte es cambiar el chip. Un `DWORD` (o un `uint32_t`) puede ser **dos cosas**:

- **un número** (por ejemplo, una suma, un contador), o
- **una caja de 32 bits** donde cada tramo guarda un dato distinto.

Cuando lo usás como caja, el **truco de máscaras y desplazamientos** es cómo entrás y salís de cada tramo. Es lo mismo que hiciste en la Parte 4 con el paquete de 16+16, pero con más tabiques.

5.3 Extraer campos: el patrón `(x >> shift) & mask`

Para leer un campo de `w` bits que empieza en la posición `s`, el patrón es siempre el mismo:

```
campos = (packed >> s) & ((1u << w) - 1)
```

- `>> s` **trae** el campo hasta la posición 0.
- `(1u << w) - 1` es una **máscara de `w` unos** (por ejemplo, `w = 5` → `0b11111`), que **borra** todo lo que no sea el campo.

Ejemplo con los bytes de un color `0x00RRGGBB` (como en un BMP de 24 bits, capítulo `15`):

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint32_t color = 0x00336699;   /* 00 RR GG BB */

    /* Cada canal son 8 bits. Extraigo con >> y recorto con & 0xFF. */
    uint8_t r = (color >> 16) & 0xFF;   /* 0x33 */
    uint8_t g = (color >>  8) & 0xFF;   /* 0x66 */
    uint8_t b = (color      ) & 0xFF;   /* 0x99 */

    printf("R=%02x G=%02x B=%02x\n", r, g, b);   /* R=33 G=66 B=99 */

    /* Y al revés: armar el color desde los canales (el | combina, el << posiciona). */
    uint32_t rebuilt = ((uint32_t)r << 16) | ((uint32_t)g << 8) | b;
    printf("rebuilt = 0x%06x\n", rebuilt);       /* 0x336699 */
    return 0;
}
```

> ***Pensalo como un edificio.*** *El `>>` es el **ascensor** que baja el campo hasta la planta baja; la máscara `& 0xFF` es el **portero** que deja pasar solo a los ocho bits de ese campo y echa al resto. Ese par —**bajar y filtrar**— es el 90% de leer formatos binarios.*

5.4 Empaquetar campos: el patrón `| (v << s)`

Armar el valor es el espejo: se **sube** cada campo con `<<` a su posición y se **combina** todo con `|`. Es exactamente lo que ya viste en 5.3, y lo que hace la función que arma el color.

> ***Un detalle que salva horas.*** *Cuando un campo tiene **1 bit**, es una bandera (Parte 3). Cuando tiene **varios bits**, es un **número** dentro de la caja. Antes de escribir una máscara, preguntate cuántos bits tiene el campo: un campo de 3 bits (por ejemplo, un nivel de `0` a `7`) necesita máscara `0b111`, no `0b1`.*

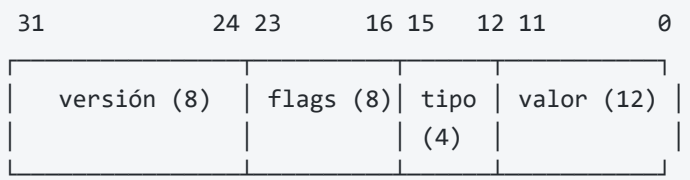
Práctica de la Parte 5

Ejercicio 17.10 — El decodificador de registros

Conceptos · `uint32_t` (`DWORD`), extracción de campos, `>>` , `&` , máscara de `w` bits, `|`

Contexto · Los protocolos de red y los archivos binarios empaquetan varios campos en un solo entero (a veces de tamaño fijo, como un `DWORD`). Simplificación: un registro de 32 bits con cuatro campos de distinto ancho que tenés que leer.

Consigna · Definí un registro de **32 bits** con esta distribución:



Implementá:

1. una función que **arme** el registro a partir de los cuatro valores (con `<<` y `|`);
2. cuatro funciones que **extraigan** cada campo (con `>>` y máscara de `w` bits);
3. una función que, dado el registro, imprima los cuatro campos.

Comprobá que armar y extraer son inversos: el valor que metiste es el que sale.

Entrada · Cuatro valores (versión `0` – `255` , flags `0` – `255` , tipo `0` – `15` , valor `0` – `4095`). **Salida** · El registro en hex y los cuatro campos extraídos.

Escenario de prueba

```
version = 2, flags = 0x84, type = 5, value = 2730
registro   = 0x02845AAA   (hex)
version    = 2
flags      = 0x84
type       = 5
value      = 2730
```

> ***Cómo comprobar el hex:*** `version << 24` = `0x02000000`, `flags << 16` = `0x00840000`, `type << 12` = `0x00005000`, `value` = `0x00000AAA`. *Sumados con* `|`: `0x02845AAA`.

Identificadores sugeridos · `packRecord`, `getVersion`, `getFlags`, `getType`, `getValue` **Criterios de aceptación**

- [ ] Ningún campo se solapa con otro (cada uno tiene sus bits).
- [ ] Las máscaras se construyen como `(1u << w) - 1` (por ejemplo, `type` con 4 bits → `0xF`).
- [ ] El registro se maneja como `uint32_t` (el bit 31 no es signo).
- [ ] Armar y extraer devuelven exactamente los valores de entrada.

PARTE 6 — Conceptos extra, de fácil a difícil

Todo lo de abajo es de la misma familia: operar bits. Están **ordenados por dificultad**, para que vayas sumando herramientas.

6.1 (Fácil) Limpiar el bit más bajo: `x & (x - 1)`

`x & (x - 1)` **apaga el bit** `1` **más a la derecha** que tenga `x`. Es la forma canónica de recorrer solo los bits encendidos: repetís esta operación y cada vuelta consume un bit.

```
x         = 0b10110000
x - 1     = 0b10101111
x & (x-1) = 0b10100000   ← se apagó el bit 4 (el más bajo encendido)
```

6.2 (Fácil) ¿Es potencia de dos? `x && !(x & (x - 1))`

Un número es potencia de dos si tiene **exactamente un bit** en `1`. Combinando 6.1: `x && !(x & (x - 1))` es verdadero solo para `1, 2, 4, 8, …`. El `x &&` inicial excluye el `0`, que técnicamente apagaría un bit que no existe.

6.3 (Media) Aislar el bit más bajo: `x & -x`

`x & -x` **deja solo** el bit `1` más a la derecha y apaga el resto. Funciona gracias al complemento a dos (Parte 1). Es la base de estructuras de datos como el *Fenwick tree*.

```
x      = 0b10110100
-x     = 0b01001100   (complemento a dos)
x & -x = 0b00000100   ← quedó solo el bit 2
```

6.4 (Media) Contar bits encendidos: *popcount*

Popcount es cuántos bits valen `1`. El camino a pie usa un bucle con `x &= x - 1` (6.1), que da una vuelta por cada bit encendido. Pero desde **C23** hay soporte **estándar** en `<stdbit.h>`:

```c
#include <stdbit.h>   /* C23 */

unsigned n = 0b10110100u;
unsigned ones = stdc_count_ones(n);   /* 4 */
```

> ***Antes de C23*** *se usaban extensiones del compilador:* `__builtin_popcount(x)` *(GCC/Clang). No son estándar, pero están por todos lados.*

6.5 (Media) Intercambiar sin variable temporal (XOR *swap*)

El XOR tiene la propiedad de que `a ^ b ^ b == a`. Encadenando tres XOR se intercambian dos variables **sin** una tercera:

```
a = a ^ b;
b = a ^ b;   /* (a^b)^b = a */
a = a ^ b;   /* (a^b)^a = b */
```

> ***Ojo.*** *Es un truco* ***didáctico****, no una buena práctica: con la variable temporal el compilador hace lo mismo o mejor, y el XOR-swap* ***falla si*** `a` ***y*** `b` ***son la misma variable*** *(se anula a cero). Conocelo por cultura, no lo uses en producción.*

6.6 (Media) Rotar: correr sin perder bits

Con `<<` y `>>` los bits que salen se **pierden**. **Rotar** es un desplazamiento circular: lo que sale por un lado **entra por el otro**. No hay operador estándar (los procesadores tienen `ROL` / `ROR`, no C), así que se combina un `<<` y un `>>`:

```
rotar 8 bits a la izquierda en 1:
0b10000001  →  0b00000011   (el 1 que salió por la izquierda reaparece a la derecha)
```

C23 también lo trae en `<stdbit.h>`: `stdc_rotate_left(x, n)` y `stdc_rotate_right(x, n)`.

6.7 (Difícil) Endianness: ¿por qué `0x1234` no siempre se guarda igual?

Endianness (el "orden de los bytes") es cómo se **almacenan** los bytes de un número multi-byte en memoria:

- **Little-endian** (x86, ARM en su modo normal): el byte **menos** significativo va **primero**. El `uint32_t 0x12345678` se guarda como los bytes `78 56 34 12`.
- **Big-endian** (muchas redes, el protocolo TCP/IP): el byte **más** significativo va **primero**: `12 34 56 78`.

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint32_t n = 0x12345678;
    /* Miramos los bytes crudos del entero, uno por uno (unsigned char sí puede aliasar).
*/
    unsigned char *p = (unsigned char *)&n;
    for (size_t i = 0; i < sizeof n; i++) {
        printf("%02x ", p[i]);
    }
    printf("\n");   /* little-endian: 78 56 34 12  |  big-endian: 12 34 56 78 */
    return 0;
}
```

> ***Por qué te importa.*** *Si leés un `DWORD` de un archivo o de la red en una máquina del signo opuesto, los **bytes vienen al revés**. Por eso existen `htonl` / `ntohl` (host-to-network) y por eso los formatos de archivo **fijan** su endianness en la especificación (el BMP del capítulo `15` es little-endian). Los bits **dentro** de un byte, en cambio, no cambian: el desorden es **entre bytes**.*

6.8 (Difícil) `bitfields`: dejar que C haga las máscaras por vos

En vez de escribir `>>` y `&` a mano, C tiene una forma declarativa de empaquetar campos dentro de un `struct`: los **campos de bits** (*bit-fields*). Después de los `:` va cuántos bits tiene el campo.

```c
#include <stdio.h>

/* Un struct de 1 byte (según el compilador) repartido en 3 campos. */
struct Color5 {
    unsigned r : 5;   /* 5 bits: 0..31 */
    unsigned g : 6;   /* 6 bits: 0..63 */
    unsigned b : 5;   /* 5 bits: 0..31 */
};

int main(void) {
    struct Color5 c = { 12, 40, 20 };
    printf("r=%u g=%u b=%u\n", c.r, c.g, c.b);
    printf("sizeof = %zu\n", sizeof c);   /* típicamente 4 (con relleno) */
    return 0;
}
```

El lado oscuro de los bitfields. *El* ***orden*** *de los bits dentro del* `struct` ***no está garantizado*** *por el estándar: cambia entre compiladores y plataformas. Además, meten relleno (capítulo* `12`*) que arruina el cálculo. Por eso,* ***para leer un formato binario externo usá*** `>>` ***y*** `&`*, no bitfields. Los bitfields sirven cuando el "paquete" es* ***interno*** *y no sale nunca de tu programa.*

6.9 (Difícil) Uniones para reinterpretar bits: *type punning*

Una **unión** (`union`) hace que varios tipos **compartan la misma memoria**. Escribís como uno y leés como otro: eso se llama *type punning*.

```c
#include <stdio.h>
#include <stdint.h>

union Word {
    uint32_t full;          /* vista como número de 32 bits */
    uint8_t  bytes[4];      /* la MISMA memoria vista como 4 bytes */
};

int main(void) {
    union Word w;
    w.full = 0x11223344;
    /* en little-endian, w.bytes[] = { 0x44, 0x33, 0x22, 0x11 } */
    printf("%02x %02x %02x %02x\n", w.bytes[0], w.bytes[1], w.bytes[2], w.bytes[3]);
    return 0;
}
```

> ***Con cuidado.*** *Reinterpretar por unión es* ***dependiente de la implementación*** *(funciona en GCC, Clang y MSVC, pero el estándar no lo garantiza al 100%). La forma* ***segura*** *de reinterpretar bytes sin violar el aliasing es* `memcpy` *, con* `unsigned char` *(como en 6.7). Conocelo, pero preferí* `memcpy` *para código serio.*

Práctica de la Parte 6

Ejercicio 17.11 — La caja de herramientas de bits

Conceptos · `x & (x-1)` , potencia de dos, `x & -x` , popcount, tipos sin signo

Contexto · Los trucos de bits son herramientas sueltas que conviene tener probadas. Simplificación: una librería de funciones que recibe un número y responde preguntas sobre sus bits.

Consigna · Implementá, con `uint32_t` :

1. `countBits(x)` : cuántos bits en `1` tiene (contalos recorriendo con `x &= x - 1`);
2. `isPowerOfTwo(x)` : verdadero solo si `x` es una potencia de dos (`1, 2, 4, 8, …`);
3. `lowestBit(x)` : el **valor** del bit encendido más bajo (con `x & -x`);
4. `highestBit(x)` : la **posición** del bit encendido más alto.

Probá cada una con varios valores, incluido `0` y un valor con un solo bit (por ejemplo `1u << 31`).

Entrada · Varios valores de 32 bits. **Salida** · La respuesta de cada función para cada valor.

Escenario de prueba

```
x = 0b10110100   (180)
  countBits   = 4
  isPowerOfTwo= no
  lowestBit   = 4        (0b100)
  highestBit  = 7        (bit 7 = 128)

x = 0b01000000   (64)
  countBits   = 1
  isPowerOfTwo= sí
  lowestBit   = 64
  highestBit  = 6

x = 0
  countBits   = 0
  isPowerOfTwo= no
  lowestBit   = 0
```

Identificadores sugeridos · `countBits` , `isPowerOfTwo` , `lowestBit` , `highestBit` **Criterios de aceptación**

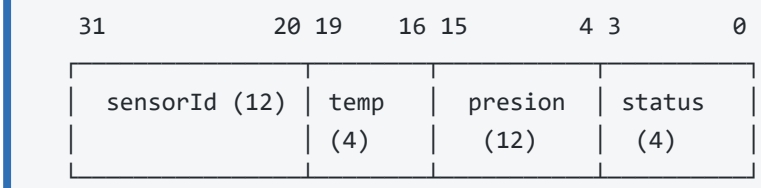
- [ ] Todo se hace con operadores de bits y tipos **sin signo**.
- [ ] `isPowerOfTwo(0)` devuelve **falso** (ojo con el `0`).
- [ ] El `1u << 31` se maneja bien (no aparece como negativo).
- [ ] Los resultados del escenario salen exactos.

Ejercicio 17.12 — El ejercicio jefe: un archivo de paquetes binarios

Conceptos · todo lo anterior — `uintN_t`, máscaras, `<<`, `>>`, `&`, `|`, endianness, archivos

Contexto · Un sistema de telemetría escribe cada medición como un **DWORD** de 32 bits donde se empaquetan varios campos. Para que este ejercicio entre en el capítulo, podés mantener los registros en un **arreglo** en memoria (sin archivos); si querés, guardalos con `fwrite` (capítulo `15`) como extra.

Consigna · Definí el siguiente registro de 32 bits:



1. Escribí una función que **empaquete** los cuatro campos en un `uint32_t`.
2. Escribí funciones que **extraigan** cada campo.
3. Cargá un arreglo con **al menos cinco** mediciones (con valores a elección, incluyendo `temp` negativa codificada en 4 bits con complemento a dos, y ambos extremos de `sensorId`).
4. Mostrá cada medición con sus cuatro campos **desempaquetados**.
5. Añadí una función que **cuente cuántas mediciones** tienen el `status` con el bit 0 encendido.

Entrada · Un arreglo de registros `uint32_t` (a elección). **Salida** · Cada registro en hex con sus campos, y el conteo final.

Escenario de prueba

```
registro = 0x054A4A32
  sensorId = 0x054       (84)
  temp     = 0xA         (4 bits, complemento a dos = -6)
  presion  = 0x4A3       (1187)
  status   = 0x2

... (cuatro mediciones más)

mediciones con status bit0 = 3
```

> ***Cómo comprobar el hex del escenario:*** *`sensorId << 20` = `0x05400000`, `temp << 16` = `0x000A0000`, `presion << 4` = `0x00004A30`, `status` = `0x00000002`. Con `|`: `0x054A4A32`. Y `0xA` en 4 bits con signo es `-6` (complemento a dos, Parte 1).*

Identificadores sugeridos · `TelemetryRecord`, `packRecord`, `getSensorId`, `getTemp`, `getPressure`, `getStatus`, `countWithBit0` **Criterios de aceptación**

- [ ] Las máscaras se construyen con `(1u << w) - 1`, nunca "a ojo".
- [ ] `temp` se extrae como número **con signo** de 4 bits (ver Parte 1) y puede dar negativo.
- [ ] Todos los campos se recuperan sin pisarse entre sí.
- [ ] El conteo del `status & 1` es correcto.
- [ ] Se usa `uint32_t` (el bit 31 no es signo).

17.13 Errores comunes

- **Confundir `&` con `&&`** (y `|` con `||`): los primeros operan bit a bit; los segundos, sobre la verdad completa.
- **Testear una bandera con `==`** en vez de `&`.
- **Confundir "alguna" con "todas"** al testear un grupo: `flags & GRUPO` vs `(flags & GRUPO) == GRUPO`.
- **Desplazar por el ancho o más** (`x << 32`): comportamiento indefinido.
- **Desplazar a la derecha un `signed` negativo**: dependiente de la implementación.
- **Operar bits con tipos con signo**: usá `unsigned`.
- **Enteros multi-byte crudos**: acordate de la **endianness** al leer archivos o redes.
- **Bitfields para formatos externos**: el orden de bits no es portable; usá `>>` y `&`.

17.14 Cierre

Bajaste un piso y ya no ves números: ves **filas de interruptores**. Con cuatro operadores (`&` , `|` , `^` , `~`) y dos desplazamientos (`<<` , `>>`) tenés todo lo necesario para:

- meter **ocho banderas en un byte** (y 32 en un `DWORD`) en vez de ocho `bool` ;
- **testear, prender, apagar e invertir** cada bandera con una máscara;
- **multiplicar y dividir** por potencias de dos corriendo bits;
- **leer y escribir campos** dentro de un `uint32_t` (`BYTE` / `WORD` / `DWORD` / `QWORD`) con el par `>>` y `&` ;
- entender por qué el mismo `0xFF` es `255` o `-1` , y por qué un `0x1234` no siempre se guarda igual (endianness).

Regla mental. Cada vez que veas un montón de datos que son "sí/no" repetidos, un conjunto de opciones, o un formato binario con campos, preguntate: **"¿esto es un número, o es una caja de bits?"**. Si es una caja de bits, ya sabés abrirla.

Resumen de la caja de herramientas.

Quiero…	Escribo…
Fabricar la máscara del bit `n`	`1u << n`
Prender el bit `n`	`x |= (1u << n)`
Apagar el bit `n`	`x &= ~(1u << n)`
Invertir el bit `n`	`x ^= (1u << n)`
Testear el bit `n`	`x & (1u << n)`
Máscara de `w` bits	`(1u << w) - 1`
Extraer un campo	`(x >> s) & ((1u << w) - 1)`
Armar un campo	`x |= (v & ((1u << w) - 1)) << s`
Multiplicar por 2^k	`x << k`
Dividir por 2^k (trunca)	`x >> k`
Apagar el bit más bajo	`x & (x - 1)`
Aislar el bit más bajo	`x & -x`
¿Potencia de dos?	`x && !(x & (x - 1))`
Contar bits en 1 (C23)	`stdc_count_ones(x)`

> ***El puente al capítulo siguiente.*** *Ahora que ves los bits, el mundo binario se te abre: leer imágenes (BMP, capítulo `15`), escribir compresores, entender protocolos de red y hasta mirar cómo un `float` guarda su signo, su exponente y su mantisa. Todo eso son* ***máscaras y desplazamientos*** *aplicados a `uintN_t`. Lo difícil ya lo tenés.*